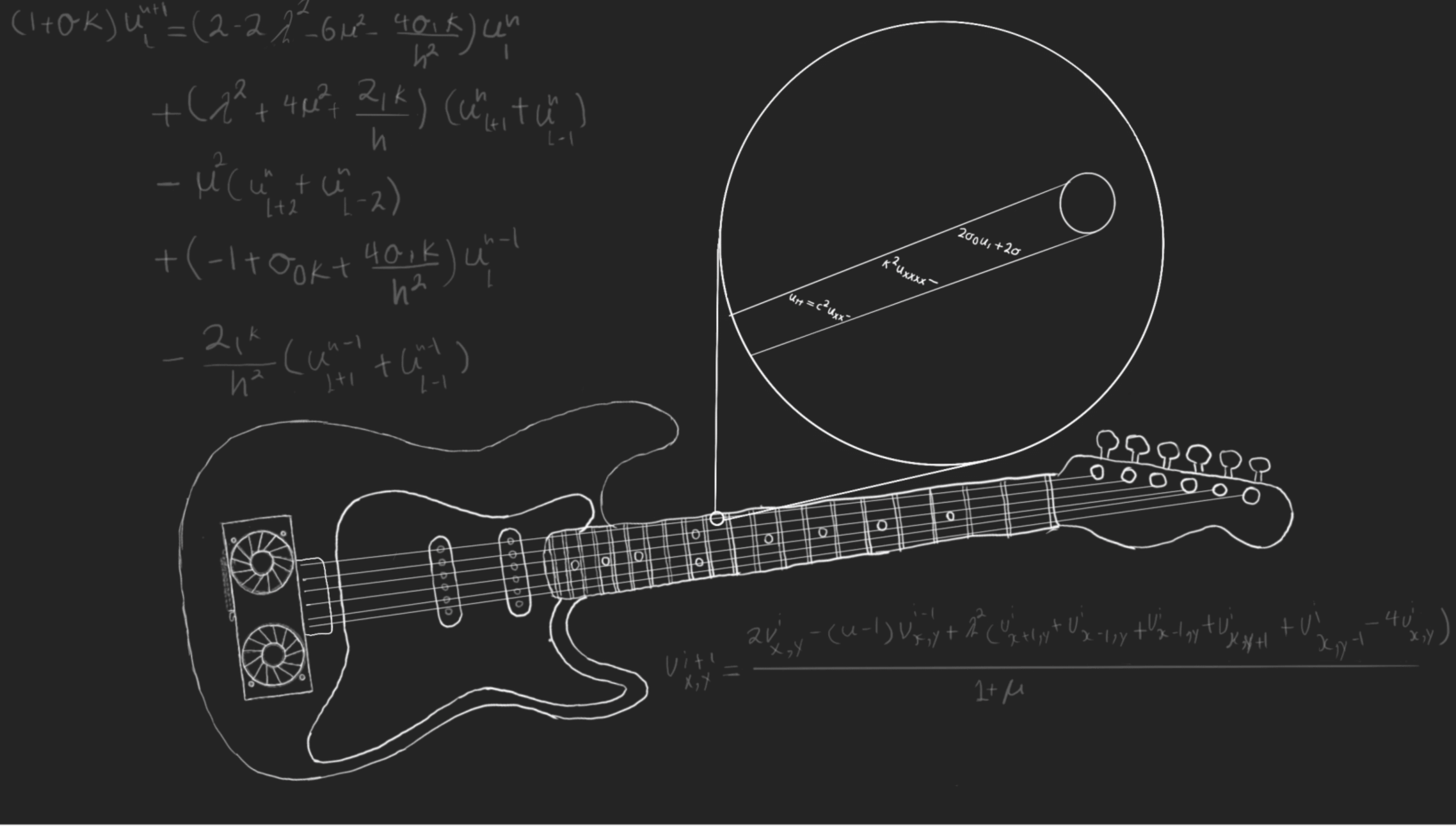

$(1+\sigma k)u_l^{n+1} = (2-2\lambda^2-6\mu^2-\frac{4\sigma_1 k}{h^2})u_l^n$
$+(\lambda^2+4\mu^2+\frac{2_1 k}{h})(u_{l+1}^n+u_{l-1}^n)$
$-\mu^2(u_{l+2}^n+u_{l-2}^n)$
$+(-1+\sigma_0 k+\frac{4\sigma_1 k}{h^2})u_l^{n-1}$
$-\frac{2_1 k}{h^2}(u_{l+1}^{n-1}+u_{l-1}^{n-1})$
$u_{tt} = c^2 u_{xx} -$
$\kappa^2 u_{xxxx} -$
$2\sigma_0 u_t + 2\sigma$
$V_{x,y}^{i+1} = \frac{2V_{x,y}^i - (\mu-1)V_{x,y}^{i-1} + \lambda^2(V_{x+1,y}^i + V_{x-1,y}^i + V_{x-1,y}^i + V_{x,y+1}^i + V_{x,y-1}^i - 4V_{x,y}^i)}{1+\mu}$

EVALUATING THE GRAPHICS PROCESSING UNIT FOR DIGITAL AUDIO SYNTHESIS AND THE DEVELOPMENT OF HYPERMODELS

UWE Bristol | University of the West of England

EVALUATING THE GRAPHICS PROCESSING UNIT FOR DIGITAL AUDIO SYNTHESIS AND THE DEVELOPMENT OF HYPERMODELS

Word Count: 42955

Harri Renney

A thesis submitted in partial fulfilment of the requirements of
the University of the West of England, Bristol for the degree of
Doctor of Philosophy in Computer Science

Faculty of Environment and Technology, University of the West of England, Bristol

December 16, 2022

Lee Richard Renney

Abstract

The extraordinary growth in computation in single processors for almost half a century is becoming increasingly difficult to maintain. Future computational growth is expected from parallel processors, as seen in the increasing number of tightly coupled processors inside the conventional modern heterogeneous system. The graphics processing unit (GPU) is a massively parallel processing unit that can be used to accelerate particular digital audio processes; however, digital audio developers are cautious of adopting the GPU into their designs to avoid any complications the GPU architecture may have. For example, linear systems simulated using finite-difference-based physical model synthesis is highly suited for the GPU, but developers will be reluctant to use it without a complete evaluation of the GPU for digital audio. Previously limited by computation, the audio landscape could see future advancement by providing a comprehensive evaluation of the GPU in digital audio and developing a framework for accelerating particular audio processes.

This thesis is separated into two parts; Part One evaluates the utility of the GPU as a hardware accelerator for digital audio processing using bespoke performance benchmarking suites. The results suggest that the GPU is appropriate under particular conditions; for example, the sample buffer size dispatched to the GPU must be within 32 to 512 to meet real-time digital audio requirements. However, despite some constraints, the GPU could support linear finite-difference-based physical models with 4× higher resolution than the equivalent CPU version. These results suggest that the GPU is superior to the CPU for high-resolution physical models. Therefore, the second part of this thesis presents the design of the novel HyperModels framework to facilitate the development of real-time linear physical models for interaction and performance. HyperModels uses vector graphics to describe a model's geometry and a domain-specific language (DSL) to define the physics equations that operate in the physical model. An implementation of the HyperModels framework is then objectively evaluated by comparing the performance with manually written CPU and GPU equivalent versions. The automatically generated GPU programs from HyperModels were shown to outperform the CPU versions for resolutions 64x64 and above whilst maintaining similar performance to the manually written GPU versions. To conclude part 2, the expressibility and usability of HyperModels is demonstrated by presenting two instruments built using the framework

List of Publications

Listed below are all publications authored and co-authored by the PhD student during the PhD project. These are ordered by their appearance in the thesis. The letters will be used in the thesis to identify which publication is being referenced clearly.

[A] Renney, H., Gaster, B. R. Mitchell, T. J. (2020b), ‘There and back again: The practicality of GPU accelerated digital audio’.

[B] Renney, H., Gaster, B. Mitchell, T. J. (2022), ‘Survival of the synthesis — gpu accelerating evolutionary sound matching’, Concurrency and Computation: Practice and Experience.

[C] Renney, H., Gaster, B. R. Mitchell, T. (2019), ‘OpenCL vs: Accelerated finite-difference digital synthesis’, in ‘Proceedings of the International Workshop on OpenCL’.

[D] Renney, H., Gaster, B. R. Mitchell, T. J. (2022), ‘HyperModels - A Framework for GPU Accelerated Physical Modelling Sound Synthesis’.

Contents

Chapter 1

Introduction

Since the 1960's, the reported processing power from microprocessors has been doubling every two years, primarily because of the transistor density improving. The frequently quoted Moore's law predicted this trend would continue and has held up as a reliable forecast ((Moore et al., 1965) & (Moore, 1998)). However, since 2010, manufacturers have been reporting increasing difficulties in maintaining this rate of growth (Ahmed and Schuegraf, 2011). As a result, the traditional methods for improving computing power are becoming less effective, as described illustriously by (Sutter, 2005) in "The free lunch is over". The primary factor affecting computing power growth is the approaching hard limit on capacitor density, along with other issues affecting clock speeds, like overheating (Denning and Lewis, 2017). In Figure 1.1 it can be seen that around 2005, even with the continually increasing transistor density, the clock speeds were already plateauing.

In the past, when processing power was increasing at a reliable rate, it was often acceptable to wait for the development of more powerful single-threaded hardware. Considering the expected continual decline in computational growth, alternative methods of improving processing power need to be explored. Recently, hardware manufacturers have developed multi-threading and multi-core architectures to improve compute power by processing data in parallel. As software support is initially immature and sub-optimal, the transition to multi-core architectures requires the necessary software to support the performance scaling expected of the novel architectures. This challenge is often referred to as the programmability-specialisation trade-off. Therefore, software must be developed that synergises with parallel architectures correctly. Multi-core parallel processors have a long history dating back to the 1960s Solomon computer (Slotnick et al., 1962); but it was not until the late 1990s to early 2000s that they were ready to be adopted for mainstream systems (Blake et al., 2009).

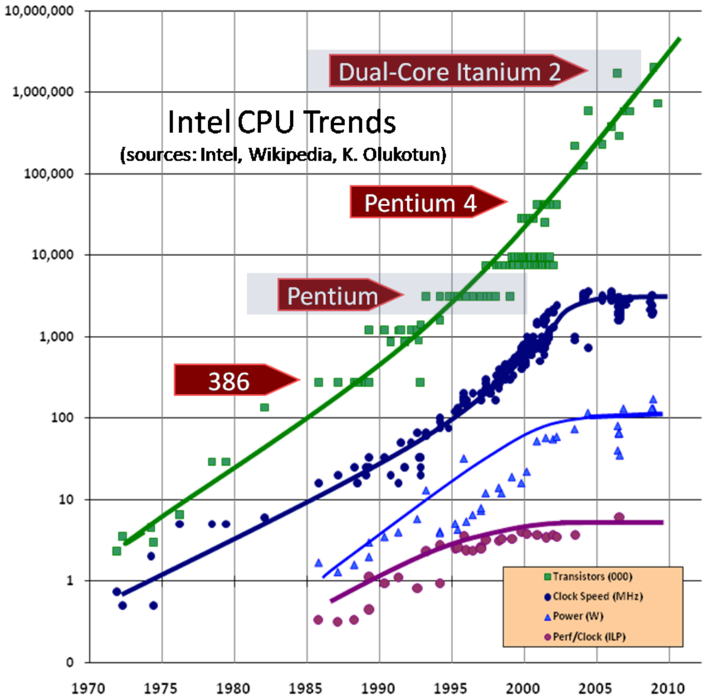


Figure 1.1: Intel CPU Trends 1970 - 2010. From, Sutter (2005)

Operating systems schedule applications across the cores and threads, improving overall performance considerably using task parallelism (Reinders, 2007). Task parallelism handles the simultaneous processing of separate independent programs, but because of the overhead involved with sharing memory and synchronisation, it does not scale to handle large amounts of data (Gaster et al., 2012). Data parallelism is another approach that uses hundreds of processors to execute the same instructions across significant amounts of data. Graphics processing units (GPU) are based on data parallelism and achieve massive parallel processing and increase the data throughput considerably. Data parallel architectures require data to be independent of one another, but as a result, avoid the overhead of memory sharing and synchronisation. As the name suggests, GPUs have been the de facto hardware accelerator in graphics processing as the architecture is intrinsically suited for graphics, leading to GPUs being fully adopted for almost all graphics processing (Blythe, 2008).

Interestingly, data parallelism is not exclusively restricted to graphics processing; many problems outside the graphics domain are suitable. Moreover, it is becoming increasingly feasible to offload suitable tasks to GPUs for considerable performance scaling. Utilising each device in parallel for processing the most suitable task is known as heterogeneous computing. A well-known and successful example of this is the *folding@home* project (Desell et al., 2010), where speedups of 20−30× were reported on the GPU. Even highly critical evaluations of heterogeneous computing, such as Lee et al. (2010), concluded there was an average of 2.5× speedup across a range of appropriate processes. In 2006, NVIDIA (Lindholm et al., 2008) released the Tesla GPU, introducing a new unified architecture that supports general compute for supporting processing outside of graphics. Contemporaneously, AMD made similar developments towards a unified architecture with the TeraScale GPU. NVIDIA and AMD have continued to develop comprehensive, general-purpose GPU (GPGPU) support in all subsequent GPU architectures (McClanahan, 2010), making GPUs an accessible hardware accelerator (Owens et al., 2007). Along with GPGPU hardware, APIs and frameworks are being developed for programming general compute on GPUs in applications. The most popular and widely used are NVIDIA's proprietary CUDA (Luebke et al., 2006), which is only supported on NVIDIA devices. Alternatively, an open standard managed by Khronos called OpenCL is designed to allow any platform and device vendor to support it. Further, it allows other special-purpose hardware to be used for general compute, not just graphics units.

GPGPU is an essential component in the future advances of heterogeneous computing; therefore, these frameworks have seen continued development, making the GPU architecture increasingly more programmable. Furthermore, with the growth of single-core clock speed plateauing, software must adapt to support the state-of-the-art parallel architectures within a heterogeneous environment.

In the 1950s, digital audio synthesis was beginning to gain traction with components such as digital oscillators, filters and stored lookup tables (Moore, 1979; Bristow-Johnson, 1996), these were used to generate sound, and later, to build synthesis techniques including AM and FM (Chowning and Bristow, 1986). These simple techniques are often highly computationally efficient (Smith, 1997) in order for them to be used in real-time applications at times when the available computation was very limited by today's standards (Bilbao, 2009, p. 3). Furthermore, these methods are considered ‘abstract’ as they do not directly associate with a physical interpretation. Physical modelling methods contrast with abstract synthesis as they are built on direct interpretation of physical phenomena. Although a broad field of physical modelling methods exists, the direct numerical physical models are the most authentic forms that directly simulate vibrations through a discretised mathematical representation (Berg and Stork, 1990). Direct numerical physical models simulate an environment by approximating vibration values through N-dimensional space and time. Increasing the resolution of the simulated space has several advantages, including: improved accuracy, more stable simulations and the space to create more sophisticated instruments. However, increasing the resolution proportionally increases the computation required to run a simulation. Considering the strict real-time requirements of audio synthesis (Lavry, 2004) (Jack et al., 2018), the usefulness of these methods has been heavily restricted. Nevertheless, with the modern advancements in computer systems, physical modelling synthesis is seeing a possible resurgence (Webb and Bilbao, 2015). For example, many academics involved in the Next Generation Sound synthesis (NESS) collaboration project believe physical models will play an important part in the future developments of sound synthesis. In 2015 and 2016, the NESS project published dozens of papers and demonstrations related to physical modelling audio synthesis and processing (NESS, 2019), including thorough experimentation and discussions of utilising GPU acceleration for physical modelling sound synthesis (Bilbao et al., 2019; Hsu and Sosnick-Pérez, 2013; Bilbao and Webb, 2012). The collective output from the NESS project highlights the benefit of increasing data throughput using the GPU. For example, in Hamilton and Webb (2013), GPU acceleration was used to improve offline processing of room acoustics and, was shown to improve performance by 46× over the equivalent serial CPU version. However, they also address the issues of processing various physical modelling techniques in parallel - particularly the incompatibility of specific mathematical methods for parallelisation (Bilbao et al., 2013). For instance, iterative methods like the Newton Raphson (Bilbao et al., 2014) for solving implicit schemes inherently involve serial stages that limits some of the parallel capability of the GPU. The considerable literature on GPU accelerated physical modelling is mainly used for advanced, non-linear systems that are processed offline. However, simple linear systems are well suited for the GPU parallel architecture and for meeting real-time performance. There is a gap in the literature here to comprehensively explore the application of GPUs for real-time physical modelling synthesis for linear systems. Furthermore, recent designs and implementations of physical models have been presented within the context of the CPU; for example, (Willemsen et al., 2021, 2020; Willemsen, Bilbao and Serafin, 2019; Onofrei et al., 2021; Willemsen, Andersson, Serafin and Bilbao, 2019*a*; Thibault, 2019), which are often restricted to one-dimensional or low resolution two-dimensional models. Research such as Zappi et al. (2017) demonstrates that the GPU can be used to support higher resolution physical models with significantly greater scale than on the CPU (zappi, 2019). Zappi's proposed design requires extensive knowledge of not only physical modelling synthesis, but the graphics domain and GPUs. Therefore, removing the complications of the GPU architecture and automating the GPU acceleration of physical modelling could expand the designers' accessible synthesis sound space. This thesis proposes the HyperModels framework for facilitating the development of GPU accelerated physical modelling instruments. This framework aims to provide a means of describing physical models in a high-level form that is automatically translated into the optimised GPU low-level equivalent code.

1.1 Research Targets

This thesis explores the idea of using the GPU as a hardware accelerator within the field of digital audio processing. The thesis begins by evaluating the objective performance of the GPU for offline and real-time requirements, comparing results to the CPU. Then, using the newfound understanding of the GPU, the strengths and weaknesses are used to design a GPU accelerated physical modelling synthesis framework for simple linear systems.

This thesis evaluates the GPU as a digital audio processing device, particularly with an application for real-time physical modelling synthesis. To clarify the contributions of the thesis, it has been split into two parts. Part 1 aims to evaluate the effectiveness of the GPU for digital audio in general. This leads into Part 2, where a GPU accelerated physical modelling framework called HyperModels is proposed.

1.1.1 Part 1

Part 1 contains three chapters. The first Chapter 3 evaluates the GPU in the domain of digital audio in general, using a benchmarking suite of performance profiling tests. This chapter highlights key strengths and limitations of the GPU and what parameters are critical to utilising it to best effect. Next, with a revised understanding of the GPU, Chapter 4 proposes the design for an offline, GPU accelerated sound matching application. Finally, observing the impressive results from the GPU for physical modelling synthesis, Chapter 3 comprehensively compares and contrasts the CPU and GPU for finite-difference based physical modelling synthesis. As a whole, Part 1 aims to answer the following research questions:

- Question_1 - What is the acceptable range of audio buffer lengths on the GPU for supporting real-time audio requirements for digital musical instruments? (Chapter 3)
- Question_2 - How does the chosen GPU interfacing software effect the performance of audio processing? (Chapter 3, 4 and 5)

- Question_3 - How do the data transfers rates between integrated and discrete GPUs effect the overall performance of audio processing? (Chapter 3 and 4)
- Question_4 - Can the GPU be used to accelerate an offline sound matching application that involves a combination of evolutionary algorithms and advanced FM synthesis? (Chapter 4)
- Question_5 - How does the performance of GPUs compare to CPUs when scaling the resolution of finite-difference based physical model synthesisers? (Chapter 5)

The answers to these research questions, while being intrinsic contributions on their own, lead into and inform the direction of the second part of the thesis to develop a high-level framework for describing physical model based instruments.

1.1.2 Part 2

Part 2 proposes the HyperModels framework - a solution for describing linear finite-difference based physical model instruments that are translated into optimised, low-level GPU programs. HyperModels proposes using a DSL for defining the physics equations that simulate models and a vector graphics-based approach for describing the shape of the models. Part 2 aims to explore the following research questions:

- Question_6 - Can a high-level description of a physical model instruments be translated into optimised low-level code that utilises the parallel processing capabilities of the modern GPU to achieve real-time audio synthesis. (Section 7.2.1)
- Question_7 - Can physics equations for physical modelling synthesis be defined using a domain-specific language? (Section 7.2.2)
- Question_8 - Can the geometry of a physically modelled environment be described with vector graphics? (Section 7.3)
- Question_9 - How does the performance of automatically generated GPU accelerated physical modelling synthesis programs compare to manually written equivalents? (Chapter 8)

The HyperModels framework proposed in Part 2 will then be used to develop two example real-time instruments outlined as:

- The Hyper Drumhead based on the GPU accelerated design proposed by Zappi et al. (2017) (Section 9.1)
- String-Plate Connection based instruments based on the designs presented by Willemsen, Andersson, Serafin and Bilbao (2019*a*) (Section 9.2)

Chapter 2

Background

This chapter contains the prerequisite background knowledge required to understand and appreciate the thesis's content and contributions fully. Four broad fields are covered in the following Sections: digital audio, numerical physical modelling, general-purpose graphics processing units and programming languages.

2.1 Digital Audio

Digital audio is often considered as a sub-field of digital signal processing (Pohlmann, 2000), where audio waves are represented as a signal composed of a sequence of samples through time. Each sample measures the displacement of air pressure at a given point in time. As each sample is considered, the pressure value constantly changes according to the signal that simulates pressure vibrations. The human auditory system perceives the sound as pressure vibrations through physical mediums such as air.

2.1.1 Digital Signal Processing

Digital signal processing (DSP) is the application of digital computer systems for signal processing. In reality, signals are continuously measurable sequences of some quantifiable phenomena. For example, the average temperature of a room over time can be considered a signal. However, due to the nature of digital systems, they must represent signals with a finite number of points and values with a fixed degree of precision (Broesch, 2008). This means a digital system must discretise the signal between equal spaces in time and quantise/approximate the accuracy of the values at each discrete point. Fortunately, modern systems can represent signals with an acceptable number of points and precision for most DSP problems. The sequence of discrete values taken from a continuous signal are often referred to as samples and are acquired or generated through time at a set sample rate. The sample rate is the number of samples used to represent a signal over time (Weik, 2012) and is measured in Hz, the number of samples per second.

Special hardware known as digital signal processors are specifically designed for processing digital signals efficiently. For example, in Thompson (2000), they are used to reduce the power consumption of medical devices by efficiently processing signals. However, digital signal processors are limited by their design to specifically process signals. For this reason, they have not been exposed as accessible programming devices as they do not support enough general-purpose processing.

2.1.2 Musical Oscillators

The oscillator was one of the first digital sound synthesis components to be used, with literature dating back to the 1960s with Risset (1965) and Freedman (1967). An oscillator is an operator that produces a waveform with a fundamental frequency and a peak amplitude. Often, when considering frequency in terms of oscillator functions, the SI unit of angular frequency Radians per second (rad/s) is used instead of Hertz (Hz) for trigonometric functions such as $\sin(\theta)$ and $\cos(\theta)$ (Ifeachor and Jervis, 2002). Using radians has the benefit of representing circular rotation, which suits the mathematical descriptions of musical oscillators. One cycle of oscillation is 1Hz and is equivalent to 2π rad/s. Therefore, when using the oscillator functions that expects frequency ω as rad/s, the frequency f in Hz can be mapped using $f = \frac{\omega}{2\pi}$.

The characteristics that give a waveform its timbre are frequency and the shape of the waveform (De Poli and Prandoni, 1997). Different oscillators can be defined that produce different waveforms and have parameters for controlling the shape and frequency of oscillation. An exact solution for a single sinusoidal oscillator can be described using an exact method:

$$u(t) = A\cos(\omega_0 t + \phi) \tag{2.1}$$

Where t is the independent variable time, and A, ω_0, ϕ are the amplitude, frequency and initial phase. An example of a sinusoidal waveform generated from Equation (2.1) is shown in Figure 2.1. Oscillators were immediately adopted as key musical synthesis components because of the perceptual significance they have with the human auditory system. Ultimately, every sound synthesis method mentioned in this text, including the

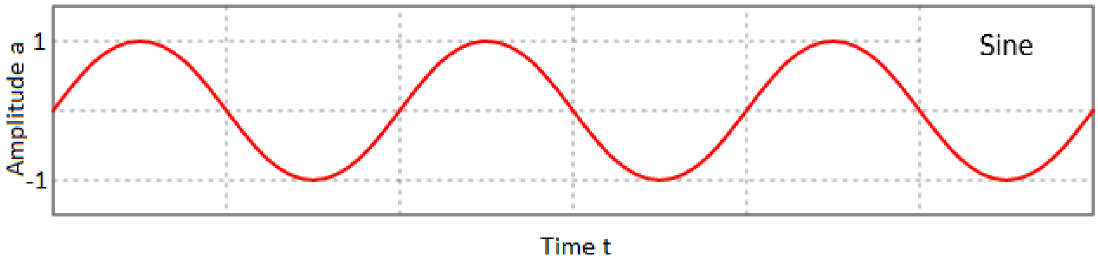


Figure 2.1: Output form sinusoidal operator Equation (2.1) across time t.

intricate physical models, is an oscillator.

2.1.3 FM Synthesis

One of the first commercially available and successful digital audio synthesis methods was Frequency Modulation (FM) synthesis (Chowning and Bristow, 1986). Invented by John Chowning, FM for audio synthesis was discovered and initially developed in the 1970s (Chowning, 1973). The technique was adopted by Yamaha in the DX and TX synthesisers that were commercially produced throughout the 1980's (Fukuda, 1985) and is still a prevalent technique used in many plugins and synthesisers that are currently available (Albano, 2016). It is regarded as a highly efficient method for generating complex and rich audio timbres with simple graphs of interconnected sinusoidal oscillators. The original simple FM synthesis equation is defined as:

$$y(t) = A\sin(ct + I\sin(mt)) \tag{2.2}$$

Where four parameters are exposed: peak amplitude A, carrier frequency c (rad/s), modulation frequency m (rad/s) and modulation index I. The function y generates an instantaneous output for FM at a given time t. The modulation frequency m sets the frequency of the modulating oscillator, the output of which is multiplied by the modulation index I to control the intensity of the frequency modulation applied to the carrier oscillator. This is then added to the input for the carrier oscillator's frequency c. Finally, the variable A is used to control the peak amplitude output. When the modulation index $I = 0$, the modulation oscillator has no effect. When $I > 0$, the modulation oscillator begins to affect the carrier oscillator, and symmetrically spaced intervals of frequencies occur above and below the carrier frequency. The number of side frequencies relates to the modulation index; therefore, as I increases, energy is taken from the carrier frequency and distributed further across the sideband frequencies. Bessel functions determine the amplitudes of the carrier and side frequencies (Chowning and Bristow, 1986). FM synthesis has a nonlinear mapping between parameters and the resulting sound (Roth and Yee-King, 2011), this means that there is a non-trivial correlation between a synthesiser's parameter space and the resulting timbre space, i.e., minor changes to the input parameters can generate vastly different changes in the resulting sound. This can make it a challenging synthesis method to navigate using the exposed parameters. This has led to the development of optimisation tools that map sounds back to the FM parameters that produce them (Mitchell, 2020).

2.1.4 Buffering

Buffering is a commonly used technique in computing to significantly improve the performance of processing and data transfers. Buffering uses a region of memory to temporarily store data while moving from one place to another. For example, in the case of an input microphone, the sequence of audio samples is collected temporarily in a buffer and processed when the processor is available. This contrasts with the method of processing each sample one at a time, as it arrives (Orfanidis, 1995, Chapter 4). Buffering improves performance by taking advantage of simultaneous data transfers and processing. When a data buffer has been processed, it can be moved out of memory to the output audio playback device. Then, the following audio buffer can be processed while the audio is still being transferred from the main memory to the device. Contrasting this with a sample-by-sample approach, buffering instead uses the time waiting on memory transfers to process more samples. The time it takes for a buffer of some length N to be processed is known as the audio buffer period and is important as it has effects on the real-time suitability of a process.

2.1.5 Offline & Real-time Processing

Digital audio processing that does not need to meet any immediate time requirements is referred to as an offline process. An example of this is generating a distribution pack of audio samples for various different digital instruments. The distributions do not need to be immediately used, but instead can take hours or even days to process and then store as samples in the distribution pack. The samples can then be played in real-time when loaded from the distribution pack into memory. However, numerous audio processes and applications depend on meeting real-time requirements that can consistently process audio samples as time advances. For example, an instrument used in live performance must consistently operate fast enough to avoid negatively impacting the performance. Therefore, real-time audio is the requirement to consistently meet a set of time limited requirements for a particular application.

The requirements vary between applications (Annett et al., 2014), where variable amounts of data must be processed within a fixed and inflexible time frame. In the case of real-time audio, a consistent number of audio samples needs to be produced every second to avoid aliasing. The real-time requirement in audio is stringent, as even a few missed samples or delays results in instantly noticeable 'glitches' in the perceived sound (MacKenzie and Ware, 1993; Meehan et al., 2003). To measure a process' ability to sustain this real-time performance, the audio-sound latency will be approximated using the audio buffer period described in Section 2.1.4. For a process that must produce samples at a chosen sample rate, the process must be able to produce the buffer length of samples within the time window determined by the fraction of a second a buffer can take for the sample rate. This means that the maximum acceptable audio buffer period for a real-time process is $p_s = \frac{b_s}{r_s}$ where p_s is the maximum acceptable audio buffer period, r_s is the chosen sample rate and b_s is the chosen buffer length. Using this formula, the maximum real-time audio buffer period at 44.1KHz for all of the profiled audio buffer lengths used in this thesis are presented in Table 2.1. If the processing of the audio buffer period exceeds the maximum acceptable period from Table 2.1, it is expected that the process will fail to maintain real-time performance and will produce buffer drops outs that result in audible pops and clicks that are unacceptable for using an audio process in real-time.

The core real-time performance metric using the *audio-sound latency* against the audio buffer period validates the suitability of an audio process for producing samples in real-time. However, in this thesis one of the objectives outlined in research question 1 is to evaluate the suitability of the audio processes for use in digital musical instruments (DMIs) that are suitable for performance without disruption. Therefore, a set of additional real-time sonic interaction requirements will be defined for evaluating audio processes. The real-time sonic interaction requirements depend on a metric known as the *action-sound latency* and *action-sound latency variation*. The *action-sound latency* is the delayed auditory feedback between tactile interaction to the sound output. Delayed auditory feedback between tactile-sound interaction has been shown to be disruptive to musical

Audio Buffer Length	Maximum Audio Buffer Period (ms)
1	0.023
2	0.045
4	0.091
8	0.181
16	0.363
32	0.726
64	1.451
128	2.902
256	5.805
512	11.610
1024	23.220
2048	46.440
4096	92.880
8192	185.760
16384	371.519
32768	743.039

Table 2.1: The maximum acceptable real-time audio buffer periods for all audio buffer lengths at 44.1KHz.

performance as early as the 1960's in works such as Havlicek (1968) and Gates et al. (1974). *Action-sound latency variation* is the variability observed in the *action-sound latency* in the context of non-isochronous control and audio signals. Although this is a sophisticated process to evaluate in its entirety, a preliminary approach would be to take these requirements and compare them to the audio buffer period as well as the original *audio-sound latency* requirement. The exact range of real-time requirements for DMIs are still being discussed, but (Repp and Su, 2013), Lavry (2004) and Jack et al. (2018) have established the following ranges presented in Table 2.2 and these will be used in this thesis when evaluating sonic interaction for real-time performance.

Requirement	Recommended	Acceptable
Sample Rate	96000	44100
Action-Sound Latency	10ms	20ms
Action-Sound Latency Variation	±1ms	±3ms

Table 2.2: The chosen real-time sonic interaction audio requirements used in this thesis.

Meeting the audio-sound latency and real-time sonic interaction requirements is of prime importance for building compelling digital musical instruments. Within the context of this thesis, we propose to evaluate the real-time capability of a process if it fits into the following requirements: 1. The maximum audio sound latency to sustain acceptable real-time performance and avoid buffer underruns Table 2.1, and 2. The real-time audio sonic interaction requirements concerned with interaction using Table 2.2.

2.2 Physical Modelling Sound Synthesis

Physical modelling sound synthesis refers to the methods that generate audio using mathematical models that simulate the physical phenomena of acoustics. Physical modelling techniques can use alternative mathematical methods for simulating physical wave propagation through time and space, from which an audio waveform can be extracted. Various mathematical methods are available for physical modelling; analytical methods are exact and highly efficient but are usually limited to basic physical models that do not capture complex physical behaviours. Numerical methods are used for simulating advanced physics using approximation and discretisation, making them intrinsically suitable methods for computation. Finite-difference methods are a type of numerical method for solving differential equations by approximating derivatives with finite-differences (Thomas, 1995). In order to understand finite difference based physical models, foundational mathematics must be covered, starting with ordinary differential equations.

2.2.1 Simple Harmonic Oscillator

Ordinary differential equations (ODE) are differential equations containing one or more functions of one independent variable and the derivatives of those functions (Butcher and Goodwin, 2008). The simple harmonic oscillator is regarded as one of the essential ODEs in musical acoustics (Matteson, 2009) and even physics as a whole (Leach and Schei, 1993). Although sinusoidal oscillators have been used extensively in more abstract forms of synthesis, such as FM and subtractive synthesis, they also emerge in physical model representations (Peetre, 2000). This thesis will use the following notation for differentials (Ahmad and Ambrosetti, 2019):

$$u_x = \frac{\partial u}{\partial x}, \qquad u_t = \frac{\partial u}{\partial t}, \qquad x_{xx} = \frac{\partial^2 u}{\partial x^2}, \qquad u_{xt} = \frac{\partial^2 u}{\partial x \partial t}, \qquad \ldots \tag{2.3}$$

As differentials are abundant, this notation helps readability and conciseness. Using this notation, the following ODE for the simple harmonic oscillator is defined as:

$$u_{tt} = -\omega_0^2 u \tag{2.4}$$

where state variable u describes the displacement of the mass from its equilibrium (in m), u_{tt} is its acceleration and ω_0 is the angular frequency of oscillation (rad/s). The harmonic oscillator involves the second-order derivative u_{tt}; therefore, it requires two initial starting conditions, the state of u at $t = 0$ and the state of the first derivative of u at $t = 0$:

$$u(0) = u_0 \qquad \left.\frac{\partial u}{\partial t}\right|_{t=0} = v_0 \tag{2.5}$$

The simple harmonic oscillator emerges in mechanics and acoustics in mass-spring system (Wu and Chen, 2001) and electrical circuit theory (Chattopadhyay, 2006). ODEs are typically trivial to solve, and it is well known that the exact solution to Equation (2.4) is the sinusoidal operator discussed previously in Equation (2.1) (Leach and Schei, 1993). The concepts in ODE lead to partial differential equations; these are not as trivial to solve using exact methods as the simple harmonic oscillator. Therefore, numerical methods become an alternative solution that can handle the challenges of solving more advanced equations.

2.2.2 Partial Differential Equations

Partial differential equations (PDE) are an extension to ODEs where instead of a single independent variable, there is more than one in the equation (Evans et al., 2012). The 1-dimensional wave equation is an example of a partial differential equation:

$$u_{tt} = c^2 u_{xx} \tag{2.6}$$

where state variable u now involves two independent variables: time t as well as spatial coordinate x. Assuming a system of length L (in m), the spatial domain becomes $x \in [0, L]$. Furthermore, c is the wave speed (in m/s). The two independent variables t and x are both controlled and affect the dependent function u. An analytical method finds the exact solution for the given PDE for the specific boundary and initial conditions. Many PDEs do not have a known analytical solution, and some that are known are still not practical to solve analytically (Shampine, 2018, Chapter 1). Alternatively, numerical methods can be used to find an approximate solution (Kirby, 2009). To prepare differential equations for numerical method simulation, they must be mapped into a discrete domain; this process is known as discretisation.

2.2.3 Discretisation

To prepare PDEs for the finite-difference method, they must be mapped into the discrete domain (Har and Tamma, 2012, Chapter 13). The temporal and spatial domains are discretised by introducing uniformly partitioned time and space mesh (Langtangen, 2016*b*). The points in the time dimension are $t_n = n\Delta t$, where $n = 0, 1, \ldots, N$ is the number of time steps up to an indefinite value N and Δt is the constant size between the time steps. Points making up the space mesh are formed similarly and can be defined in the one-dimensional case by $x_l = l\Delta x$, where $l = 0, 1, \ldots, L$ is the number of spatial grid points and $\Delta x = \frac{X}{L}$ is the constant distance between points. When considering further spatial dimensions, if the space between spatial grid points is constant and uniform in all variables, it is often represented as h for finite differences, in this case, $\Delta x = \Delta y = \cdots = h$. Therefore for clarity, the timestep size will also be represented as the letter $k = \Delta t$. Visually, a one-dimensional example using this notation would result in a mesh as seen in Figure 2.2. In this example, the discretised independent variables are x and t; the value at each point is calculated by inputting the variables into the unknown function u_l^n using $\Delta t = 0.01$ and $h = 0.2$. Previously, the notation u is used for the function when defining differential equations. However, when transitioning to the discrete domain, the function must be described as a specific point in time and space using the notation u_l^n. The white points indicate that the unknown function u_l^n can be approximated using finite-differences using neighbouring values either side of the white point. However, the black points at the edge of the system do not have adequate neighbour information and must therefore be calculated using an alternative boundary condition $\Phi(l)$. Finite differences can be used to map the equation from a continuous/exact form into the discrete domain. With the right finite-differences, a recursively solvable system can be formed where the state of the system across all spatial steps x can be approximated at t from which the system can be incremented forward in time by Δt and used to calculate the next time step $t + \Delta t$.

2.2.4 Initial and Boundary Conditions

Many PDEs require either initial conditions (IC) or boundary conditions (BC) to establish the specific problem correctly. Conditions are a necessary part of making a problem well-posed, as described by Kahane (1991). A well-posed problem: has a solution that exists, the solution is unique, and the solution depends on the initial conditions in a continuous way. A problem that is not well-posed will have none or too many solutions leading to unpredictable errors (Sizikov et al., 2011). Therefore, to satisfy these properties, it

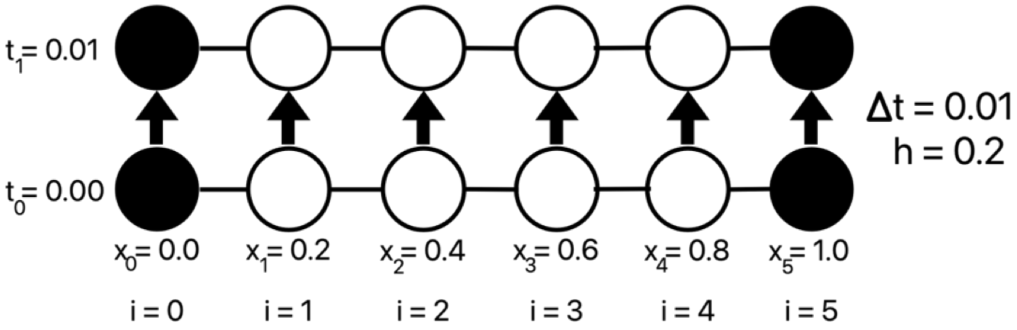


Figure 2.2: Example of a discrete space and time mesh for a single time-step.

is essential to define the right conditions to establish a well-posed problem that can be solved. Simple well-posed initial and boundary conditions are well understood for the heat diffusion equation in works such as in Mebrate (2015). For sound synthesis, more advanced and contextually interesting conditions are explored by (Willemsen, Andersson, Serafin and Bilbao, 2019*b*).

Initial conditions define the initial/starting state of the system, usually considered at $t = 0$. This condition specifies the values of all the spatial grid points at $t = 0$. Therefore, the initial condition can be defined as $u(0,l) = \Phi(l)$, where Φ is a function which generates initial values for a position. For audio synthesis applications, the initial condition for all points can be set to $\Phi(l) = 0$, setting all points to 0 at the beginning. After the simulation begins from the initial conditions, the models are then excited at later points by external forces referred to as excitations.

Boundary conditions define the values for the spatial variables at the edges of a system of equations through all steps in time (Zwillinger, 1998, Chapter 17). In the one-dimensional case, this requires two assignments at x_0 and x_L. These are $u(t,0) = \Psi(t,0)$ and $u(t,L) = \Psi(t,L)$ respectively, where Ψ is a function which generates values at the extremes ends of the spatial variable l for all points in time t. The simplest case is to set it to a clamped value, like $\Psi(t,0) = 0$ and $\Psi(t,L) = 1$, this is known as a Dirichlet boundary condition (John, 1941). Boundary conditions start to get interesting when they interact in a meaningful way with interior values in the mesh. For example, the boundary condition can be set as the derivative of the boundary. At $\Psi(t,0)$, the derivative can be set to $\frac{\delta x}{\delta t} = 0.5$, this is known as the Neumann condition. The Neumann condition can be used to reflect a portion of the energy back into the system. Other advance boundary conditions include the Robin condition (Ryan et al., 2010), which is a hybrid of the Dirichlet

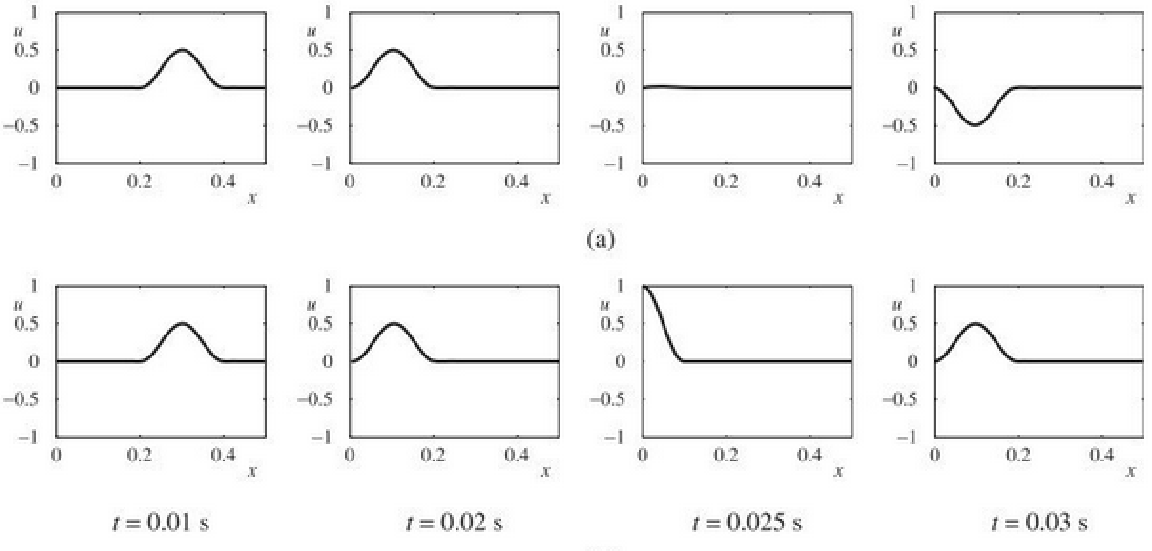


Figure 2.3: Reflection of left-travelling wave on boundary (a) with inversion using dirichlet condition and (b) without inversion using a Neumann condition. (Bilbao, 2009)

and Neumann conditions. Defining the boundary conditions in the discrete domain will be covered after finite-differences are introduced.

2.2.5 Finite Differences

Finite differences are used to replace the discrete derivatives in an equation so that they can be approximated numerically at each spatial and temporal point. Finite differences are derived from applying Taylor's theorem to a PDE, resulting in approximations at grid points (Folland, 2020). There are different forms of finite differences, including forward and backward differences; these determine the grid points taken into account. Forward differences comprise grid points that involve the current and next points in time. Backward differences use the previous and current grid points in time. The central difference involves the current, next and previous points in time. The formal definitions of all the relevant finite differences used in this thesis have been collected in Table 2.3 for the example of replacing derivatives with respect to x.

The choice of finite-difference affects the overall accuracy of the scheme. The growth of error (and therefore inaccuracy) of a finite-difference is described using big O notation and grows with respect to the variable step size h (Iwaniec and Kowalski, 2004). In algorithm analysis, big O notation describes how an algorithm's execution time or memory requirements grow as the input size grows. This method can be used in the context

Type	Notation	Derivative	finite difference
Forward	δ_{x+}	u_x	$\frac{u_{l+1}-u_l}{h}$
Backward	δ_{x-}	u_x	$\frac{u_l-u_{l-1}}{h}$
Central	$\delta_{x\cdot}$	u_x	$\frac{u_{l+1}-u_{l-1}}{2h}$
Centered Second-order	$\delta_{xx} = \delta_{x+}\delta_{x-}$	u_{xx}	$\frac{u_{l+1}-2u_l+u_{l-1}}{h^2}$
Discrete Laplacian	$\delta_{\Delta\boxplus} = \delta_{xx} + \delta_{yy}$	Δu	$\frac{u_{l+1,m}+u_{l-1,m}+u_{l,m+1}+u_{l,m-1}-4}{h^2}$
Discrete Biharmonic	$\delta_{\Delta\boxplus,\Delta\boxplus} = \delta_{\Delta\boxplus}\delta_{\Delta\boxplus}$	$\Delta\Delta u$	$\frac{1}{h^4}\begin{pmatrix} (\mathrm{u}_{l+2,m} + u_{l-2,m} + u_{l,m+2} + u_{l,m-2}) \\ +2(u_{l+1,m+1} + u_{l+1,m-1} + u_{l-1,m-1} + u_{l-1,m+1}) \\ -8(u_{l+1,m} + u_{l-1,m} + u_{l,m+1} + u_{l,m-1}) \\ +20 \end{pmatrix}$

Table 2.3: Lookup table containing all relevant finite-differences used in this thesis.

of finite differences to describe how the error introduced by approximating grows with the step size, h. $O(h)$ describes the error growth is proportional as the step size h increases. $O(h^2)$ grows at a rate of the input size h multiplied by h, an exponential growth. Therefore, for $O(h^2)$, the error decreases at a faster rate than $O(h)$, when $h < 1$. Forward and backward differences have an error growth of $O(h)$, while central differences have $O(h^2)$ (Chazarain and Piriou, 2011), meaning the central difference converges to 0 error at a faster rate. In sound synthesis applications, due to perceptual considerations, it is agreed that highly accurate models are rarely needed (Bilbao, 2009, Chapter 2 p. 33). This is because the temporal resolution is relatively high such that the sample rate is supported. With the high temporal resolution, the spatial resolutions can afford to be much lower. To work out the exact ranges of acceptable spatial, stability conditions are calculated.

Applications like acoustics that require a high temporal resolution are most suited for forming explicit finite-difference schemes. Explicit schemes calculate the state of a system at a later unknown time step using the state of the system from the current and previously known time steps (Ascher et al., 1997). This means that when replacing derivatives with finite-differences, only a single unknown point in the next time step $u^{n+1}_{...}$ should be in the equation. The unknown point can then be isolated on one side of the equation and all known points on the other side such that all the known points can be used to calculate the unknown point. The explicit scheme can then be solved recursively by repeatedly calculating the equation and stepping forward through time. By recursively solving the scheme, a simulation of the environment emerges.

2.2.6 Wave Equation

The wave equation is a core PDE that arises in various fields, including electromagnetics (Zubair et al., 2011), fluid dynamics (Durran, 2013) and particularly acoustics (Alford et al., 1974). Constraining the wave equation to 1-dimension is not a realistic representation on its own. However, it is often used as a test problem in numerical models or extended to incorporate more advanced physics. Furthermore, it operates with only a single spatial dimension, forming a physical model with significantly small computation and memory requirements. The 1-dimensional wave equation from Equation (2.6) is a second-order PDE that can be used naively as a first approximation for the transverse motion of strings (Gerver, 1970). When applied in this way, the wave speed $c = \sqrt{T_0/\rho A}$, where T_0 is the applied string tension, ρ is the string density, and A is the strings cross-sectional area. It can also be used to approximate the longitudinal motion of a uniform bar (Morse and Ingard, 1986, Chapter 5) where instead, $c = \sqrt{E/\rho}$ and uses an additional constant Young's Modulus E. The context of the equation can also be adjusted to approximate a wind instrument by modeling air vibrations through a tube of uniform cross-section by instead assigning $c = \sqrt{B/\rho}$, where B is the bulk modulus and the material density of ρ is the air density.

Using the finite-differences from Table 2.3, the differentials in Equation (2.6) can be replaced to form the explicit finite-difference equation:

$$\frac{u_l^{n+1} - 2u_l^n + u_l^{n-1}}{\Delta t^2} = c^2 \frac{(u_{l+1}^n - 2u_l^n + u_{l-1}^n)}{h^2} \tag{2.7}$$

Here, u_{tt} and u_{xx} have been replaced with a second-order finite-differences δ_{tt} and δ_{xx} respectively. This finite-difference scheme can now be re-arranged to form a recursive algorithm for simulating a model:

$$u_l^{n+1} = 2u_l^n + \lambda^2(u_{l+1}^n - 2u_l^n + u_{l-1}^n) - u_l^{n-1} \tag{2.8}$$

Where $\lambda = \frac{\Delta t c}{h}$, is the key coefficient that captures the timestep Δt, spatial step h and wave propagation speed c. The control of this coefficient determines if the simulation is stable. This recursive equation can now be updated explicitly at each time step n from previously computed values. The entire physical model can be simulated by continually updating values, storing them, and incrementing the timestep. Defining the boundary conditions for this one-dimensional equation can be done with the following form:

$$u_0^i = u_{N_x}^i = 0 \forall x, (Dirichlet, fixed) \tag{2.9}$$

$$\delta_x u_0^i = \delta_x u_{N_x}^i = 0 \forall x, (Neumann, free) \tag{2.10}$$

where two finite-difference points on either end of the one-dimensional system can be identified as boundary points for all points in time i by defining the index of spatial points at the first edge of the system 0 and then at N_x which is the number of spatial finite-difference points in the system and therefore the point at the far edge of the system opposite the index at 0.

The 1-dimensional models considered so far involve one temporal and one spatial variable. As further dimensions are added, the number of spatial variables increases. In the case of the wave equation, when it is extended to 2-dimensions, it becomes [1]:

$$v_{tt} = c^2(v_{xx} + v_{yy}) \tag{2.11}$$

Adding further dimensions adds second-order derivatives but for additional spatial dimensions. To concisely define these equations, the Laplace operator (LLC, 2021) can be used to describe the summation of the second-derivatives of all spatial variables as $\Delta u = u_{xx} + u_{yy}$. Therefore, the two-dimensional wave equation can be defined as:

$$v_{tt} = c^2 \Delta v \tag{2.12}$$

The explicit scheme for the two-dimensional equation can be formed by replacing the derivatives u_{tt}, u_{xx} and u_{yy} with second-order finite differences δ_{tt}, δ_{xx} and δ_{yy} respectively:

$$v_{l,m}^{n+1} = 2v_{l,m}^n - v_{l,m}^{n-1} + \lambda^2(v_{l+1,m}^n + v_{l-1,m}^n + v_{l,m+1}^n + v_{l,m-1}^n - 4v_{l,m}^n) \tag{2.13}$$

Another useful shorthand notation as difference equations start to involve higher derivatives (like linear plate equations) is to use the *biharmonic* operator:

$$\Delta\Delta u = u_{xxxx} + u_{xxyy} + u_{yyyy} \tag{2.14}$$

As covered, boundary conditions can be defined for 1-dimensional equations with $u_0^i = 0$ and $u_{N_x}^i = 0$ for clamped Dirichlet conditions. However, a comprehensive definition for the entire edge in a two-dimensional system needs to be formally defined. An example

[1] To improve consistency in this thesis, the notation u will be used for one-dimensional models and v for two-dimensional

of boundary conditions for a two-dimensional system, such as the wave Equation (2.13), can be defined using the following form:

$$\left.\begin{aligned} u^i_{0,y} = u^i_{N_x,y} = 0 \forall y, \\ u^i_{x,0} = u^i_{x,N_y} = 0 \forall x, \end{aligned}\right\} (Dirichlet, fixed) \tag{2.15}$$

$$\left.\begin{aligned} \delta_x u^i_{0,y} = \delta_x u^i_{N_x,y} = 0 \forall y, \\ \delta_y u^i_{x,0} = \delta_y u^i_{x,N_y} = 0 \forall x, \end{aligned}\right\} (Neumann, free) \tag{2.16}$$

Here, values across the edge of the system override the calculation of Equation (2.13) and instead set them to an alternative value, in the case above, to 0.

2.2.7 Excitation

Interacting with physically modelled instruments involves adding energy into the system. Spreading operators are used for adding a signal coherently into a centre position and its neighbouring points (Bilbao, 2009, Chapter 5 p. 101). In the continuous domain, the Dirac Delta function $\delta(x_i - x)$ (Hassani, 2009) identifies that at positions $x_i \in [0, 1]$, an excitation signal is input into the system according to a function that is defined when mapping to the discrete domain. Taking the 1-dimensional wave Equation (2.6) with the same parameters, it can be coupled to an excitation mechanism, giving:

$$u_{tt} = \gamma^2 u_{xx} - \delta(x_i - x)e \tag{2.17}$$

Here, the Dirac Delta function and a force function are appended onto the one-dimensional wave equation. The excitation function e can be any function that produces an excitation signal that can be inserted into the system. When transitioning to the discrete domain, the Dirac Delta operator is replaced with a spreading operator. The notation $J(x^n_i)$ is used to define spreading functions, the most basic being the zeroth-order spreading distribution that generates input at a single point x_i that is mapped to the nearest grid point to the left using $l_i = floor(x_i/h)$:

$$j_{l,0}(x_i) = \frac{1}{h} \qquad \text{for } l = l_i, \qquad \text{otherwise } 0 \tag{2.18}$$

Notice the spreading function effectively scales the excitation function e by $1/h$ when on an excitation point x_i. There are more advanced spreading operators used for more realistically modelling excitation distributions, such as linear and cubic operators that are elaborated on in (Bilbao, 2009, Chapter 5 p. 101). The discrete spreading component $J(x_i^n)$ can now be integrated into explicit finite-difference schemes. For example, continuing from the one-dimensional explicit scheme in Equation (2.19), the following scheme is formed:

$$u_i^{n+1} = 2u_i^n + \lambda^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) - u_i^{n-1} - j_{l,0}(x_i)E(t) \tag{2.19}$$

The scheme now has values generated from the discrete excitation function E into positions in the system through the spreading operator $j_{l,0}$.

2.2.8 Stiff String

The "ideal" string is often presented using the one-dimensional wave equation; it generates an output with harmonic partials that are integer harmonics of the configured fundamental frequency (Morse et al., 1948, Chapter 3). However, real strings exhibit dispersion because of the material stiffness. Stiffness can be modelled by adding an additional 4th-order derivative term to the one-dimensional wave equation, along with parameters for controlling material properties. The following PDE can be used to simulate the additional tension and stiffness resonating forces (Ducceschi and Bilbao, 2016):

$$\rho A u_{tt} = T u_{xx} - E I u_{xxxx} \tag{2.20}$$

where ρ is the material density, $A = \pi r^2$ and r are the cross-sectional area and radius of the string, T is the tension and E is the Young's modulus and the area moment of inertia is $I = \pi r^4/4$. By grouping a number of parameters into the following coefficients $c = \sqrt{T/\rho A}$ and $\kappa = \sqrt{EI/\rho A}$, the compact form of Equation (2.20) can be written as:

$$u_{tt} = c^2 u_{xx} - \kappa^2 u_{xxxx} \tag{2.21}$$

The 4th-order spatial derivative models stiffness causes desired wave dispersion through the string. This phenomenon causes higher frequencies to travel faster than lower frequencies. Furthermore, frequency dispersion also causes some inharmonicity effects. This results in a string equation that models a more realistic and interesting string and will be used in this thesis's more advanced physical model designs.

2.2.8.1 Multiple Strings

Once a single physical model like a string or bar has been created, it is possible to begin arranging multiple versions. Multiple models can be used to synthesise different notes for an instrument using different parameters and dimensions. Another interesting use of multiple models is employing several at once to synthesise a single note. A real example of this is found in pianos, where several strings are struck at once for each note. Various characteristics can be introduced by changing properties of each model - For example, unique tensions for each string can produce desirable beating effects .

To consider a collection of M uncoupled strings using the explicit scheme formed from the one-dimensional wave (Equation (2.19)) the following notation can be used (Bilbao, 2009, Chapter 7 p. 185):

$$u_{q,l}^{n+1} = 2u_{q,l}^{n} + \lambda^2(u_{q,l+1}^{n} - 2u_{q,l}^{n} + u_{q,l-1}^{n}) - u_{q,l}^{n-1} \quad \text{for} \quad q = 1, \ldots, M \tag{2.22}$$

Here, M is the number of strings and $u_{q,l}^{n}$ indicates the transverse displacement of the Mth string at discrete time point n and position l. This form can be extended to model piano strings, using a centre pitch of $f_0 = \lambda^q/2$ and detuning parameter D in cents as:

$$\lambda^q = 2^{1+\frac{(2q-1-M)D}{2400(M-1)}} f_0 \quad \text{for} \quad q = 1, \ldots, M \tag{2.23}$$

2.2.9 Ideal Linear Plate

Thus far, the two-dimensional wave equation has been covered for simulating a vibrating membrane for sound synthesis. However, materials with inherent stiffness properties are of further interest in musical acoustics (Hambric, 2006). The physics of vibrating plates are considered far more complex than that of a membrane described using the two-dimensional wave equation (Junger and Feit, 1986, Chapter 7). Although considered more complex, linear plate equations can result in less computation than two-dimensional wave equations (Bilbao, 2009, Chapter 7 p. 168). The Kirchoff thin plate (Morse and Ingard, 1986, Chapter 5 p. 213) equation for modelling a uniformly thin isotropic plate is defined as:

$$\rho H v_{tt} = -D\Delta\Delta v \tag{2.24}$$

where $D = EH^3/12(1-\nu^2)$ is a stiffness coefficient parameterised by Young's Modulus E (Wang, 1984), thickness H and the dimensionless Poisson's ratio ν. Being a linear equation, it does not hold for relatively thick plates and only accounts for low-amplitude vibrations. These constraints can be assumed in instrument simulations making the model sufficient for musical applications. Using a centred second-order δ_{xx} finite-difference to approximate u_{xx} and the discrete biharmonic $\delta_{\Delta\boxplus,\Delta\boxplus}$ for $\Delta\Delta u$, the following explicit scheme is formed:

$$\begin{aligned}\frac{\rho H}{k^2}(v^{n+1}_{l,m} - 2v^{n}_{l,m} + v^{-1n}_{l,m}) = -\frac{D}{h^4}((&v^{n}_{l+2,m} + v^{n}_{l-2,m} + v^{n}_{l,m+2} + v^{n}_{l,m-2})\\ &+ 2(v^{n}_{l+1,m+1} + v^{n}_{l+1,m-1} + v^{n}_{l-1,m-1} + v^{n}_{l-1,m+1})\\ &- 8(v^{n}_{l+1,m} + v^{n}_{l-1,m} + v^{n}_{l,m+1} + v^{n}_{l,m-1}) + 20)\end{aligned} \tag{2.25}$$

This scheme requires not only accessing the adjacent neighbouring spaces (like the one-dimensional wave equation), but neighbouring points two spaces in each spatial direction. Rearranging this explicit scheme for a suitably recursively solvable scheme leads to:

$$\begin{aligned}v^{n+1}_{l,m} = &-\frac{Dk^2}{\rho H h^4}((v^{n}_{l+2,m} + v^{n}_{l-2,m} + v^{n}_{l,m+2} + v^{n}_{l,m-2})\\ &+ 2(v^{n}_{l+1,m+1} + v^{n}_{l+1,m-1} + v^{n}_{l-1,m-1} + v^{n}_{l-1,m+1})\\ &- 8(v^{n}_{l+1,m} + v^{n}_{l-1,m} + v^{n}_{l,m+1} + v^{n}_{l,m-1}) + 20)\\ &+ 2v^{n}_{l,m} - v^{-1n}_{l,m}\end{aligned} \tag{2.26}$$

2.2.10 Damping Components

In the physical world, vibrating media lose energy and dampen out over time because of resisting forces from air viscosity (Sasajima et al., 2010) and thermoelastic effects (Serra and Bonaldi, 2009). Adding general damping attenuates all frequencies uniformly and can be modelled by adding a frequency-independent damping component $\sigma_0 u_t$ where σ_0 controls the intensity of the frequency-independent damping that emerges from u_t. Furthermore, real materials exhibit complex frequency dependant loss characteristics. Various frequencies do not decay at the same rate, and instead, the amount of attenuation increases with frequency. This leads to sounds with wide-band attacks such that the vibration decays to only a few particular harmonics. Modelling frequency-dependent loss in linear strings was used in Bensa et al. (2003) as a component defined as $\sigma_1 u_{txx}$ where σ_1 controls the intensity of the frequency-dependent damping that emerges from more involved $u_t xx$. The frequency-dependent and frequency-independent damping components can be added to the stiff string Equation (2.20) leading to:

$$u_{tt} = c^2 u_{xx} - \kappa^2 u_{xxxx} - 2\sigma_0 u_t + 2\sigma_1 u_{txx} \tag{2.27}$$

2.2.11 Connections

Modelling more advanced musical instruments begins with interactions between individual resonators. Thus far, the physical models are described as being formed by a single model describing one physics equation which will now be referred to as a resonator. By forming two or more resonators inside the modelled environment, these can be connected using connection techniques to form complex instruments that exhibit non-linear behaviour. Connections apply an external force to two separate positions between two or more physics equations. Like the excitation methods discussed in Section 2.2.7, these use Dirac Delta functions $\delta(x-x_i)$ in the continuous formulation of the equations to describe connection force from one equation being added to a position in another equation. An example of two one-dimensional wave equations u and w with domains $x \in D_u$ and $\chi \in D_w$ connected at $x_c \in D_u$ to $\chi_c \in D_w$ yields the following PDEs:

$$\begin{aligned} u_{tt} &= c_u^2 u_{xx} + \delta(x - x_i)\frac{f}{\rho_u A_u} \\ w_{tt} &= c_w^2 w_{\chi\chi} - \delta(\chi - \chi_i)\frac{f}{\rho_w A_w} \end{aligned} \tag{2.28}$$

where ρ and A are each string's density and cross-sectional area is each string [2] and f is the connection force, which must be equal and opposite for the two connected systems according to Newton's third law (note the inverse signs between both equations) (Hellingman, 1992). The definition of f depends on the connection type used and determines the behaviour between the interacting resonators. These are then transformed into the discrete domain with the following form:

$$\begin{aligned} u_l^{n+1} &= \cdots + J_{l,u}(x_i)\frac{F^n}{\rho_u A_u} \\ w_m^{n+1} &= \cdots - J_{m,w}(\chi_i)\frac{F^n}{\rho_w A_w} \end{aligned} \tag{2.29}$$

Here, the definition of the force function can take the form of a rigid connection. A bidirectional rigid connection between two resonators is defined as $u(t,x) = w(t,\chi)$, where x is the connection point in the first resonator u and χ is the connection point in the second resonator w. Following the definition of the discrete rigid connection defined in Willemsen

[2] Recall $c = \sqrt{T/\rho A}$

(2021) where zeroth-order spreading operators are used, the following connection force function is formed:

$$F^n = \frac{c_u^2 \delta_{xx} u_l^n - c_w^2 \delta_{\chi\chi} w_\chi^n}{\frac{1}{\rho_u A_u h_u} + \frac{1}{\rho_w A_w h_w}} \tag{2.30}$$

Within the context of this thesis, the connection forces will not advance beyond the rigid connections using zeroth-order spreading operators shown in Equation (2.30). The physical modelling methods covered in this section are fundamental to understanding this thesis's instrument designs and implementations. The following section will cover the parallel processing paradigms and architectures of the graphics processing unit used for processing these physical models in parallel.

2.3 General Purpose Graphics Processing Units

The finite-difference schemes covered in the previous section are naturally suited for the data-parallelism of the GPU and are often referred to as "embarrassingly parallel" problems (Moler, 1986). Therefore, this section covers the GPU's parallel processing paradigms and architectures and how they are used to process the physical models in parallel, starting with their position in the heterogeneous environment.

2.3.1 Heterogeneous Computing

Modern computer systems are now built as heterogeneous platforms (Terzo et al., 2019) that promote the simultaneous use of different processing devices to maximise efficiency. Heterogeneous computing aims to accelerate overall performance by simultaneously processing tasks on the most suitable hardware available (Khokhar et al., 1993) (Yu et al., 2009). Usually, the CPU takes the role of the controller/host, managing the other hardware devices. Therefore, the CPU manages the allocation of tasks to other devices that process the task and return the results to the CPU once complete. At the time of writing, CPUs typically have at least 4 fast cores[3], and may have dozens[4] or even hundreds of cores in high-end server CPUs[5]. These cores are usually reserved for most general pro-

[3] https://www.intel.co.uk/content/www/uk/en/products/sku/217187/intel-core-i71195g7-processor-12m-cache-up-to-5-00-ghz/specifications.html

[4] https://www.amd.com/en/products/cpu/amd-ryzen-9-pro-3900

[5] https://www.tachyum.com/datasheets/Prodigy%20PB%2016128%20v1.0_220510.pdf

cessing whilst the modern GPU contains hundreds[6] to tens of thousands[7] of specialised parallel processors that can be used to offload and accelerate suitable tasks like graphics and physical models.

Heterogeneous computing has become a foundational concept in the continual development of computational growth as it supports the inclusion of various processing devices like the GPU. The tools and frameworks supporting heterogeneous computing have continuously matured in academia and industry such that now they can be included in designs and implemented on modern systems (Wen-mei, 2015). A widely adopted open-source standard and framework for heterogeneous computing is The Open Computing Language (OpenCL). OpenCL offers an abstract programming model for devices supporting OpenCL. This means that a single OpenCL program can be written and run across various systems, which then utilises all supported heterogeneous devices available. (Gaster et al., 2012) describes in detail the abstract model and how to efficiently program OpenCL to offload and accelerate particular processes to heterogeneous devices, like GPUs.

Whilst heterogeneous computing is used to increase computing power, a further benefit is that it can lead to more efficient power consumption. In (Che et al., 2009, p. 8), the heterogeneous implementations utilizing the GPU were observed to consume more power than the CPU versions. However, the power consumption to performance ratio was much better for the GPU versions in all but one case across the nine benchmarks. This thesis explores the design of physical modelling audio synthesis within the modern heterogeneous environment to promote further compute power and energy efficiency.

2.3.1.1 Processing Units

Processing units are a collection of hardware components that collectively enable the execution of program instructions (Iwai and Ohmi, 2002). There is a variety of processing unit designs and architectures, the most fundamental being the central processing unit (CPU). The (CPU) is the primary processor for most computer systems (Baer, 1980), it handles the processing of arithmetic, logic, control and input/output operations. CPUs can be made up of many fast processing cores, each core in the range of 3.50 to 4.2 GHz. Although initial CPUs would use a single core and execute programs in sequence, modern CPUs involve multi-core and multi-threading for parallel processing (Duncan,

[6] https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1050-ti/specifications

[7] https://www.nvidia.com/en-gb/geforce/graphics-cards/30-series/rtx-3080-3080ti

1990). Using Flynn's Taxonomy, this parallel processing environment on modern CPUs is referred to as a Multiple-Instruction Multiple-Data format (MIMD) (Flynn, 2011).

Graphics processing units (GPU) are a special kind of processing unit originally designed for graphics processing (Hopgood et al., 1986). At its core, the GPU utilises the Single-Instruction Multiple-Data (SIMD) processing format (Flynn, 1972). Although initially designed for accelerating the processing of image data for display devices, the SIMD format used by the GPU is suitable for many other types of problems. Instead of building a new specialist SIMD device, developers utilise the GPU for accelerating appropriate SIMD processes because of the commercial success and wide adoption of the GPU now present in nearly all systems.

In Danalis et al. (2010), a benchmarking suite comprehensively tests and compares a range of GPUs and CPUs. The GPUs demonstrated superior processing throughput when compared to the CPUs. However, the GPUs did not always meet the ideal throughput calculated according to their specification. This is a common challenge when mapping problems into a GPU environment, the target is to reach the theoretical performance according to the specification of the hardware. However, as is usually the case, there is overhead for most problems, including memory transfers, synchronisation and under-utilised resources, but by following the vendor best practices (Allalen et al., 2017), performance can be maximised. The performance implications of the GPU and the low-level programmability harms its adoption; therefore, one of the core aims of this thesis is to highlight the strengths and limitations of the GPU within the context of digital audio.

2.3.2 Parallelism

Task parallelism is a format of parallel processing where several different tasks that can have unique sets of instructions are executed simultaneously. This enables each task to independently execute on its own thread or processing core without relying on separate tasks (Bocchino et al., 2009). However, often tasks need to share data, requiring data sharing techniques to be employed (Chatterjee et al., 2013). An example of task parallelism is pipelining (Cosnard and Trystram, 1994). Pipelining works by breaking a problem into sections that can be processed independently and passing their resulting data to the next stage in the pipeline. In addition, task parallelism usually has coarse-grained features, meaning it has less communication overhead between parallel elements; thus, the ideal scenario results in close to negligible overhead (Gordon et al., 2006).

Data parallelism is the processing of a collection of data elements at the same time by applying the same sets of instructions to all the data (Reinders, 2007). Data parallelism is incorporated into the CPU as additional vector processors that are designed to execute the same instructions efficiently and effectively large one-dimensional arrays of data (Eichenberger et al., 2004). Some examples of these vector processors include SSE (Raman et al., 2000) and the more modern AVX (Lomont, 2011) instructions sets. These can be accessed using low-level intrinsic functions provided by the hardware vendor (like Intel (Hassan et al., 2016)) or mapped to high-level interfaces like OpenMP (Chandra et al., 2001). The GPU achieves massive data parallelism using numerous data-parallel processors, often referred to as streaming multiprocessors. Data parallelism usually has fine-grained features, meaning it has a high communication overhead. Therefore, thread communication should be avoided when targeting data parallelism. GPUs combine the fine-grained SIMD paradigm of data parallelism with the coarse-grained features of thread parallelism by arranging multiple streaming multiprocessors resulting in the Single-Instruction Multiple-Thread paradigm (Kilgariff and Fernando, 2005).

2.3.3 GPU Architecture

An abstract view of the modern GPGPU architecture is shown in Figure 2.4 (Weber, 2014). Here, the CPU interfaces with the GPU unit across a *bridge* and a *host interface* loads instructions and programs onto the GPU (Nickolls and Dally, 2010). The device then loads the program across the compute devices; these manage the execution of instructions from the program across their numerous Processing Elements (PE). According to the program instructions, all the PEs then execute the same instructions simultaneously on different sections of data in memory by using the ID of the stream of execution to index into memory. This means that each compute unit supports the single-instruction multiple-data (SIMD) paradigm. Note that the compute device has multiple compute units that can each have different sets of program instructions loaded onto them. This extends the SIMD paradigm to single-instruction Multiple-thread (SIMT) (Corporation, 2009) and this arrangement enables the massively parallel processing environment of the GPU. Each compute unit can apply the same sets of instructions in lock-step to data across all its PEs. The lock-step execution blocks diverging branches from executing simultaneously on a compute unit and negatively impacts performance. However, as there are multiple compute units providing a SIMT arrangement, the GPU can handle some branching but must avoid excessive branching. In this paper, the term *core* will be used to refer to streams of execution on each PE and not the physical cores. This

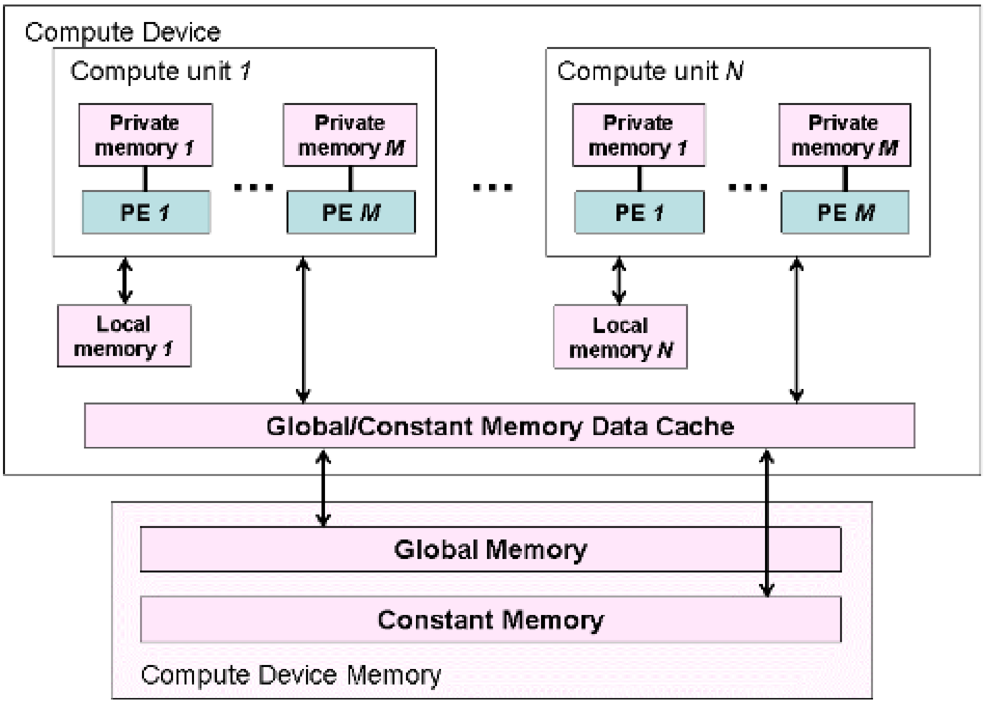


Figure 2.4: The modern GPU architecture (Weber, 2014).

abstraction helps describe how data is processed fully in parallel, but in reality, all of this processing might not execute simultaneously if the number of items to process exceeds the number of physical cores.

2.3.4 GPU Programming

The GPU can be programmed in two ways; the original approach is to program the GPU using the graphics pipeline. The second approach is to use general compute shaders. The graphics pipeline consists of a series of primary operations that process raw vertex data into a final presentable image (khronos, 2017). Figure 2.5 presents the stages of the graphics pipeline[8]. First, a collection of points describing 3D geometry called vertices are used to form primitives such as triangles in the vertex shader. The space between triangles is then rasterised to form a collection of fragments that can then be processed into coloured pixels in the fragment shader. The frame of pixel data is then sent to a display device. Graphic shaders are programmed in languages supported by the GPU standards, such as the OpenGL Shading Language (GLSL) for OpenGL (Marroquim and

[8] Modern GPU program can involve more advanced stages and programs in the pipeline

Maximo, 2009).

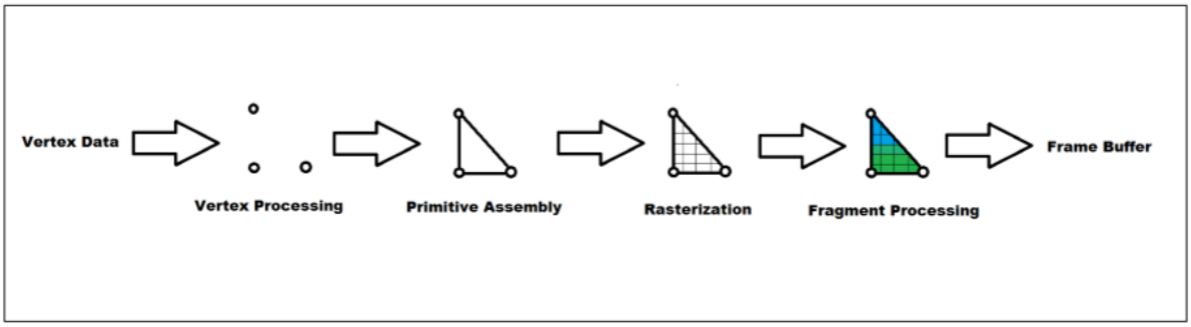


Figure 2.5: Fundamental graphics rendering pipeline stages.

With the development of the unified architecture from Figure 2.4, an alternative approach using compute shaders can be used instead of the graphics pipeline. This approach uses one or more general compute shaders to describe instructions for the PEs on the compute device to process data elements loaded from memory simultaneously. The following relevant GPU standards support general compute shaders:

OpenCL - An open standard for parallel programming of heterogeneous systems where general-compute shaders are describe using the OpenCL kernel language (Trevett, 2013).

CUDA - Parallel computing API for interfacing GPUs as GPGPU units. The GPU is programmed as CUDA kernels written in the CUDA programming language Sanders and Kandrot (2010).

OpenGL - OpenGL is primarily a 3D graphics rendering API for interfacing with the GPU using the graphics pipeline and graphics shaders written in GLSL.

Vulkan - Vulkan is the next-generation 3D graphics rendering API. Known for its absolute control and low-level intricacies, Vulkan exposes the GPU to its core. Vulkan supports custom pipelines that can be arranged from graphical shaders and GPGPU compute kernels that are written in GLSL.

The various GPU interfacing standards use their own notation for describing GPGPU, this thesis will conform to use the OpenCL programming model (Group et al., 2013)(Gaster et al., 2012) (Corporation, 2009). Figure 2.6 provides a view of the abstract OpenCL model; notice how this maps almost directly into the modern GPU architecture shown in Figure 2.4. Each PE executes as a *workitem*, which has access to its own region of fast access private memory. All *workitems* are contained within a *workgroup*, which is essentially a compute unit managing its workitems. All *workitems* in a *workgroup* have access to a shared region of local memory. Local memory is slower than the private memory accessible to each *workitem* but is faster than the global memory that all *workitems* can access. The host memory is associated with the CPU and must be transferred to the GPU over system buses to fill the global memory and then processed and moved to other regions of GPU memory. This memory hierarchy is similar to that found on the GPU, where memory types vary in size and access speed (Hestness et al., 2014).

Besides memory visibility between PEs, the GPU also involves different levels of visibility for the CPU acting as the host. There are three common types of memory in the relationship between the CPU and GPU (Gregg and Hazelwood, 2011). GPU local memory is only accessible on the GPU, and all data that is processed on the GPU should be stored here. Host visible memory is memory on the GPU accessible by the CPU but is not necessarily accessible by PEs on the GPU. Host visible memory is slower for the GPU to process and therefore should be used as a staging area to transfer data from the CPU to the GPU. Finally, there are push constants (Larsen, 2019), these represent high-speed ways to write constant memory to a limited memory space. Therefore, push constants are suitable for improving the update speed of constants that do not require considerable amounts of memory.

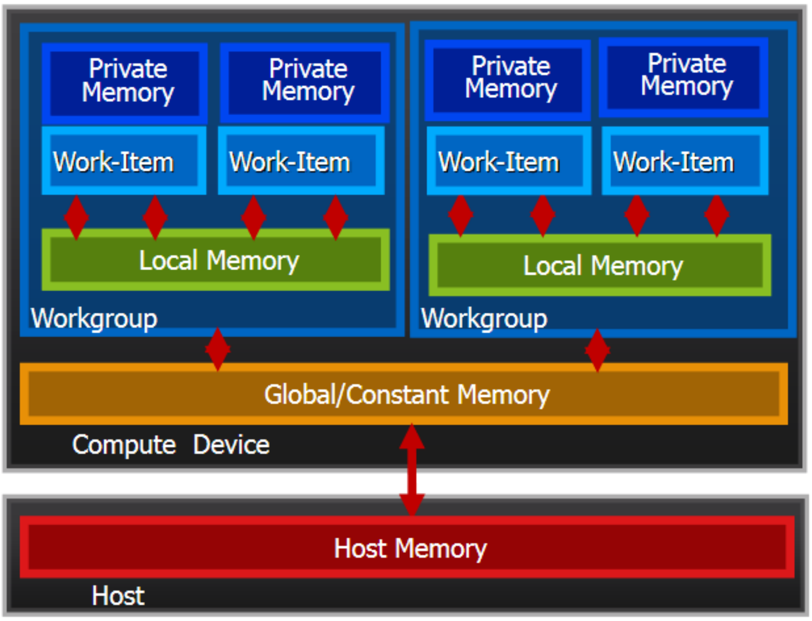


Figure 2.6: The abstracted OpenCL memory model used when interfacing with OpenCL. (Tompson and Schlachter, 2012)

Adhering to the OpenCL model, it can be used to process data in parallel by dispatching workitems to the GPU and executing a program that uses the ID of each workitem to index into memory, process and store the calculations. An example demonstrating the mapping of a list doubling function from a serial approach to an OpenCL parallel form is given as follows:

Algorithm 1 Vector Double List

```
function serialDouble(aList, aListSize)
    for $i$ = 1 to aListSize do
        aList[i] *= 2
function vectorisationDouble(aList, aListSize)
    idxWorkitem = getWorkItemID()
    idxGroup = getWorkGroupID()
    stride = idxGroup
    for $i$ = 1 to aListSize when i += stride do
        aList[i + idxWorkitem] *= 2
```

Here, a list of data is processed in parallel using the workitem index *idxWorkitem* and loop index *i*. The data is accessed this way to ensure memory coalescing by having each workitem accessing data adjacent to each other. This improves the access speed of global memory as a compute unit fetches a block of data once for all adjacent workitems to access, instead of accessing memory in entirely different locations in memory that would require fetching data from multiple regions of memory. Failure to arrange programs for memory coalescing is detrimental to the overall performance. The contrasting memory access patterns gather and scatter have notoriously poor performance. Gather and scatter operations require threads to access completely different chunks of memory, often resulting in long memory accessing overheads that should be avoided and removes the performance benefits from memory caching. Gather and scatter operations are formally described in (Duff et al., 2002) with the following linear algebra definitions:

Gather	$x \leftarrow y\|_x$	$x(i) = y(idx(i))$
Scatter	$y\|_x \leftarrow x$	$x(idx(i)) = y(i)$

Figure 2.7: Linear algebra and mathematical equations for gather & scatter operation.

In Figure 2.7, the gather operation is described as a process that iterates through every element in the lists x, y. 'i' is used to access the data elements in sequence. 'idx(i)' is used to access a random element of y. Understanding this, it can be seen that random values are taken from y, and stored in sequence in x. Hence, this is called gathering. The scatter operation is described as the values in sequence in y are stored at random positions x; essentially the inverse of gathering.

In He et al. (2007), the authors study the gather and scatter operations on GPUs and compare them to equivalent CPU applications. Their work shows that a naive scatter-gather is approximately 33.8 times slower than a sequentially accessed benchmark. The authors highlight that it is best to avoid gather and scatter operations entirely. However, if a designer has no choice, like with sparse linear algebra operations, there are optimisations. For example, by using a multi-pass scheme for radix sort, it is seen to achieve a 30% improvement to a naive implementation on the GPU. Additionally, in comparison to an Intel CPU, the GPU application achieved a 7.2× speedup for a hash search. These results highlight the importance of memory coalescing and accessing memory optimally. Processing finite-difference models provide an optimal data structure to operate on as gather and scatter operations can be avoided. For example, the recursively solvable explicit schemes can map each workitem to an adjacent element in memory and only access neighbouring values to that element in memory.

2.3.5 GPU Type

GPUs are often classed as one of two types, discrete or integrated GPUs. Discrete GPUs are separate from the CPU and connected across system buses such as Peripheral Component Interconnect (PCI). However, communicating over the PCI buses involves data transfer overhead. In contrast, integrated GPUs are tightly coupled to the CPU, even sharing regions of physical memory known as unified memory. This means that the latency overhead accumulated over PCI bus transfers is avoided, and communication between CPU and GPU is faster than the discrete GPU (Renney et al., 2020). However, integrated GPUs are limited by the physical space and power they can consume and therefore typically provide an inferior data throughput than discrete GPUs. This makes integrated GPUs more suitable for processing minor data-parallel problems, while discrete GPUs are for processing tasks requiring higher data throughput.

2.4 Programming Languages

Programming languages are notations for describing computation for humans and machines. There are many different fundamental types of programming languages, and researchers have explored this since the 1950s when the first major programming language FORTRAN was established (Backus, 1979). Human-readable source code must be converted into a form that digital systems can execute. Programming languages have a defined set of syntax that a programmer uses to describe the instructions to execute; this ultimately takes a semantic meaning that takes effect when the program executes. A compiler is a tool that translates a program's source code into another target language. A compiler is made up of two distinct parts: the front-end parser and the back-end code generator. Among the many roles of a compiler, one of the most important is reporting and describing errors in the source code that prevent compilation.

Based on the description by Aho et al. (2020), the stages of a minimum compiler are shown in Figure 2.8. First, the input of the front-end parser takes the high-level source code comprised of a series of human-readable characters called a character stream. Next, the character stream undergoes lexical analysis, where the code is scanned, and meaningful sequences of characters are grouped into output tokens. Next, the syntax analyser uses the output tokens to build an abstract syntax tree (AST), capturing the meaning and structure of the program. Finally, the back-end code generation component uses the AST to generate the equivalent target code. This is can be low-level machine code for execution on particular hardware or another high-level, human-readable source code that can then be compiled by another compiler for execution.

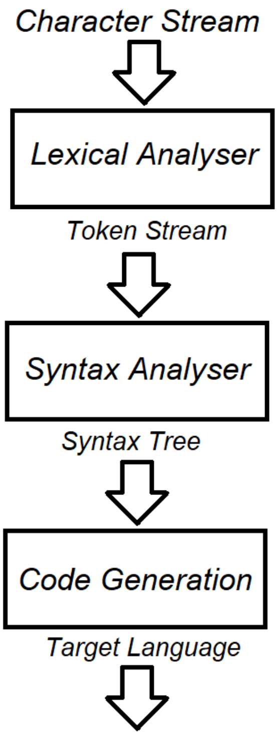


Figure 2.8: The stages involved in a minimal compiler.

2.4.1 Functional Programming

Functional programming is one of the two major programming paradigms available to programmers, the other being imperative programming. Imperative programming languages achieve results by executing statements that explicitly change the system's state; in effect, the programmer is telling the computer 'how' to do something. In contrast, functional programming is considered a declarative paradigm. Declarative programming is considered a more abstract way of programming where the computation is described without explicitly describing the program's flow (Fahland et al., 2009) such that the programmer is describing 'what' needs to happen. This approach lets the compiler avoid translating explicit statements into binary code and provides the freedom to apply a broader range of optimisations (Ullrich and de Moura, 2019). Functional programming treats computation as mathematical functions (Hutton, 2016), as demonstrated in Figure 2.9, this code describes a recursive function for summing all elements in a list that avoids the need for state variables. A possible imperative equivalent in the C programming language is proposed in Figure 2.10 and requires state variables 'i' and 'sum'.

```haskell
sumAll :: Num a => [a] -> a
sumAll (x:xs) = x + sumAll xs
sumAll [] = 0
```

Figure 2.9: Concise and recursive Haskell function for summing all elements in list.

```c
int sum_array(int a[], int num_elements)
{
   int i, sum=0;
   for (i=0; i < num_elements; i++)
   {
         sum = sum + a[i];
   }
   return(sum);
}
```

Figure 2.10: Possible equivalent imperative C code to the functional Haskell code in Figure 2.9

As described by Olsen (2018), the two core concepts of functional programming are:

- *Pure functions*, as taken from abstract mathematics, are functions that take an input that always maps to the same output and causes no side effects. In other words, there is no external state that influences or is influenced by this mapping.

- *Immutable Data Structures* enforces that all data is constant, and therefore when writing new data to memory, it must be copied to a new data structure. To avoid costly copying, data trees are used to allow only small sections of data structures to be copied between huge data sets.

A powerful emerging feature of immutable data and pure functions is the intrinsic suitability for parallel processing. Recall that the processing of data in parallel is optimal when data is independent of one another. Therefore, the guarantee of pure functions and immutable data to prevent side effects and data inter-dependence leads to an optimal parallel program (Lisper, 1996). Although these concepts can be limiting, data-parallel processes such as SIMD often operate comfortably in these requirements. The suitability of pure functional languages for parallel processing has been demonstrated in implementations like Walinsky and Banerjee (1990) and Jones et al. (1996) where the authors propose a functional programming language called “Concurrent Haskell”. The Haskell-level threads are easily mapped onto OS-level threads, usually one per available processing core.

One of the most compelling arguments for the use of functional programming is understandability (Fahland et al., 2009) and expressiveness (Felleisen, 1991). Although still widely debated, expressiveness is usually considered the amount of information represented in a programming language with the least number of characters [9]; the functional language’s deep rooting in lambda calculus allows familiar programmers to be expressive. This thesis will use a purely functional programming paradigm to define the HyperModels DSL for two reasons. Firstly, it provides a powerful agnostic representation of the design that can be understood and implemented in various forms through open interpretation. Secondly, the suitability of functional languages for parallel processing includes the opportunity for the design to be optimised further beyond the scope of this work. Next, the methods for defining a language will be described.

[9] At the time of writing, the literature investigating the superior language paradigm for understandability and expressiveness is inconclusive.

2.4.2 Defining a language

Formally describing a language is achieved by defining the language's grammar. In computer science, context-free grammar is frequently used for designing and building DSLs (Sakakibara, 1992). The core components of a grammar are a set of rules built on defining expressions. Expressions are a syntactic entity used to evaluate and then determine the value of a side-effect (Mitchell and Apt, 2003). An expression can be defined as one or more constants, variables, functions, operators and many other forms. Each rule is made up of two parts, a name and an expression of the name. Using the English language, it is possible to define a rule as:

expression may expand into *expression* + *expression*

Grammars are typically defined as mathematical objects, so instead of writing out "may expand into", a mathematical like notation is used. In this thesis, a variation on the Backus-Naur Form (BNF) (McCracken and Reilly, 2003) notation that includes regular-expression-like operators and omits <...> around expressions. To define new expressions, the ::= operator is used:

name ::= *expansion*

Therefore, the original statement can be more concisely expressed as:

expression ::= *expression* + *expression*

Every *name* begins describing a new pattern defined in the expansion. An expansion is one or more expressions containing terminal symbols and other defined *name* expressions. A terminal symbol is a literal, like "+" or "function" or a whole domain of literals, like *digits*. Placing expressions after one another indicates sequencing of expressions, meaning they are expected to appear in that order in the language. A vertical bar "|" indicates a choice, so when used between two sequences of expressions, the grammar is looking to match a pattern on one or the other side of "|". The classic expression grammar in BNF is (Backus et al., 1963):

expr ::= term "+" expr | term

Curly braces "{ }" indicate that the expression may be repeated zero or more times. So provided *arg* has been defined, a conventional comma separated list can be:

args ::= arg { "," arg }

To indicate precedence, BNF grammars can use brackets to explicitly define the order of an expansion:

expr ::= term ("+" | "-") expr

Along with these, the following BNF notation will be relevant in this thesis:

- postfix * means "repeated 0 or more times"
- postfix + means "repeated 1 or more times"
- expressions inside [ ] are optional.

The basic BNF form covered here can be used to define context-free grammars (Nelson, 2019), providing a powerful and expressive way for describing languages. The BNF grammar can be used to implement the compiler AST parser to begin forming the translation from source to equivalent target code.

2.4.3 Front-End Parser

Once the language has been formally defined, a parser that uses several stages to parse source code into the equivalent AST can be implemented. The first stage is referred to as lexical analysis (Aho et al., 2020), this is where the input character stream is used to create a set of meaningful tokens. A token takes the following form:

$$\langle token-name, attribute-name\rangle$$

This captures the meaning of each element in the source code and labels it with a unique ID. These tokens are then used in the subsequent syntax analysis stage for generating the AST. For example, suppose a language grammar was defined that supported variables, binary operators and assignment. The following statement can be written in the language:

$$position = initial + rate * 60$$

from which the following tokens would be generated:

$$\langle id, 1\rangle\langle =\rangle\langle id, 2\rangle\langle +\rangle\langle id, 3\rangle\langle *\rangle\langle 60\rangle$$

The syntax analysis stage uses this more meaningful set of tokens to generate the statement's abstract syntax tree representation. The tree is formed by finding operators like "+" as inner nodes with children nodes attached to it representing the arguments for that operator. Taking the example statement considered thus far, the abstract syntax tree in Figure 2.11 is generated. Note that the tokens identify variables with numbered ids because the parser can track these easily, unlike a human who prefers a human-readable variable. The AST captures the precedence of operators and variables as IDs in a navigable form and is prepared for code generation by the compiler back-end.

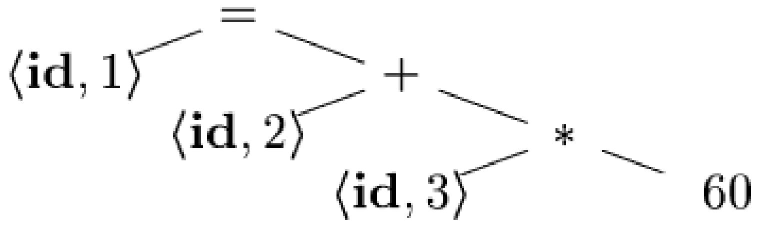


Figure 2.11: The AST of the example statement from Section 2.4.3

2.4.4 Back-End Compiler

The compiler's back-end includes all of the processes involved in taking the prepared AST from the parser and generating the target output code. For the context of this thesis, only the target code generation stage, as shown in Figure 2.8 will be described; more advanced compilers include additional stages such as an intermediate language representation and optimisation. The details of the code generation stage depend on the target language being generated. A compiler that generates machine code will typically use three-address instructions for mapping a low-level language such as assembly. In contrast, a source-to-source compiler (Schordan and Quinlan, 2003; Loveman, 1977) will use the AST to generate another high-level or source language. Examples of source-to-source compilers include the Closure Compiler (Bolin, 2010), Emscripten (Zakai, 2011) and features of the Mephisto transpiler (Demir, 2015).

Source-to-source compilers operate on the source and target languages, usually at the same level of abstraction, while a traditional compiler translates from high-level to low-level code. The primary aim of source-to-source compilers is to translate the programming source from a language used by one group of people to another. An additional benefit is the optimisation of code that might seem unintuitive or make the code less readable. A frequent technique used for source-to-source compilers is to build templates of boiler plate code that include necessary code from the target language, then using the AST to build up the target program by assembling pieces of the different templates and generating snippets of target code that can be injected into significant positions in the template code. Continuing to use the same example, the compiler can generate an equivalent c program function declaration from the AST by first forming the template code for a function:

```
$return-type$ $function-name$($variables$);
```

Here, there are three symbols inside the template called "$return-type$", "$function-name$" and "$variables$"; these are symbols that will be replaced with generated code snippets. Therefore, using the AST, the code generator can navigate the tree and collect the necessary information required to insert the correct return type, function name and variables involved in the statement. An example output of this generated C function declaration would be:

```
float getPosition(float idTwo, float idThree);
```

Using this technique, a whole c program can be formed by inserting all function declarations and their respective definitions into a template calling them from the main entry point to the C program. Provided the c program generated is correct, it can then be compiled for execution using one of the well established c compilers.

2.4.5 Domain Specific Language

A Domain-Specific Language (DSL) is a computer language that has been carefully constructed to support a particular field/domain. One of the core purposes of a DSL over a general-purpose language is to provide an accessible and expressive language to users familiar with a domain (Liu et al., 2005). For example, in Kramer et al. (2010), a DSL for developing mobile apps is presented with some evidence that the DSL enhances development for two particular case studies. Another study in (Kieburtz et al., 1996) involved users tasked with developing and maintaining software using general language template techniques and then a DSL language. Their analysis concluded that the DSL language proved better across all the chosen metrics: flexibility, productivity, reliability, and usability. Various other sources continue to suggest the improved productivity and usability of DSLs in particular scenarios (Kieburtz et al., 1996; Kos et al., 2010; Visser, 2007). The

Faust [10] functional programming language for sound synthesis and audio processing is a relevant example that supports the successful adoption of DSL by audio developers (Smith III, 2010). With a strong focus on the design of synthesizers, musical instruments and audio effects, Faust demonstrates the DSL form is effective within the audio domain (Smith, 2007).

DSLs have also been extensively used to enable high-performance computing (HPC) for non-programmers. Python frameworks such as OpenSBLI[11] have enabled developers to build finite-difference models using a set of Python functions that are mapped to multiple threading environments on the CPU and GPU (Jacobs et al., 2017). OpenSBLI might abstract developers from the low-level processing architectures, however, they still require the developer to be familiar with a general purpose programming language. DSLs like *Saiph* (Macià et al., 2018) have been designed to ease the task of simulating complex fluid dynamics systems without the use of general purpose programming languages. Saiph is written at the level of continuous PDEs and does the transformation from continuous to discrete versions in the compiler. This is a useful abstraction but comes at the cost of losing control of how the PDEs are represented in the discrete domain. Saiph has shown promising results for modelling real physical systems when comparing the performance on a particular High-Performance Computer called BSC-MareNostrum IV[12]. In their performance evaluation sections, Macià et al. (2022) demonstrated that for certain tasks Saiph was shown to outperform two other computational fluid dynamics software tools CREAMS (Ferrer et al., 2014) by 1.2x and OpenSBLI by 1.4x. The Saiph DSL supports the descriptions of the mesh geometry within the DSL, this approach suits programmers well but can be considered as less accessible for mathematicians. Therefore, visual tools that provide Direct Manipulation (Shneiderman, 1997) such as in Kusama and Saito (2021) are viewed favorably and promotes an ease of use alternative to non-programmers (Newton and Browne, 1992). Considering the existing literature, in this thesis a format using SVGs is proposed to support descriptions of the geometry using visual interfaces rather than describing them within the DSL like *Saiph*.

Another notable DSL for mesh based PDE simulations is Liszt[13]. Liszt is a Scala-based DSL for solving PDEs on meshes that can be mapped to scalar, cluster and GPU architectures for processing; all shown to have favourable performance when compared to the results reported in the literature (DeVito et al., 2011). Whilst both Saiph and Liszt

[10] `https://faust.grame.fr/`

[11] https://opensbli.github.io

[12] https://www.bsc.es/marenostrum/marenostrum

[13] https://graphics.stanford.edu/hackliszt/

include the definition of the mesh geometry in the DSL, Saiph simulation code is written as continuous PDEs while Liszt is written for the discretised mesh layer. Considering mathematicians are the target audience, most are expected to be competent and would want the opportunity to make manually make this mapping. Therefore in this thesis, the proposed DSL will operate at the level of discrete representations of the PDEs.

Considering the feasibility of DSLs and the evidence of their effectiveness in the HPC and audio domain already, this thesis suggests combining ideas from these fields to propose a new DSL for creating audio physical models that can be mapped to parallel processors, as seen in HPC.

Part I

Part 1 of the thesis explores and establishes the GPU's practicalities for processing digital audio for offline and real-time requirements. Quantitative performance profiling methods gather execution times across various tests to expose the limitations and highlight the strengths of the GPU. Chapter 3 describes and presents the results of a GPU audio benchmarking suite. This suite consists of micro tests starting with simple CPU to GPU data transfers up to macro tests involving complete audio synthesisers. These tests are profiled across several systems, and the results are used to evaluate the fundamental viability of the GPU for digital audio processing. With a foundational understanding of the GPU, two GPU accelerated designs are proposed and evaluated for an offline and a real-time application. Chapter 4 presents the design of an offline GPU accelerated evolutionary sound matching application, while Chapter 5 comprehensively covers the real-time support of a physical model synthesiser. Analysing the results highlights the contextual advantages between the CPU and GPU for suitable tasks. The evaluation of the GPU from Part 1 provides valuable insight for the development of HyperModels in Part 2.

Chapter 3

GPU for Digital Audio Evaluation

As covered in the literature, GPUs have been used for various general computation tasks, including digital audio applications. However, a comprehensive study benchmarking various core and fundamental digital audio processing on the GPU, particularly with real-time constraints, is missing. These include identifying a range of usable audio buffer sizes that meet real-time audio requirements, how different GPU interfacing software effects performance and how the GPU compares to the CPU for physical modelling audio synthesis. Therefore, to advance this thesis and justify the motivation for the contributions later, this chapter presents a benchmarking suite for evaluating the viability and practicality of using the GPU for processing digital audio by answering these fundamental questions. The evaluation is led by collecting performance profiles across distinct systems using a specifically designed benchmarking suite. The suite contains various tests for general digital audio and real-time requirements. One of the benchmarking suite's key features is the automated testing of an enumeration of sample buffer lengths dispatched to the GPU. This gives insight into the importance of the buffer length and how it affects the audio-sound latency and the performance to support real-time sonic interaction requirements.

The results of this chapter have been peer-reviewed and published in Paper [A] at the conference of New Instruments for Musical Expression (NIME) 2020 under the title "There and Back Again: The Practicality of GPU Accelerated Digital Audio" (Renney et al., 2020).

3.1 Benchmark Methodology

The benchmarking suite is used to manage the execution and profiling of a collection of various tests. The suite has been implemented in C++ to support the low-level GPU interfacing APIs OpenCL and CUDA. All tests are functionally equivalent but implemented separately to target OpenCL and CUDA. The tests are categorised into two types, general tests and real-time tests. The general tests include various types ranging from isolated micro-benchmarks to more extensive full-featured programs. The real-time tests are designed to measure the performance at a set sample rate, demonstrating direct performance within real-time constraints. The benchmarking tests take inspiration from the heterogeneous benchmarking suite Hetero-mark, proposed by Sun et al. (2016). In Hetero-mark, benchmark tests are defined as being unidirectional or bidirectional. Unidirectional tests involve a single direction of data transfers between the CPU and GPU, while Bidirectional tests involve data transfers in both directions. The tests are all executed and profiled for an enumeration of different buffer lengths. Buffer length is an essential controllable parameter that significantly affects the real-time audio-sound latency and sonic interaction requirements. Therefore, profiling the range of buffer lengths will expose the ranges of acceptable buffer lengths for real-time requirements. The buffer sizes tested are an enumeration starting from 1 and increasing by powers of 2 up to 32768. This series of buffer sizes were chosen as powers of 2 are flexible for further processing in DSP, such as when FFT is calculated (Bailey, 1988). The benchmarking suite follows a conservative approach; for example, explicit synchronisation between the CPU and GPU is confirmed after every GPU interfacing event. This approach gives more confidence in the results as they do not rely on any case-specific advantages. This means that the explicit synchronisation can be turned off in practice, and the implicit synchronisation managed by the GPU driver will only improve performance further.

All the systems used to collect benchmarking results in this chapter are outlined in Table 3.3. For reference, the relevant specification for the CPUs including cores and clock speeds can be found in Table 3.1 and for the GPUs in Table 3.2. These systems range from a modest mid-range laptop that contains GPUs with hundreds of cores to professional high-end desktops with powerful NVIDIA and AMD GPUs. The mid-range laptop has an optional integrated GPU, as described in Section 2.3.5, these share their memory space with the CPU over a ring bus memory interface, providing fast memory transfers between CPU and GPU, unlike the other discrete GPUs that have the added overhead of transferring data across a PCIe interface.

Device	**Intel Core i7-8550U**	**Intel Core i7-9800X**
Cores	4	8
Threads	8	16
Clock Speed (MHz)	1800	3800
Peak FP32 performance (GFLOPS)	230	972
Cache L1 (KB)	64	64
Cache L2 (KB)	256	1000

Table 3.1: Specification of all CPUs used in systems in the thesis.

When collecting benchmarking results from the hardware systems, the software has been consistently selected to directly compare hardware results. In this chapter, all systems run on the same Windows 10 Version 1909[1] operating system and use the OpenCL standard 1.2[2], whilst all CUDA programs use SDK version 10.2[3]. Each hardware vendor provides it's own implementation of OpenCL 1.2 and the versions used in this chapter are:

- Intel = Intel CPU SDK for OpenCL Applications 2019
- AMD = AMD APP SDK v2.9.1
- NVIDIA = NVIDIA's OpenCL implementation is from the CUDA SDK version 10.2

It might be noted that the OpenCL version used is released 4 years before the CUDA version used despite there being a later version of OpenCL. This is because, at the time of writing, the NVIDIA GPUs do not support versions of OpenCL higher than 1.2. This puts OpenCL at a disadvantage when comparing with CUDA and for a fair comparison of these technologies, only equally supporting hardware would be used. However, in the context of this thesis, the computation of hardware from multiple vendors is considered and compared and therefore the most widely supported version of OpenCL must be used.

3.1.1 Performance Profiling

CPU timers are a reliable and consistent method for measuring the execution time between isolated program sections. Using the C++ standard library, timestamps can be

[1] https://learn.microsoft.com/en-us/windows/release-health/status-windows-10-1909

[2] https://registry.khronos.org/OpenCL/specs/opencl-1.2.pdf

[3] https://developer.nvidia.com/cuda-10.2-download-archive

Device	Intel UHD Graphics 620	AMD Radeon 530	Radeon Pro WX 7100	Nvidia GeForce RTX 2080 Ti	Nvidia Titan RTX
SM	24	6	36	68	72
Cores	192	384	2304	4352	4608
Clock Speed (MHz)	300	730	1188	1350	1350
Peak FP32 performance (GFLOPS)	384	786	5728	13450	16310
Private Memory (KB)	N/A	16	16	64	64
Local Memory (KB)	N/A	128	2000	5500	6000
Global Memory	N/A	4GB	8GB	11GB	24GB
Memory Bus Bandwidth (GB/s)	N/A	14.40	224.0	616.0	672.0
Memory Bus Interface	Ring Bus	PCIe 3.0 x8	PCIe 3.0 x16	PCIe 3.0 x16	PCIe 3.0 x16

Table 3.2: Specification of all GPUs used in systems in the thesis.

Specification	Mid-range Laptop	High-end AMD	High-end NVIDIA GeForce	High-End NVIDIA Titan
CPU	Intel Core i7-8550U	Intel Core it-9800X	Intel Core it-9800X	Intel Core it-9800X
Integrated GPU	Intel UHD Graphics 620	None	None	None
Discrete GPU	AMD Radeon 530	Radeon Pro WX 7100	GeForce RTX 2080 Ti	Titan RTX
RAM	8GB	32GB	32GB	32GB

Table 3.3: System hardware specification used for benchmarking

taken on either side of a program section and the difference calculated between the timestamps is the execution time of that section. In this thesis, CPU timers have been used over alternative GPU profiling tools for several reasons. Firstly, CPU timers provide a realistic representation of the time between initiation and completion of a process on the GPU, including any of the communication overhead between the CPU and the GPU. Secondly, using GPU profiling tools leads to complications with inconsistent profiling tools, as each GPU vendor has different tools that measure execution times differently. Instead, CPU timestamps will be used to provide a standard and consistent method for fair comparison when analysing the results.

The benchmarking suite measures each test's execution time by timestamping points before and after the test completes. The two timestamps are then used to calculate the elapsed time to execute the test. The elapsed time is recorded in an array for each iteration of the test executed and collected together. Once all test iterations are completed, all the timestamp measurements can be used to calculate the average execution time, max execution time and execution time variance. This collection of profiling metrics will provide the necessary information for evaluating the real-time capability of the GPU.

A theoretical estimate for the number of FLOPs a profiled program requires will be calculated as:

$$FLOPS = N_x * N_y * r_s * N_o \tag{3.1}$$

where N_x and N_y are the resolutions of the physical model in the x and y directions and r_s continues to be the sample rate. N_o is the typical number of operations executed per grid point and in this thesis a naive approach will be used and every floating point operation will just be considered as 1 FLOP. In practice, this theoretical FLOP count will be inaccurate as each hardware device will have a different FLOP count for their operations. However, this is the most accessible way to generalise the flop count across various hardware and gives an indication of how intense a program is expected to be. This gives further insights when compared with the actual observed results.

3.1.2 Real-Time Requirements

The real-time requirements discussed in Section 2.1.5 will be used for evaluating the real-time performance in the benchmarking suite. The results measure and present the *Average Execution Time*, *Max Execution Time* and *Execution Variability*. For evaluating the core real-time *audio-sound latency*, the *Average Execution Time* will be compared with the maximum acceptable audio buffer period from Table 2.1; if it exceeds the corresponding maximum audio buffer period, it is coloured red[4], otherwise it is coloured green. For evaluating the real-time sonic interaction, the *Max Execution Time* will be compared to the *Action-Sound Latency* limits and the *Execution Variability* will be compared to the *Action-Sound Latency Variability* from Table 3.4. Therefore, if either of these measurements are less than the *Recommended* range, they are coloured green. If they are outside of the *Recommended* but inside the *Acceptable* range, they are coloured orange. However, if they are outside of these ranges, they are considered to fail to meet real-time sonic interaction and coloured red. If the results demonstrate that a specific configuration can support these real-time requirements in the *Recommended* range, this shows good evidence that they can be supported in real-time. If they are only within the *Acceptable* range, this is evidence it might work in real-time but is particularly sensitive to any additional overhead and additional sources of latency such as those considered in detail by McPherson (2017).

3.1.3 General Tests

In this Chapter, the general tests are executed 10,000 times for each enumeration of the buffer lengths where performance measurements are recorded. These include the average and maximum execution times for processing buffers along with the execution time variation observed. The general tests in the benchmarking suite are:

[4] The colour palette used follows a colourblind appropriate palette proposed by Wong (2011).

Requirement	Recommended	Acceptable	Fail
Sample Rate	96000	44100	<44100
Action-Sound Latency	10ms	20ms	>20ms
Action-Sound Latency Variability	±1ms	±3ms	>±3ms

Table 3.4: Real-time audio requirement ranges for evaluating real-time performance as derived from Table 3.6

- *null kernel* - A minimal test to measure the threshold overhead to execute an empty program on the GPU with no data transfers.
- *cpu to gpu* - A unidirectional test measuring the data transfer from CPU to GPU.
- *gpu to cpu* - A unidirectional test measuring the data transfer from GPU to CPU.
- *cpu to gpu to cpu* - A bidirectional test measuring the round-trip transfer time between CPU and GPU.
- *simple buffer processing* - Simple buffer processing is a bidirectional test which applies a constant attenuation rate to the input samples.
- *complex buffer processing* - Applies a triangular smoothing operation (O'Haver, 1997, p. 34) to the input signal and returns smoothed buffers to CPU. Involves bidirectional transfers and multiple memory accesses in the kernel.
- *simple buffer synthesis* - Generation of a sinusoidal signal at a given frequency, generating sine values that fill the buffer length in parallel. This operation involves only unidirectional memory transfers from the GPU to the CPU, returning synthesised sample buffers.
- *complex buffer synthesis* - The complex buffer synthesis is an application that has a challenging amount of computation, bidirectional CPU-GPU transfers and involves memory management. An application meeting these high requirements is a finite-difference time-domain physical model synthesizer, such as Willemsen et al. (2020). The full design and implementation is covered in Chapter 6.
- *Interrupted buffer synthesis* - This test extends the "simple buffer processing" program by adding a 50% chance that the current buffer must discarded. This interrupting mechanic demonstrates how the GPU manages frequent unpredictable events that invalidate the results being processed.

The execution of the above general tests are inserted into the following template for extensive profiling:

```
void generalTest() {
    prepareTest(hostVariables, deviceVariables);

    if(isWarmup) {
        executeTest();
    }
    for(int i = 0; i != numRepeats; ++i) {
        startTimer();
        executeTest();
        endTimer();
    }
    elapsedTimer();
    checkTestResults(testResults);
    cleanup(hostVariables, deviceVariables);
}
```

Here, each test makes all preparations by initializing CPU and GPU memory used in the test inside *prepareTest()*. Further, kernel code is prepared and loaded onto the GPU if necessary. Both CUDA and OpenCL take considerably longer executing programs and transferring data the first time. This is expected as GPU preparations and optimizations are established to increase the performance of subsequent execution. For this reason, a warmup variable has been added to execute the test once prior to profiling. The timestamping can be seen either side of *executeTest* and concludes with calculating performance metrics when *elapsedTimer* is used. After the test has been completed, the integrity of the results is checked. Finally, the CPU and GPU memory allocations are deallocated in the *cleanup* function, ready for the next test.

3.1.4 Real-Time Tests

The real-time tests are designed to measure the performance by processing a number of samples equal to a specific sample rate. The performance measurements will then provide insight into acceptable performance as the enumeration of buffer sizes are tested. Four real-time tests have been designed to test simple and complex unidirectional and bidirectional scenarios:

- *unidirectional baseline* - Transfers data from the GPU to the CPU without any processing at the specific sample rate.
- *unidirectional processing* - Generates samples using the *simple buffer synthesis* test and then transfers them from the GPU to the CPU.

- *bidirectional baseline* - Transfers data from the CPU to the GPU and back without processing at a specific sample rate.
- *bidirectional processing* - Data transfers to and from the GPU for the *complex buffer synthesis* test at the set sample rate.

The real-time tests process a full second's worth of samples at a given sample rate. Therefore, smaller buffer lengths will require more iterations and data transfers to the GPU, improving the action-sound latency and the action-sound latency variance at the expense of overall performance to maintain the audio-sound latency requirement. This is achieved using the following real-time template:

```
void realtimeTest() {
    if (isWarmup) {
        executeTest();
    }
    while (numSamplesComputed < sampleRate) {
        startTimer();
        executeTest();
        endTimer();

        numSamplesComputed += bufferLength;
        checkTestResults(testResults);
    }
    elapsedTimer();
    cleanup(hostVariables, deviceVariables);
}
```

Here, instead of executing for a set number of times like in the general tests, the test is executed for a second's worth of samples at the sample rate. Therefore, the number of times the test must execute is dependant on the *bufferLength* and *sampleRate* variables. For example, if the test sets a *sampleRate* of 44.1KHz and a *bufferLength* of 128, the test is executed $\lceil 44100/128 \rceil = 345$ times. In these tests, the profile timer started using *startTimer()* and ended with *endTimer()* recordes the execution time of each buffer to calculate the *Average Execution Time*. These results can then be checked against the core audio-sound latency requirements from Table 2.1 and if the *Average Execution Time* is below the corresponding buffer length's *Maximum Audio Buffer Period*, then it meets the audio-sound latency requirement. Taking the same *bufferLength* of 128 as above, if a *Average Execution Time* of 1.521ms was recorded, this would satisfy the audio-sound latency requirement as the corresponding *Maximum Audio Buffer Period* for an audio buffer length of 128 is 2.902ms.

The timer profiles two further measurements, *Max Execution Time* and *Execution Variability*. These are compared to the ranges of real-time sonic interaction requirements from Table 3.4. For example, in a real-time test regardless of *bufferLength*, if a *Max Execution Time* of 15.450ms was recorded and 0.789ms for the *Execution Variability*, then the real-time sonic capability would be evaluated as acceptable. This is because although the *Execution Variability* was within the recommended *action-sound latency variability* range, the *Max Execution Time* exceeded the recommended *action-sound latency* but satisfied the acceptable range.

3.2 Results

The results presented in this section build up from the most fundamental operations between a CPU and GPU to complete applications. The results are then reviewed to demonstrate how the GPU takes advantage of its massively parallel architecture.

3.2.1 Minimum GPU Overhead

The absolute bare minimum GPU overhead can be seen in the measurements recorded from the *null kernel* test. This test transfers no data and only executes an empty program on the GPU. If the absolute bare minimum overhead exceeds or is even close to any of the real-time requirements, this would suggest that the GPU is fundamentally unsuitable for real-time applications. The bare minimum GPU overhead recorded in OpenCL for the *null kernel* test are: $0.002051ms$ on the *Radeon 530*, $0.000455ms$ on the *Intel UHD Graphics 620*, $0.009392ms$ on the *GeForce RTX 2080 Ti*, $0.011468ms$ on *Titan RTX*. These are significantly short execution times that fit the real-time requirements and leave plenty of room for important data transfers and processing.

The *bidirectional baseline* real-time test measures the time to transfer buffers of data to the GPU, execute a null kernel and then return the buffer of samples to the CPU. This demonstrates the performance of purely back and forth data transfers before any processing is included. Table 3.5, presents the results for the *bidirectional baseline* test using OpenCL and CUDA at a sample rate of 44.1KHz on the *GeForce RTX 2080 Ti*. The results show the *Average Execution Time* that is compared with the core maximum acceptable buffer period from Table 2.1 and coloured Green if within this limit and red if outside this range. It can be seen that for buffer lengths 1, 2 and 4, the maximum audio buffer period is exceeded and these buffer lengths will not be supported in real-time even before any processing is applied to the samples. The results for the other discrete GPUs have similar results and suggest that smaller buffer lengths below 8 will not meet the sample rate for discrete GPUs. However, the integrated GPUs seem to be an exception and the *Intel UHD Graphics 620* can operate of buffer lengths as short as 1 because it does not rely on transferring data over a PCI bus and instead uses the low overhead unified memory between the CPU and GPU. This makes the integrated GPU a good choice when designing applications requiring smaller buffer lengths.

GPUs are designed for transferring large amounts of data from the CPU to the GPU as they are typically handle large texture graphical assets that can involve gigabytes of data. This is reflected in the specification seen in Table 3.2, the *Nvidia GeForce RTX 2080 Ti* used in the *High-end NVIDIA GeForce* system uses PCIe 3.0 x16 to transfer data between the CPU and GPU memory and has a peak theoretical transfer rate of 224GB/s. Considering audio processing only requires a single channel of audio at 44100Hz, if single-precision floating point values of 4 bytes are used per sample and considering that these samples must go both directions, this only requires a transfer rate of $44100*4*2 = 0.0003528GB/s$. Comparing this with the peak theoretical transfer limit of 224GB/s, theoretically these audio processes only utilise 0.000001575% of the possible transfer capabilities; a negligible fraction of the potential transfer rate available. However, despite being smaller buffers, the minimum latency involved with each independent transfer exceeds the acceptable maximum buffer period and prevents this peak performance being reached. The evidence for this is shown in Table 3.5, where the time to transfer small buffer lengths between 1 to 4 exceeds their respective maximum audio buffer periods and can not be supported in real-time. By taking the smallest buffer length of 1 and observing its *average execution time* is 0.139ms, it could be considered as the minimum latency involved in any data transfers across the PCI 3.0 x16 bus. This leads to the hypothesis that the accumulation of the minimum latency for data transfers adds up and prevents the real-time sample rate being sustained. This observation emphasises the limitation increasing the number of individual transfers per second has on the data throughput. Even before any processing begins, this transfer latency forces the program to be bandwidth bound when smaller buffers are used in real-time audio processing.

The action-sound latency and variation requirements seem to comfortably fit into the recommended limits for almost all buffer lengths (with the exception of buffer length 1 on CUDA having satisfactory variation). This suggests the sonic interaction requirements can be supported when transferring data between the CPU and the GPU. However, now the results from tests that involve additional sample processing will be considered.

	GeForce2080_cl			**GeForce2080_cuda**		
Buffer Length	Average Execution Time	Max Execution Time	Execution Variability	Average Execution Time	Max Execution Time	Execution Variability
1	0.139	0.902	0.763	0.128	1.157	1.029
2	0.138	0.967	0.829	0.128	0.869	0.741
4	0.137	0.549	0.412	0.128	0.792	0.664
8	0.136	0.307	0.171	0.128	0.755	0.627
16	0.137	0.302	0.165	0.137	0.872	0.735
32	0.137	0.373	0.236	0.133	0.749	0.616
64	0.139	0.318	0.179	0.138	0.796	0.658
128	0.141	0.341	0.200	0.149	0.751	0.602
256	0.142	0.411	0.269	0.156	0.398	0.242
512	0.145	0.334	0.189	0.336	0.782	0.446
1024	0.144	0.177	0.033	0.375	0.823	0.448
2048	0.141	0.155	0.014	0.357	0.636	0.279
4096	0.146	0.166	0.020	0.346	0.763	0.417
8192	0.151	0.159	0.008	0.229	0.251	0.022
16384	0.166	0.181	0.015	0.235	0.264	0.029
32768	0.184	0.186	0.002	0.254	0.256	0.002

Table 3.5: Comparison of OpenCL and CUDA versions of the baseline bidirectional real-time test at 44.1KHz using the *High-end NVIDIA GeForce* system in milliseconds.

3.2.2 Unidirectional Transfers Compared

The two unidirectional tests *cpu to gpu* and *gpu to cpu* isolate the data transfers between the CPU and GPU in one direction. Being reversed tests, one might presume that these would perform equally. However, reports from Sandgren (2013) and Ketchum et al. (2012) investigated data transfers between the CPU and GPU using OpenGL and found that transferring data to the GPU is faster than from it. However, OpenGL is primarily designed as a graphics rendering API, which does not benefit from optimising GPU to CPU transfers. On the other hand, OpenCL and CUDA are designed for general compute with bidirectional data transfers; therefore, the results are worth investigating. Figure 3.1 presents results for the two tests using the OpenCL and CUDA. The *cpu to gpu* tests are plotted as solid lines and *gpu to cpu* by dashed lines - OpenCL versions of the tests at preceded by cl_ in the legend and coloured as blue. CUDA versions of the tests are preceded by cu_ and are green. It can be seen that in both APIs, the data transfers from CPU to GPU execute quicker than GPU to CPU. Looking specifically at buffer length 128, the difference between transfer directions is around 0.007 and 0.008ms for CUDA and OpenCL. To put this into perspective, this increases the transfer times by only 0.0015%. Therefore, bidirectional synthesis applications should not experience issues with the additional performance implications of data transfers back to the CPU. These results support the continued design and development of the GPU synthesis tools presented in this work.

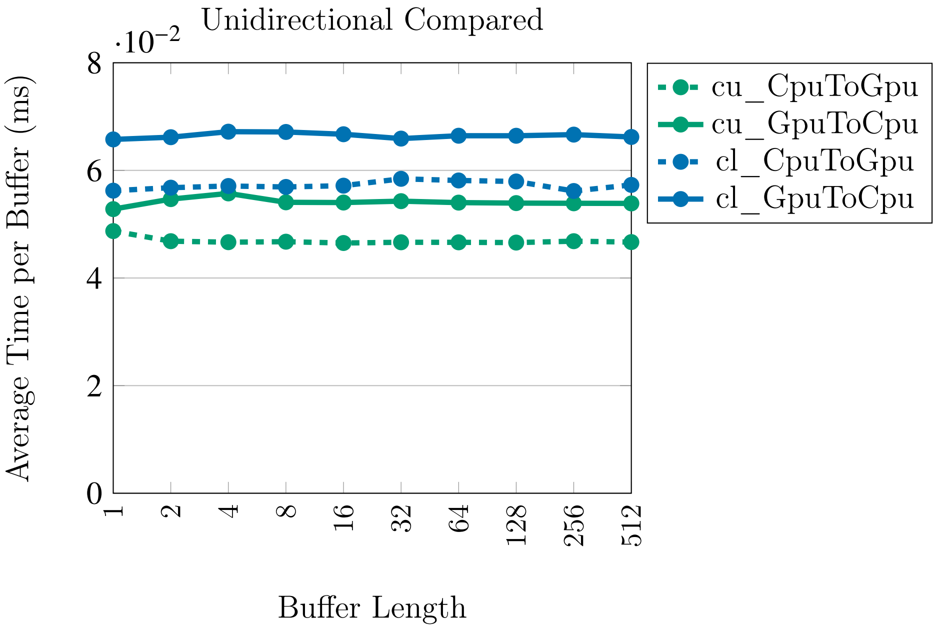


Figure 3.1: Execution time for *cpu to gpu* and *gpu to cpu* tests when scaling the buffer length. OpenCL and CUDA versions indicated by preceding cl_ and cu_ respectively.
System = High-end NVIDIA GeForce

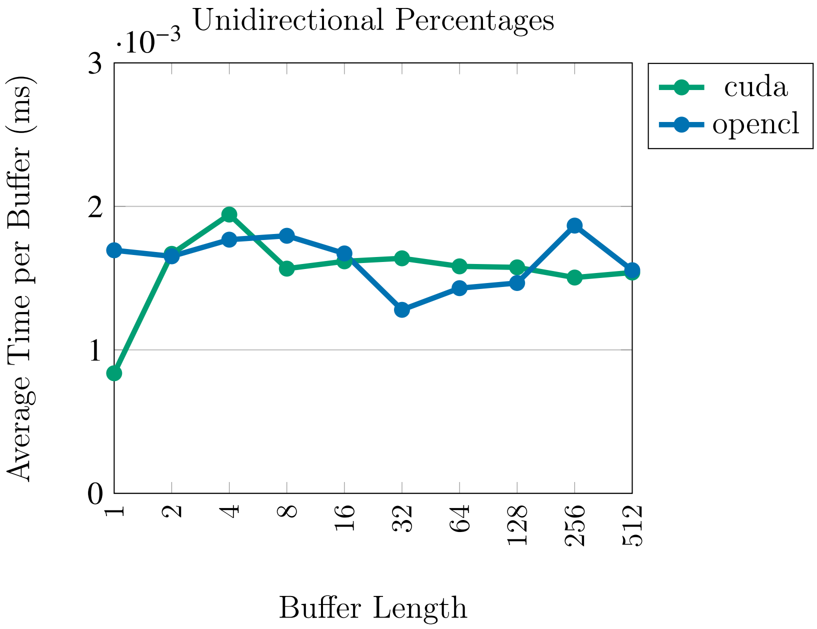


Figure 3.2: Execution time for *cpu to gpu* and *gpu to cpu* tests when scaling the buffer length. OpenCL and CUDA versions indicated by preceding cl_ and cu_ respectively.
System = High-end NVIDIA GeForce

3.2.3 Warmup

The "warmup" refers to a test run executed once prior to timestamping and measuring performance. The GPU interfacing APIs prepare optimisations the first time the instructions are executed to improve performance in subsequent execution of the same instructions. The benchmarking suite exposes a variable controlling the inclusion of the warmup test run or to exclude it. A substantial performance difference can be observed across all the tests in OpenCL and CUDA when enabling and disabling the warmup phase. Figure 3.3 presents the results for the *simple buffer synthesis* test including and excluding the warmup run for OpenCL and CUDA where the buffers are executed 100 times and averaged over this period. The results for OpenCL and CUDA including the warmup phase in the performance profiling measurements are indicated by dashed lines of blue and green, respectively; the corresponding results excluding the warmup phase in measurements are shown as a solid lines. Both OpenCL and CUDA appear to perform significantly faster on average when excluding the warmup phase (excluding one measurement for CUDA with buffer length 64). OpenCL has a more significant difference across the buffer lengths around 0.6 - 1.1ms, while CUDA is between 0.04 - 0.29ms. This suggests that OpenCL may take longer on its first execution, while CUDA initially starts faster and optimises to a similar level. Once optimised after the warmup run, OpenCL and CUDA operate at a similar performance for this test. These results highlight that the GPU performance is affected by regularly changing the program executing on it. Therefore, it is recommended to set up the program to run on it once and reuse it. This is in line with the GPU recommended use and therefore supports an existing guideline (Cheng et al., 2014).

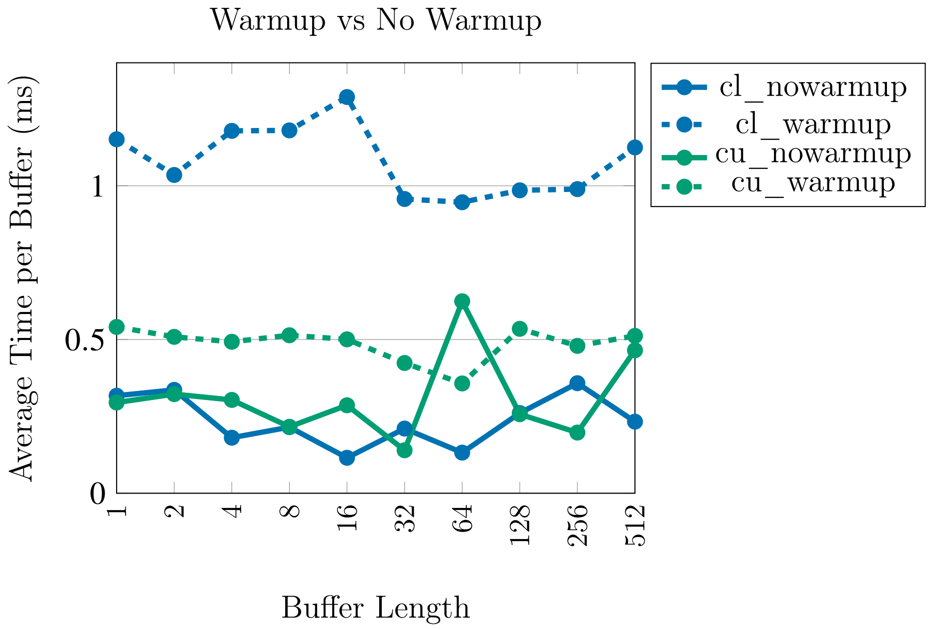


Figure 3.3: Execution time for *simple buffer synthesis* test using a warmup run against no warmup run. **System** = High-end NVIDIA GeForce

3.2.4 Intensive GPU Overhead

This section analyses the results of the most computationally intensive test in the benchmarking suite. The *bidirectional processing* test involves round-trip data transfers for a computationally intensive physical model synthesiser with 2D model dimensions of 64x64. This test involves high computation relative to the previous *bidirectional baseline* test yet still only requires the same data transfer of $0.0003528GB/s$. This is because although the physical model synthesiser processes considerably more data to generate audio samples, the majority of the data remains on the GPU memory and is optimised to only transfer the necessary audio sample data between the CPU and GPU. The theoretical processing throughput for this synthesiser when taking some liberties to ignore the boundary condition, works out to be $44100 * 64 * 64 * 12 = 2.167$GFLOPS. Considering the peak floating point performance of the *Nvidia GeForce RTX 2080 Ti* is 13450GFLOPS and 5728GFLOPS for the *Radeon Pro WX 7100*, the GPUs should be able to process the physical model synthesiser as they have considerably more processing throughput than required by the synthesiser. However, as with the data throughput seen in Section 3.2.1, because of the limitations imposed by buffering and real-time audio synthesis, the buffer length determines how much of this theoretical peak throughput can be used. Furthermore, the utilisation of the peak processing throughput is reduced further by the physical model synthesiser as it includes synchronisation stages in between the calculation of every sample.

Table 3.6 presents the average and maximum execution times of buffers along with the variation observed between the recorded buffers for the *Nvidia GeForce RTX 2080 Ti* and the *Radeon Pro WX 7100* using OpenCL. The same colour coding used in Table 3.5 is used again here. Here it can be seen that smaller buffer sizes that have shorter maximum audio buffer periods are not supported in real-time with the *Radeon Pro WX 7100* supporting buffer lengths 8 and up while the *Nvidia GeForce RTX 2080 Ti* is less capable and can only support buffer lengths 32 and above[5]. Therefore, considering only the core real-time requirement, buffer lengths greater than 32 can be supported in real-time for this physical model synthesiser for both GPUs considered here. However, considering the real-time sonic interaction requirements some further limitations are applied to the buffer length. Looking at the *Max Execution Time* for both GPUs, buffer lengths of 4096 and above fail to meet the acceptable action-sound latency and would therefore not be suitable for any applications that require real-time interactions. However, buffer lengths as high as 2048 can be used for both GPUs and meet the acceptable action-sound latency requirement of $>20ms$ but must go down to 512 and below to meet the recommended requirement of $>10ms$. The action-sound latency variation requirement measured in the *Execution Variability* column provides an unclear separation between buffer lengths that are supported as some buffer lengths above 512 work and others are not. However, it still seems clear that all buffer lengths below 512 meet the recommended action-sound latency variation requirement while some above 512 do not.

In conclusion, the results observed here suggest for core real-time using the audio-sound requirement, buffer lengths 32 and higher are supported. For real-time sonic interaction, the recommended requirements can be satisfied using buffer lengths 512 and lower. For this particular synthesiser, this gives the GPU a reliable buffer length range of 32-512 to use. These observed results fit well into the audio processing paradigm as typically digital audio workstations recommend using buffer lengths 32, 64, 128, 256, 512, and 1024 (Sweetcare, 2022).

3.2.5 Integrated vs Discrete

As covered in Section 2.3.5, there are two types of GPU, integrated and discrete GPUs. These GPU types have different memory arrangments and are expected to significantly

[5] Although the *Nvidia GeForce RTX 2080 Ti* is a more powerful GPU, it is possible that the vendor does not support OpenCL as well as the *Radeon Pro WX 7100*

	GeForce2080			Radeon7100		
Buffer Length	Average Execution Time	Max Execution Time	Execution Variability	Average Execution Time	Max Execution Time	Execution Variability
1	0.154	1.127	0.973	0.099	0.469	0.370
2	0.171	0.459	0.2887	0.106	0.476	0.370
4	0.198	0.564	0.3665	0.132	0.455	0.323
8	0.256	0.577	0.321	0.175	0.546	0.371
16	0.380	0.664	0.284	0.244	0.617	0.373
32	0.478	0.940	0.462	0.401	0.950	0.549
64	0.693	1.103	0.410	0.737	1.198	0.461
128	1.191	1.667	0.476	1.350	1.667	0.317
256	2.253	2.787	0.534	2.269	2.591	0.322
512	4.295	6.064	1.769	4.261	4.670	0.409
1024	8.572	12.736	4.164	7.704	8.022	0.318
2048	16.516	16.854	0.338	14.083	14.946	0.863
4096	32.900	33.704	0.804	27.492	28.727	1.235
8192	65.643	66.226	0.583	53.704	54.844	1.14
16384	138.268	142.148	3.88	106.810	107.738	0.928
32768	285.371	286.467	1.096	213.848	215.347	1.499

Table 3.6: Comparison of *High-end NVIDIA GeForce* and the *High-end AMD* systems for an intensive physical model synthesiser bidirectional real-time test at 44.1KHz measured in milliseconds.

affect performance. Results have been collected from multiple discrete GPUs and an integrated GPU. In Figure 3.4, results for the *bidirectional processing* test using OpenCL at 44.1KHz is presented. The discrete GPUs are represented as solid coloured lines; the integrated GPU is a dashed blue line. The total time to compute the 44.1KHz of samples has been presented on a logarithmic scale to emphasize the subtle differences at the lower measurements. For small buffer lengths, the communication overhead experienced by most of the discrete GPUs heavily outweighs the benefits of using it. The integrated GPU avoids communication overhead and performs better. However, as larger buffer lengths are used, and the transfer overhead is reduced, the discrete GPUs performance surpasses the integrated GPU from buffer length 8 onwards. This is the expected behaviour as although the discrete GPUs take longer to communicate and transfer smaller buffers, they have high processor clock speeds. Therefore, the communication cost is increasingly negated when larger buffers are used. Note that all the discrete GPUs settle beneath the 1000ms threshold by buffer length 16, whilst the integrated GPU is not powerful enough to compute any buffer lengths in real-time for this intensive test.

The results here indicate that the faster data transfers of the integrated GPU are more suitable for applications that depend on smaller buffer lengths. On the other hand, the discrete GPU is preferred for tasks requiring substantial processing, and the communication overhead can be reduced using larger buffer lengths.

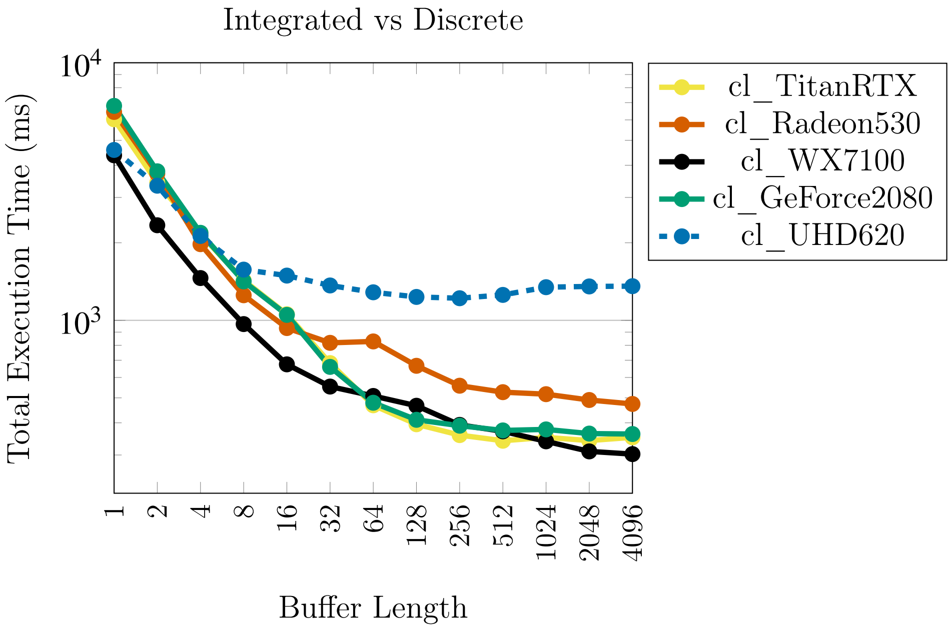


Figure 3.4: Total execution time for *cpu to gpu to cpu* test when scaling the buffer length.
System = Various

3.2.6 Standard vs Pinned

Figure 3.5 presents the *cpu to gpu to cpu* general test for the OpenCL implementation. This test measures the round-trip data transfer from the CPU to the GPU and back for various buffer lengths; no processing is executed on the GPU. The graph displays the standard buffer allocations as solid coloured lines for the different GPUs and dashed lines for the pinned memory buffer versions. There is a consistent improvement in using the pinned memory for most GPUs, excluding the integrated GPU. The smallest improvement seen is around ±0.04ms for the GeForce2080, and the largest for the Radeon530 is ±0.12ms. The choice of standard or pinned memory on the integrated Intel GPU appears to have little to no impact on the performance. This is because the integrated GPU shares unified memory space with the CPU, as described in 2.3.5. Therefore, OpenCL and CUDA likely default to using the unified memory for both types of memory specified in the program. However, for the discrete GPUs, the consistent improvement of pinned memory is beneficial. Therefore, using pinned memory is a key optimisation that should be incorporated into the design of GPU audio to improve overall performance.

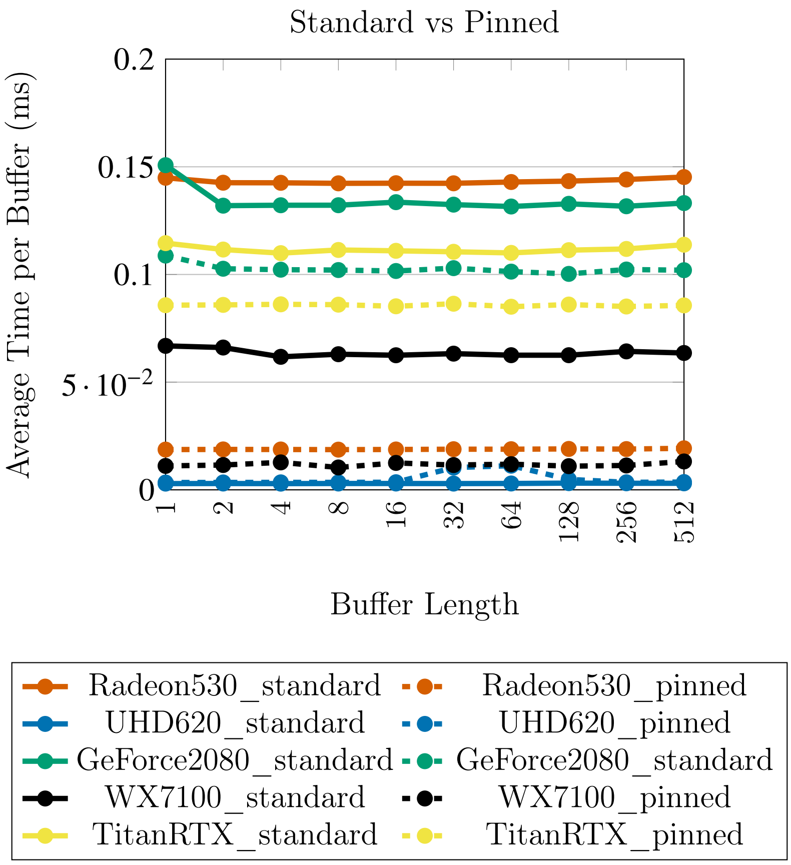


Figure 3.5: Execution time for *bidirectional processing* test when scaling the buffer length for various systems.

3.2.7 GPU Evaluation

The results collected and analysed in this chapter have provided valuable insight into the strengths and limitations of the GPU, particularly surrounding the real-time requirements. The results suggest that the GPU can operate in real-time provided certain conditions are met. The baseline results showed that fundamentally, before any processing begins, the GPU operates within all the real-time requirements. The implications of the GPU overhead involved with data communications back and forth from the GPU were analysed and shown to operate within the real-time limits for buffer lengths 8 and above. This is because the minimum data transfer latency even for a single byte is roughly 0.139ms which exceeds the maximum buffer period for buffer lengths below 8 transfers. This explains why despite needing to use a negligible fraction of the PCI 3.0 x16 peak transfer rate, sectioning the 44100 bytes per second to numerous independent transfers accumulates and makes the program bandwith bound. On the other end of the spectrum, intensive processing tests were executed and highlighted the acceptable buffer lengths for meeting the core real-time audio-sound latency requirements by comparing the average execution time to the maximum audio buffer period. For the most intensive benchmarking task, it was found that for GPUs tested, buffer lengths 32 and above are supported in real-time. Applying further limitations for assessing real-time sonic interaction using the action-sound latency and action-sound latency variation requirements, the results provide evidence that buffer lengths 512 and lower mostly support the recommended requirements for reliably supporting real-time sonic interaction. Combining all the real-time requirements, the buffer range supported by the GPU 32-512 appears to be well supported by the GPU and this fits into the typical buffer length range for digital audio workstations that ranges between 32-1024 (Ramm, 2021). Therefore, only buffer lengths in the range of 32-512 will be used for the real-time applications presented in the rest of this thesis.

The warmup results demonstrate that the first execution of a GPU program involves considerable overhead compared to subsequent executions of the same unaltered program. Therefore, running the same program repeatedly with updated parameters avoids any overhead of changing the GPU program. As a result, physical modelling synthesisers can be designed to repeatedly execute the same program and maintain peak performance.

The relationship between CPU and GPU memory depends on the type of GPU. For most tests, integrated GPUs were shown to perform better than the discrete GPUs for smaller buffer sizes, usually 8 and under. The discrete GPUs were more suited for processing-intensive tests where the negatives of the communication overhead are surpassed by the benefits of a more powerful GPU, from roughly buffer length 16 and above. Pinned memory was slightly beneficial when used appropriately for memory regularly accessed by the CPU. The direction of memory transfers was consistently faster from CPU to GPU than GPU to CPU. However, the difference was 0.0015% and, therefore, unlikely to affect most applications.

Chapter 4

GPU Offline Evaluation - Evolutionary Sound Matching

Processing digital audio to meet the real-time requirements (outlined in Section 2.1.5) is challenging and imposes strict limitations. However, numerous useful digital audio applications do not require real-time requirements, including sample pack generation (Histibe, 2016) and synthesiser sound matching (Mitchell and Creasey, 2007). Further, as explored in output from the NESS project, GPUs have been used to accelerate offline processing of room acoustics (Hamilton and Webb, 2013) and sophisticated non-linear physical models (Bilbao and Webb, 2012).

This chapter continues the investigation of applying GPU acceleration to offline audio applications by proposing a synthesiser parameter matcher. The summary of contributions of this chapter has been published in Paper B in the Journal of Concurrency and Computation: Practice and Experience with the title "Survival of the Synthesis - GPU Accelerating Evolutionary Sound Matching" (Renney, Gaster and Mitchell, 2022). An evolutionary strategy (Rechenberg, 1965) is an optimisation algorithm used for efficiently searching a problem space. In parameter sound matching, the problem space is the mapping between the synthesiser parameters and the output sound. In this chapter, a design for a GPU accelerated FM synthesis parameter matcher is proposed. The performance is evaluated by comparing the GPU against a functionality identical CPU version using a comprehensive benchmarking suite. The application of evolutionary computation for FM synthesis parameter matching is a well-explored area and has proven to be an effective and accurate technique for simple and more advanced synthesisers (Mitchell, 2020) (Horner, 1998*a*) (Yee-King and Roth, 2011). However, a limitation of this method is that it uses considerable computational resources; thus, researchers have been motivated to optimise this approach (Das and Suganthan, 2010). Furthermore, there has been considerable research demonstrating that the massively parallel GPU architecture is appropriate for processing evolutionary strategies (Pospichal and Jaros, 2009). The design presented in this chapter extends the existing idea of GPU accelerated evolutionary strategies to capture the processing of FM synthesis for sound matching.

4.1 Advanced FM synthesisers

Simple FM synthesis, as covered in Section 2.1.3, is used as a fundamental building block in more complex and musically interesting synthesisers. This thesis considers two further synthesisers that build upon Chowning's simple FM synthesis technique. The first has been established as a "real-world" problem by Das et al. and is used to evaluate evolutionary algorithm performance (Das and Suganthan, 2010). This will be referred to as nested modulator FM synthesis (Horner, 1998*b*):

$$y(t) = Asin(ct + I_1 sin(m_1 t + I_2 sin(m_2 t))) \tag{4.1}$$

Where y is the output of the nested modulator oscillator, t is the input variable time, A is the peak amplitude, c is the carrier frequency (rad/s), m is a vector of modulator frequencies (rad/s), and I is a vector of modulation indices. This equation adds a second nested modulation oscillator that modulates the original modulation oscillator of the simple FM structure seen in Equation (2.2). This arrangement requires a total of 6 parameters to control the output sound. The third FM synthesis method considered in the scope of this chapter uses the idea of combining separate FM synthesis components in parallel. For the context of this work, the parallel FM (Mitchell, 2012) will be used:

$$y(t) = \sum_{i=1}^{3} A_i \sin(c_i t + I_i \sin(m_i t)) \tag{4.2}$$

Where y is the output of the parallel FM, t is the input time, A, c, m and I are vectors of the peak amplitude, carrier frequencies (rad/s), modulation frequencies (rad/s) and modulation indices for each FM oscillator. This equation is the summation of three separate simple FM structures, each having four parameters and therefore requires a total of 12 parameters to control the output. Each of these FM synthesiser arrangements will be considered when evaluating the GPU accelerated design, starting with the single FM component.

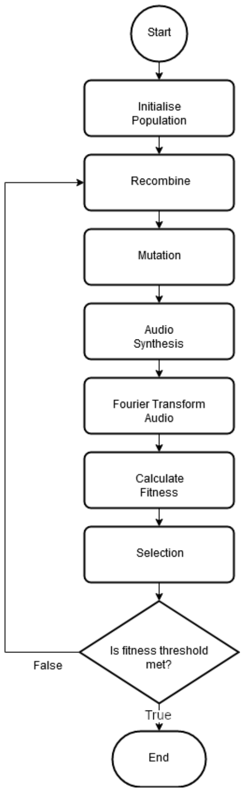


Figure 4.1: Flow diagram for the overall evolutionary parameter matching design flow.

4.2 Evolutionary Strategies

Evolution strategies (ES) (Rechenberg, 1965) are a type of evolutionary algorithm (Beyer and Schwefel, 2002) that are used for optimisation problems, where a suitable solution needs to be found within a search space of numerous possible solutions. Evolution strategies follow ideas inspired by Darwinian evolution and natural selection to create progressively fitter solutions to a problem by iteratively applying mutation and recombination operations on a set or population. Evolutionary strategies can be used for optimisation problems, including sound matching, where the algorithm searches the parameter space of an FM synthesiser to find the parameters that closely replicate a given target sound (Mitchell, 2010).

A population of solutions is made up of individuals. Each individual contains a complete set of synthesis parameters that can be used to generate a candidate sound or solution to the sound matching problem. For example, the simple FM synthesiser corresponds to four parameters and an additional four step-size parameters are used by a self-adaptive mutation operator. Recombination blends two or more parent individuals' genetic material (or parameters) to form new offspring. Figure 4.2, shows how the uniform discrete recombination (Beyer and Schwefel, 2002) operator works, which is one of the standard ES recombination operators used in this work. The value at each position is taken from a random parent in the current population and combined to create a new individual, used in the next generation offspring population.

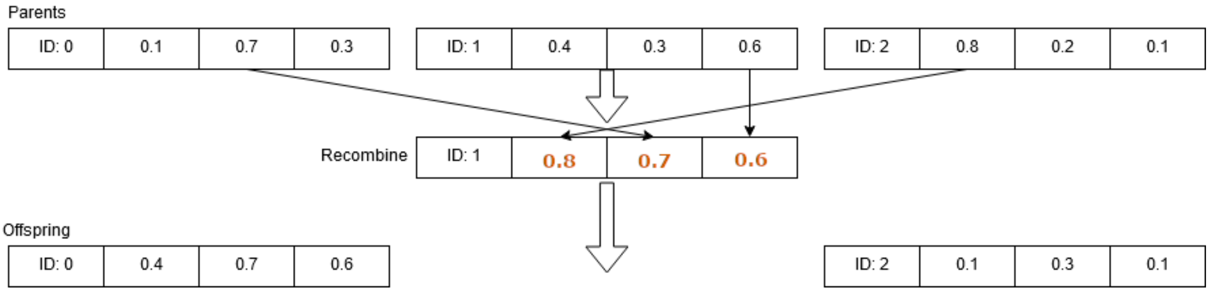


Figure 4.2: Recombine operator working on three example individuals

Figure 4.3, shows how the mutation operator operates. First, the random values shown in red are generated for each element using a pseudorandom number generator and Gaussian Distribution (Patel and Read, 1996), scaled in proportion to the step size. These values are then added to the individual parameters to introduce novelty into the population. Gaussian distribution increases the likelihood of smaller mutations occurring more frequently than larger values. This results in smaller steps in values but occasionally larger steps that improve exploration of the search space by escaping local optima.

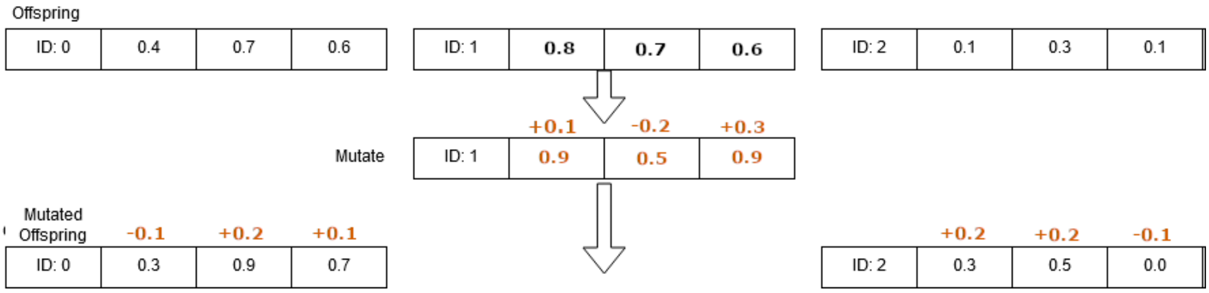


Figure 4.3: Mutation operator working on three example individuals.

4.3 Fitness

The fitness stage of evolutionary strategies is context-dependent and varies depending on the optimisation problem. Here, it requires values to be taken from each individual in the population and used to generate a sample of audio using the chosen FM synthesiser. The generated audio samples are then compared to the target audio, and the similarity between them is used to determine how "fit" each individual is.

Each individual in the evolutionary strategy population has four values when used for simple FM synthesis parameter matching. These 4 values vary between individuals and are mapped to the 4 parameters of the simple FM synthesis algorithm in Equation (2.2): a, c, m and I. A snippet of audio is then generated from the FM synthesiser and stored contiguously in memory, ready for further processing to determine similarity with the target audio.

The similarity between the generated and target audio is determined by first mapping the audio signal to the frequency domain using a Fourier transform. The fast Fourier Transform (FFT) (Frigo and Johnson, 1998) is the de facto algorithm for calculating the Fourier transform of a signal efficiently. However, the FFT algorithm requires the input data to be cyclic, spanning from one end and back to the beginning. Therefore, before the FFT algorithm can process an audio signal, it must first undergo a form of pre-processing; this is known as FFT windowing. FFT windowing aims to taper the edges of the audio samples to enable the FFT to calculate the correct spectrum, ignoring any discontinuity arising from the cyclic interpretation of the audio. The Hann windowing algorithm can be applied to each audio data set associated with individuals $a \in P$ in preparation for the FFT. Therefore, to reduce the occurrence of artefacts as a result of frame boundary discontinuities, a Hann window (Harris, 1978) is used:

$$\omega(n) = 0.5 \times \left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{4.3}$$

Where $\omega(n)$ is the processed Hann value, n is the value of the current input sample, and N is the total number of samples processed. Once the audio signal is prepared using the Hann windowing, it can be processed by the FFT algorithm (Schloss, 2019). The audio signal represented in the time domain is input into the FFT algorithm and processed to produce a series of complex numbers representing the same signal in the frequency domain. FFT is a well understood and supported algorithm that is available in software libraries such as FFTW (Frigo and Johnson, 1998), and these handle the intricacies of the implementation. With each respective individual's audio signal mapped to the frequency domain, the similarity of the signal to the target audio can be calculated. A primary method for comparing the error/difference between two frequency spectra can be achieved using relative spectral error (Beauchamp and Horner, 2003). Studies performed by Beauchamp and Horner (2003) have shown that the relative spectral error delivers the best correspondence to average discrimination data extracted from human listeners when compared with alternative spectral error metrics. The equation for relative spectral error is defined as:

$$rse = \sqrt{\frac{\sum\limits_{b=0}^{N_{bin}} (T_b - S_b)}{\sum_{b=0}^{N_{bin}} T_b^2}} \tag{4.4}$$

Where rse is the output relative spectral error, T is a vector of the target audio frequency spectrum, S is a vector of the synthesised audio frequency spectrum, and N_{bin} is the number of frequency bins produced by spectrum analysis. Here, the relative spectral error rse between two audio signal frequency domains T and S can be calculated. The number of frequency bins N_{bin} is essentially the resolution of the frequencies represented in the audio frequency spectrum; this must be the same for both T and S. Using the rse, each individual's fitness as a candidate for matching with the target signal can be determined. As the rse is the error between two frequency spectra, the fitness is the inverse of the rse. Therefore, an $rse = 0.0$ is a perfect match and an increasing rse measures differences between the signals. Evolutionary strategies typically involve a stopping criteria, such as when a sufficiently "fit" individual is found, the evolutionary strategy iterations stop, and the individual is used as the solution. In the context of this work, the stopping criteria is a set number of iterations/generations as the performance is being assessed, not the accuracy.

The selection stage involves taking a set number of individuals from the offspring and using them in the next generation's parent population. A basic approach is to sort the individuals by fitness and select a number of individuals from the top for the next parent population. A parallel merge sort is an effective sorting algorithm that maps efficiently to the GPU architecture (Davidson et al., 2012).

4.3.1 Data Structure

All data that is processed on the GPU starts with the ES population. Figure 4.4, provides a visual aid to help describe the data structure that is used in the GPU design.

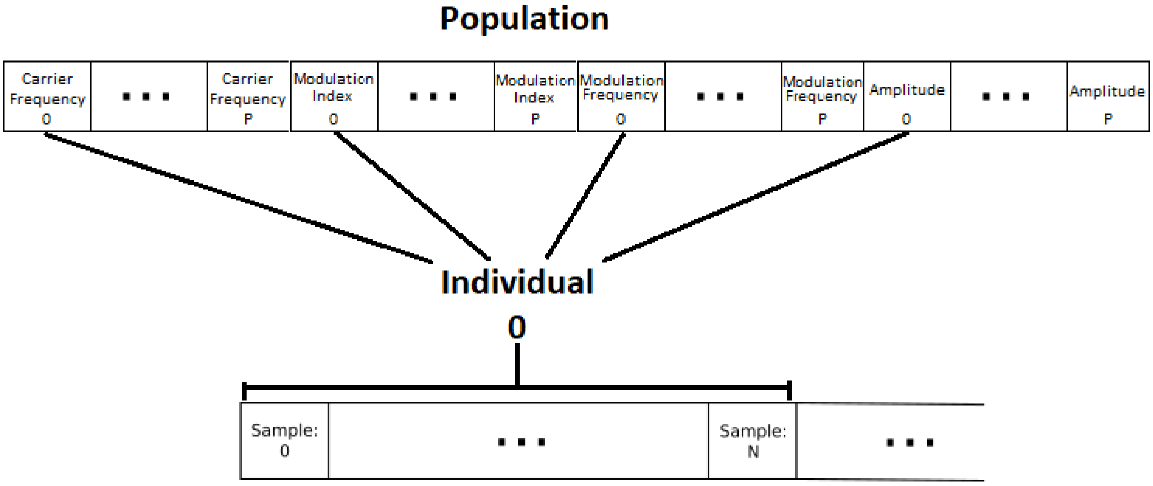


Figure 4.4: The data structure format for the GPU design.

The first buffer indicated by *Population* contains all the individuals' parameters. Although naturally, it seems coherent to place each individual's parameters contiguously in memory, this structure is sub-optimal within the GPU. Therefore, when designing data structures for the GPU, it is more efficient to store all of the first parameters contiguously, then the second parameters, as a structure of arrays. Various researchers have demonstrated this to be the most efficient GPU data layout for comprehensive processing (Mei and Tian, 2016) (Micikevicius, 2012). The parameters of an individual are fetched by indexing into each array of parameters using the individual's ID. Each individual must store an audio block generated from the parameters during the synthesis stage. Each audio block is held contiguously in memory against the generated audio of the following individual's parameters. With this format, two buffers must be allocated to the GPU memory; the population buffer P and the audio samples buffer A. These are calculated as $P = 2 \cdot population\_size \cdot num\_params$ and $A = population\_size \cdot num\_samples$ where $population\_size$ is the number of individuals in the population, num_params is the number of parameters for each individual and $num\_samples$ is the length of the audio sample blocks. As the GPU is optimised for allocating fewer buffers, the two buffers P and A are both allocated as a single large buffer each that use indices to access individual data sets. Further, buffer P is allocated twice the size of the population in order to store a temporary copy of the sorted population when evaluating population fitness. A rotation index can identify an offset to the beginning of the sorted population. This improves performance and memory usage by avoiding unnecessarily copying memory.

Memory coalescing techniques have been applied to optimise GPU memory accesses. An example of correctly programming the GPU to achieve memory coalescing can be found in the following FFT Hann windowing kernel:

```
void applyWindowPopulation(float* audioDate, uint32_t audioLength) {
    int idxWorkitem = get_global_id();
    float mu = ( FFT_ONE_OVER_SIZE - 1) * 2.0f * M_PI;

    int strideFactor = audioLength;
    for(int i = 0; i < POPULATION_COUNT; i++) {
        int idxCoalesced = strideFactor * i + idxWorkitem;
        float fftWindowSample = 1.0 - cos(index  * mu);
        audioDate[idxCoalesced] = fftWindowSample
                                        * audioDate[idxCoalesced];
    }
}
```

Here, Equation (4.3) is implemented on the GPU with an optimized memory access pattern. The index is multiplied with a *stride_factor* equal to the audio wave size. This forces the indices of each workitem to access memory adjacent to one another when processing the windowing. All the indices in the respective workitems then stride to the next audio wave to process when i increments. This means that each workitem can share the memory fetched by its neighbouring workitems without requesting a memory fetch somewhere else in memory itself.

4.3.2 Processing Format

The population size determines the number of workitems the GPU allocates for processing in parallel. These workitems are dispatched to execute each individual's recombination, mutation, audio synthesis, fitness, and selection. Each individual is entirely independent of all other individuals and can execute efficiently without stalling to synchronise data. The FFT processing on the GPU is implemented using the well established, open-source clFFT library (Natarajan, 2013).

During the audio synthesis stage, the chosen FM synthesis arrangement will typically make calls to the trigonometric functions like *sine*. These functions involve a relatively high number of computation cycles. Alternative optimised implementations might use Taylor series expansion or the CORDIC algorithm (Bertrand, 1992). In this design, a look-up table of previously calculated *sine* values can is used (Raghunath and Rambaud, 1999).

As covered in Section 4.3, each individual is tasked with generating an audio block of samples using simple FM synthesis. A pre-calculated table of values approximating *sine* is uploaded to the GPU and indexed instead of the computationally expensive trigonometric function. A finite number of values can be stored in the look-up table, resulting in quantisation. Therefore, sufficient resolution must be used in the look-up table. The wavetable optimised synthesis stage (without interpolation for simplicity) is demonstrated in the GPU kernel code snippet below:

```
int idxWorkitem = get_global_id();

modIdxMulModFreq = I * f_m;

float posOne = 0.0f;
float posTwo = 0.0f;
for(int i = 0; i < WAVE_FORM_SIZE; i++)
{
     curSample = wavetable[posOne] * modIdxMulModFreq + carrierFreq;
     outAudioWaves[idxWorkitem * WAVE_FORM_SIZE + i] = wavetable[posTwo]
                                                         * carrierAmp;

     posOne += wavetableIncrementOne;
     posTwo += (WAVETABLE_SIZE / 44100.0) * curSample;

    if (posOne >= WAVETABLE_SIZE)
         posOne -= WAVETABLE_SIZE;
     if (posOne < 0.0f)
             posOne += WAVETABLE_SIZE;
     if (posTwo >= WAVETABLE_SIZE)
         posTwo -= WAVETABLE_SIZE;
     if (posTwo < 0.0f)
         posTwo += WAVETABLE_SIZE;
}
```

Here, the section of the code that generates the FM synthesis waveform is shown. This essentially implements Equation (2.2) in a suitable way for the GPU. The variable *modIdxMulModFreq* is $I \cdot m$, *carrierFreq* is c, *carrierAmp* is a. Instead of making calls to $\sin(\ldots)$, the wavetable is accessed using *posOne* and *posTwo* that are incremented in such a way as to access approximate pre-computed values of $\sin()$[1]. Variable *idxWorkitem*

[1] This approach is shown to improve performance in the results shown in Section 4.5.6

multiplied with *WAVE_FORM_SIZE* is used to stride through the memory written to in a GPU optimized way.

In an offline application, there are no real-time audio-sound latency requirements. Therefore, processing larger buffers or batches of processing has fewer restrictions. However, GPUs have limited memory, meaning there is an upper limit on the buffer length of the population buffer P and audio samples buffer A. This can quickly become the case in this evolutionary sound matching application, as the population size is a controllable parameter that can exceed GPU memory limits. The amount of GPU memory resources depends on the system's hardware. Taking the systems in Table 3.3 as examples, the *Nvidia GeForce RTX 2080 Ti* has 11GB of memory whilst the *Radeon 530* has only 4GB. In the *Mid-range Laptop* system, the *Intel UHD Graphics 620* shares 8GB of RAM with the CPU. In order to avoid misusing the GPU memory, the GPU design breaks up data into manageable, equally sized blocks, loads them onto the GPU and processes them one after another. Further, batching processes to the GPU follows the same technique as buffer sampling, increasing performance. The pseudo-code for the batching is given below:

```
void parameterMatchAudio(float* aTargetAudio, uint32_t aTargetAudioLength)
{
    blockSize = objective.audioLength;
    blocks = aTargetAudioLength / blockSize;

    for (uint32_t i = 0; i < blocks; i++)
    {
        setTargetAudio(&aTargetAudio[i*blockSize], blockSize);
        initPopulationCL();
        executeAllGenerations();
    }
}
```

An additional advantage to using the batching technique is that the system can easily be extended in the future to involve analysing dynamically changing sounds. This means if the characteristics of a sound are changing over time, analysing each block can identify the parameters necessary to match the changing sound in each block. If this advancement was explored, there would be a set of parameters for each audio block analysed.

4.4 Benchmarking Targets

The benchmarking suite involves the autonomous execution and performance profiling of a collection of planned tests. This chapter targets the same systems from Table 3.3 and uses the same from Chapter 3 except the windows 10 version is 20H2[2] and the CUDA SDK version used is 11.1.0. Configurations of parameters are exhausted, and the results are collected and written to files for analysis. The GPU implementation will synchronize between each OpenCL call to confirm the action is finished to measure the execution time accurately. The benchmarking suite is open-source and publicly available online[3]. Three different implementations have been developed for comparison:

CPU Serial - To establish the minimal baseline performance, a serial single-core implementation targeting the CPU.

CPU OpenCL - A parallel implementation targeting the CPU using OpenCL.

GPU OpenCL - OpenCL implementation that targets the parallel processing on the GPU.

The benchmarking suite is designed to measure the overall execution for default parameters and is used to measure execution time when scaling controllable parameters. The controllable parameters affect the performance and accuracy of the algorithm and therefore play a crucial role. Unless stated, the default parameters will have the values shown in Table 4.1. The default parameters have been chosen as they provide solutions with acceptable fitness and give sufficient benchmarking processing to profile and consider. Instead of using a fitness threshold as the stopping criteria, the benchmarking suite will execute a fixed number of generations. This ensures parity between results by guaranteeing that all implementations processed an equal amount of computation, mitigating any stochastic variations. Using the default parameters, the best solution found by the application converges on the exact solution with fitness 0.0 for almost all test runs.

Overall Execution - The overall execution time for the program to complete is the primary concern. The overall execution time includes all stages shown in Figure 4.1.

Program Stage Execution - In order to highlight the computational implications at each stage of the processing, the stages in Figure 4.1 are independently timed. This target exposes potential bottlenecks and the most intensive stages.

[2] https://learn.microsoft.com/en-us/windows/whats-new/whats-new-windows-10-version-20h2

[3] https://github.com/Harri-Renney/Survival_of_the_Synthesis-GPU_Accelerated_Frequency_Modulation_Parameter_Matcher

Parameter	Value	Notation
Number Generations	1000	G
Number Parameters	4	D
Parent Population Size	1024	P
Offspring Population Size	7168	O
Target Audio Length	2048	T
Audio Block Size	2048	N
GPU Workgroup Size	32	W

Table 4.1: Default benchmarking parameters

Audio Analysis Block Size - The size of each block of audio analysed at a time by the application. The block size controls the number of data transfers between CPU and GPU. Therefore, this target focuses on evaluating the impact of scaling the audio block size.

Population scaling - It has been shown in Tsoy (2003) that increasing the population size in evolutionary algorithms can produce fitter solutions in a more complicated problem space. Therefore, measuring the performance as population size is increased will expose the ability to scale.

Optimisations On/Off - The GPU specific optimisations outlined in the design will be analysed by comparing the performance with the optimisations turned on and off.

Discrete vs Integrated - The discrete and integrated GPUs have unique relationships with the CPU and memory. Therefore, comparing the results of these two GPU types will expose critical differences in performance.

4.5 Results

In this section, the results collected across the benchmarking suite are presented, along with a discussion of the results. This thesis considers the salient results collected from the same set of systems outlined in Table 3.3 [4]. Particular focus on the results is given to the High-End NVIDIA GeForce desktop that better reflects the expected audience of synthesis parameter matching. The *mid-range laptop* is used to collect results from integrated and discrete GPUs and can also be compared to the *high-end* desktops.

[4] The full collection of results, including another High-End NIVIDA Titan setup, can be found at: `https://muses-dmi.github.io/benchmarking/benchmarking_database_survival_of_the_synthesis`

Implementation	Total Execution Time (s)
CPU Serial	409.98
CPU OpenCL	28.33
GPU OpenCL	3.19

Table 4.2: Total execution time of the application for the CPU and GPU implementations. **Parameters** = Default **System** = High-end NVIDIA GeForce

4.5.1 CPU vs GPU

The results displayed in Table 4.2 include the total time across all three implementations using the default parameters for the High-End NVIDIA system. The CPU serial version takes the longest at 409.98s, whilst the OpenCL CPU and GPU recorded 28.33s and 3.19s, respectively. Using the CPU's SIMD parallel vector processors in OpenCL CPU, there is a clear 14× improvement over the serial version. GPUs advance this data-parallel processing further, achieving a speedup of 128× over the serial CPU version. This demonstrates that not only are ES and FM synthesis suitable for data-parallel processing, but they are also suitable in combination using the design proposed in this paper. A more direct comparison between the CPU and GPU is demonstrated when the CPU is used to its full potential in parallel using OpenCL. The *Intel Core i7-9800X* CPU profiled with only 8 cores has a peak floating point performance of 972GFLOPS compared to the *Nvidia GeForce RTX 2090 Ti* GPU with 4352 cores and 13450GFLOPS. Therefore, provided the application is suitable for parallel processing, theoretically the GPU should be faster and could be as high as 13.8× faster than the CPU version. Refering back to the results from Table 4.2, the GPU was observed to have a speedup of 8.88× over the CPU for this particular setup. This is expected as the GPU architecture is designed to maximise the data-parallel throughput for this kind of suitable application across thousands of available GPU cores that a CPU does not have.

4.5.2 Implementation Stages Compared

The execution time for each stage in the application on the High-end NVIDIA system has been recorded and displayed in Figure 4.5. The execution time is marked in *ms* on a logarithmic scale using the default parameters. When comparing each implementation, the stages follow a similar pattern of the CPU serial taking the most time, followed by CPU OpenCL and finally GPU OpenCL. Particularly, there is a significant improvement

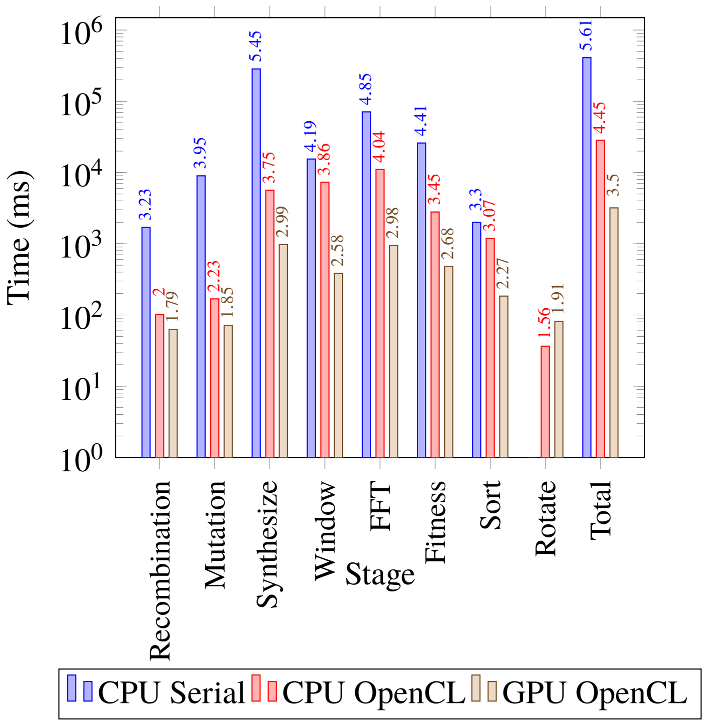


Figure 4.5: Execution time of each stage across the CPU and GPU implementations.
Parameters = Default **System** = High-end NVIDIA GeForce

on the GPU for the following *synthesis*, *Window* and *FFT* stages. Therefore, the results highlight the significance of optimising these domain-specific stages. The data-parallel version improves over the CPU versions in all stages, except for the *Rotate* stage. The measurement recorded for the CPU Serial was 0.3544ms, A negligible as it only involves updating a variable in CPU memory. In contrast, the OpenCL versions involve more measurable overhead in the *Rotate* stage. For example, the GPU version requires updating the rotation index in GPU; this involves a data transfer over the PCI interface to update the variable in GPU memory. This is a stage the CPU will naturally surpass the OpenCL versions. However, relative to all the other stages, this additional overhead is small and exceeded by the improvement in all other stages.

4.5.3 Hardware Systems Compared

This section compares the GPU OpenCL Version results when scaling the population for both hardware systems. The High-End NVIDIA desktop is expected to execute faster for all population sizes as it utilizes the more powerful NVIDIA GeForce RTX 2080 Ti, whilst the Mid-range Laptop contains a modest AMD Radeon 530. It can be seen in

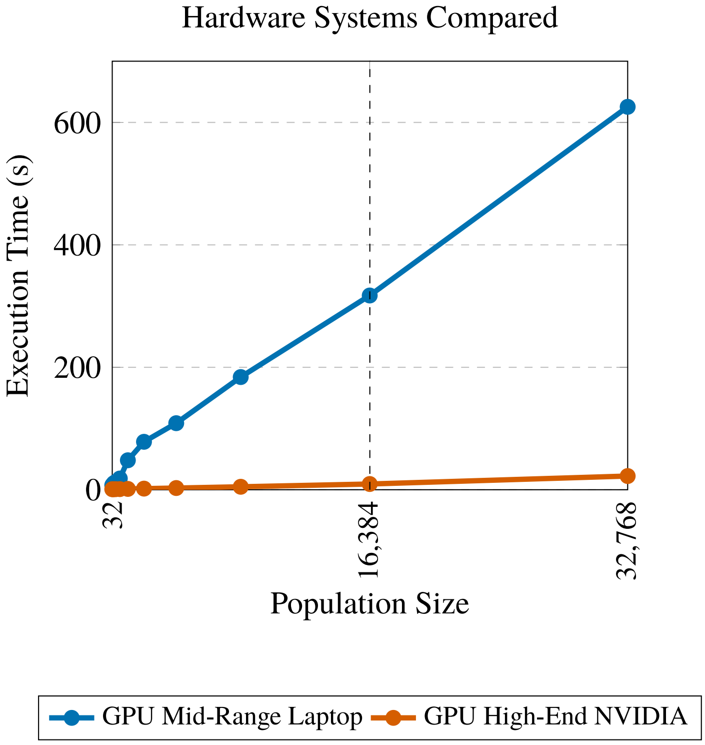


Figure 4.6: Execution time when scaling the population size for the High-End and Mid-range systems. **Parameters**: G = 1000, P+O = Scaled **System** = High-end NVIDIA GeForce & Mid-range Laptop

Figure 4.6 that the NVIDIA 2080 GPU is far more capable of handling the application and scaling the population size up to 32768. Whilst the AMD 530 GPU has a roughly directly proportional increase in execution time with population size, the NVIDIA 2080's numerous streaming processors with higher clock speeds can accommodate the extra individuals in parallel and has a far more gradual increase. This demonstrates that increasing the number of faster streaming processors improves the performance of these data-parallel tasks that depend on higher throughput.

4.5.4 Kernel Execution Time Ratio

Figure 4.7 shows the relative execution times for every stage of the GPU OpenCL implementation running with default parameters. This highlights where the majority of the processing takes place. The stages processing the ES population: *Recombine*, *Mutate* and *Sort* together only account for less than 13% of the time. These stages execute quickly as they only process the population $(P+O) \cdot D$, which is $(1024+7168) \cdot 4 = 32,768$ floating-point values for the default parameters. The stages processing audio for the population: *Synthesis*, *Window*, *CLFFT* and *Fitness* take considerably longer at 87.43% of the time.

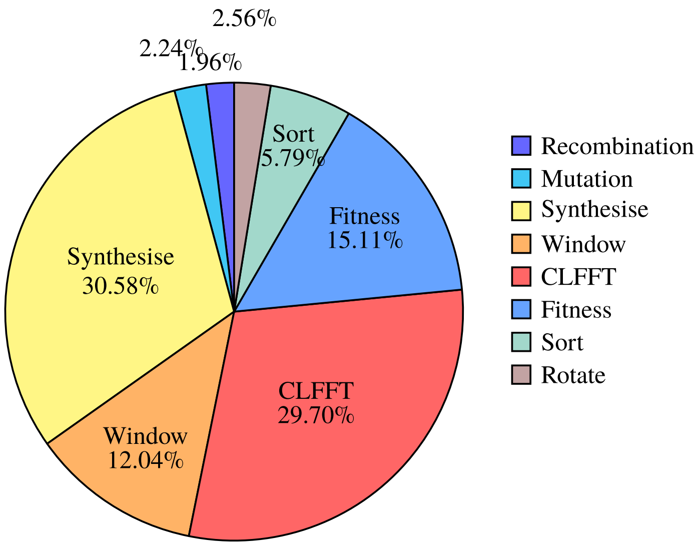


Figure 4.7: Ratio of execution time across stages in the GPU OpenCL implementation.
Parameters = Default **System** = High-end NVIDIA GeForce

The audio stages require processing $(P+O)\cdot N$ which is $(1024+7168)\cdot 2048 = 16,777,216$ floating-point values. The majority of the execution time is taken up by the synthesis and CLFFT stages, taking around 30% of the time each. This highlights the importance of the design as it efficiently processes the evolutionary algorithm stages and incorporates the domain-specific processes that take up the majority of the execution time.

4.5.5 Integrated vs Discrete

Figure 4.8 shows the results comparing the integrated Intel and AMD discrete GPU in the mid-range laptop. It can be seen that initially when the population size is small, the integrated GPU performs better than the discrete GPU. This is because the PCI-e system bus involves communication overhead when transferring between the discrete GPU and the CPU. As a result, the communication overhead exceeds the benefits of using a discrete GPU at the lower population sizes. The integrated GPU shares unified memory space with the CPU, avoiding this overhead. However, at around population size 2048 and beyond, the benefit of the powerful discrete GPU begins to exceed the communication overhead and scales better than the integrated GPU.

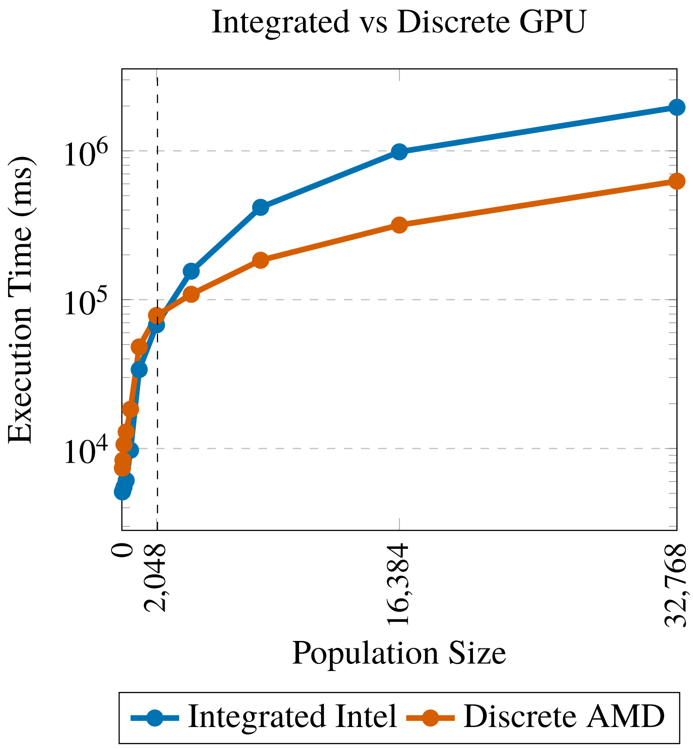


Figure 4.8: Execution time when scaling the population size for the GPU OpenCL implementation targeting the integrated and discrete GPU.
Parameters: Default, G = 1000, P+O = Scaled **System** = Mid-range Laptop

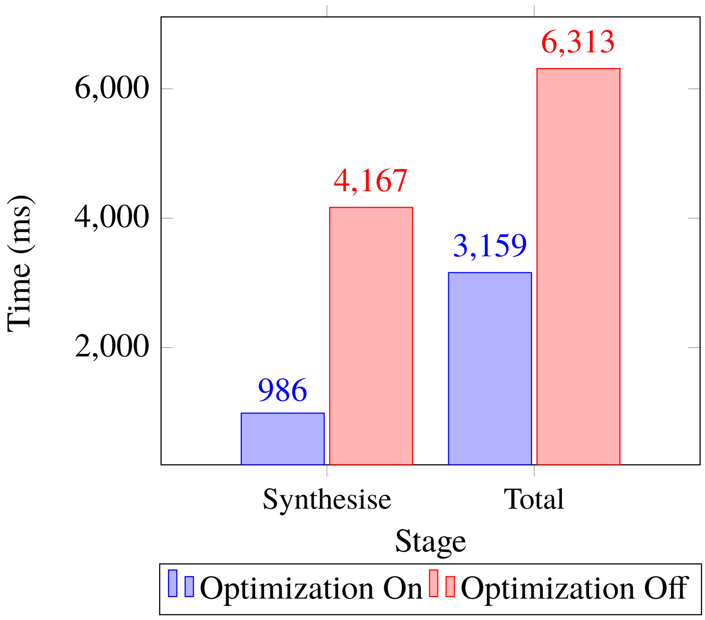


Figure 4.9: The total and synthesis stage execution time when the FM wavetable lookup optimization is on and off.
Parameters = Default **System** = High-end NVIDIA GeForce

4.5.6 Optimized vs non-optimized

Figure 4.9 presents a comparison between the results of the GPU design with and without optimisations. For example, one of the key optimisations used is a pre-processed lookup table for FM synthesis. This optimisation significantly improves the performance of the synthesis stage. Considering in Figure 4.7, it can be seen that the synthesis stage was one of the most time consuming stages. Therefore, the lookup table optimisation offers a considerable 4× speedup in the synthesis stage for the default parameters. This demonstrates the importance of context-specific optimisations that are usually introduced in the *fitness* stage.

4.5.7 Population Scaling

Figure 4.10, provides a comparison between the implementations on the High-End NVIDIA system when scaling the population size. The execution time is plotted on a logarithmic scale in seconds. The CPU Serial version initially starts at a higher execution time at 1.87s compared to the GPU OpenCL version taking 0.89s. As the population size scales, the execution time for the CPU versions is significantly higher, especially considering that this is a logarithmic graph. The GPU version's execution time increases gradually

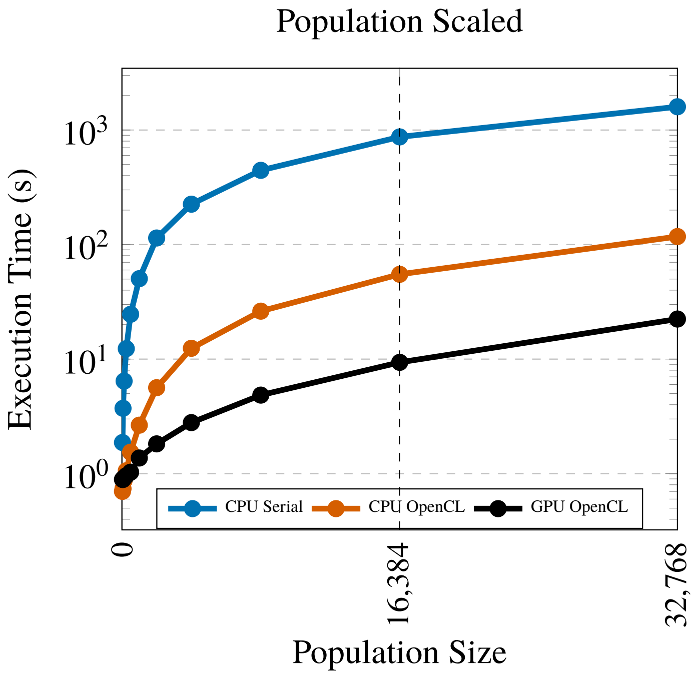


Figure 4.10: Execution time when scaling the population size for all implementations.
Parameters: G = 1000, P+O = Scaled **System** = High-end NVIDIA GeForce

as it can process far more individuals in parallel. At a population size of 16,384, the GPU version runs at approximately 10s, while the CPU OpenCL version records 55s, roughly 5× slower and the CPU Serial at 870s, 87× slower. The GPU continues to show it provides a speedup even at higher population sizes, beyond the 8192 size used in the default configuration. The improved GPU performance brings the application closer to a practical, real-time tool. However, complex synthesizers that may require substantial population sizes will still see considerable execution times that may exceed the practical limits of a real-time tool. This is subject to future investigations outside of the scope of this thesis, which will begin to focus on real-time physical modelling synthesis.

4.5.8 Audio Analysis Block Size Scaling

Figure 4.11 shows the total time of the GPU OpenCL application when scaling the audio block size on the NVIDIA 2080 discrete GPU. The audio blocks size determines the number of audio samples processed and analysed on the GPU at a time. Decreasing the audio block size splits the target audio into further separate blocks for analysis. There are two reasons a smaller block size might be considered. First, the GPU memory

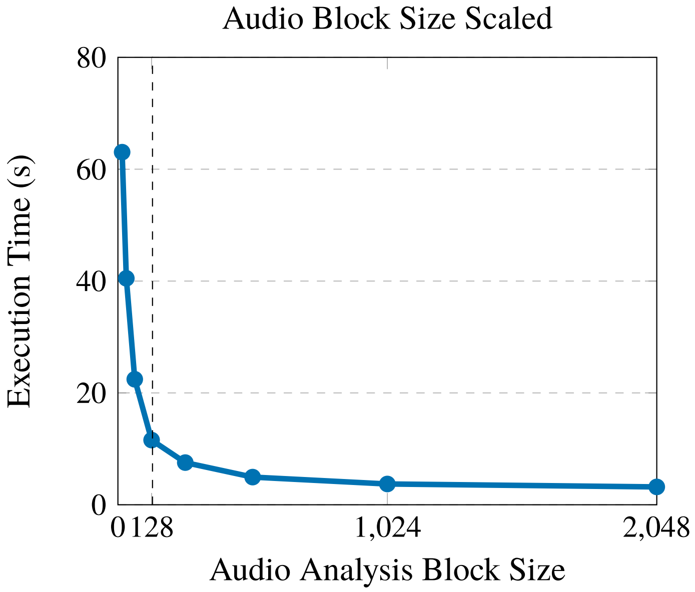


Figure 4.11: Execution time when scaling audio block size for the GPU OpenCL implementation. **Parameters**: Default, N = Scaled **System** = High-end NVIDIA GeForce

would not be able to accommodate the larger blocks of audio. Second, this approach is advantageous if the audio being analysed is dynamic, such that the synthesiser parameters change with time. However, decreasing the audio block size increases the number of dispatches to the GPU, which increases the communication overhead over the PCI bus and also decreases the possible resolution of the FFT calculation and therefore effects the accuracy of the similarity calculation that the fitness function depends on. The overhead adds up considerably, resulting in severely reduced performance when the audio block size is below 128. Beyond the block size of 128, the performance reaches a significantly improved state and the FFT resolution begins to increase and only up to audio block size 512 and 1024 does it begin to reliably calculate the audio similarity to accurately match the target sound. Continuing to increase the audio block size to the target audio size has a less significant performance growth. These results show that although a smaller block size can be used, a size below 128 has a significant impact on performance and below 1024 the accuracy is not sufficient to reliably match the target sounds. Therefore, future developments of dynamic audio samples that involve frequent changes in timbre will be a challenging additional feature to support as the reduced FFT resolution and performance implications must be resolved.

4.5.9 Advance FM Synthesisers

To demonstrate the effectiveness of the GPU optimised framework, two further advanced FM synthesisers have been used in place of the simple FM synthesis. The first is double series FM, and the second is triple parallel FM synthesis. Figure 4.12 presents the results of the simple, double series and triple parallel FM synthesisers on all three implementations. Again, a large population size of 32768 has been used, and the time in seconds is again plotted on a logarithmic scale. For the CPU Serial implementation, both the double series and triple parallel synthesisers take considerably longer with an additional 10699s and 11024s, respectively. In contrast, the GPU OpenCL implementation only requires an additional ≈ 4s for the double series and ≈ 6s for triple parallel FM. This demonstrates that the GPU accelerated framework for handling the evolutionary computation for parameter matching supports more advanced forms of FM synthesis. Although this cannot be extrapolated to all possible arrangements of FM synthesis, for these two examples, it continues to improve performance over a naive serial implementation.

4.6 Offline Evaluation

This chapter has presented the design for a GPU optimised algorithm for parameter matching with several FM based synthesisers. The results suggest that the GPU can be used as a hardware accelerator for offline processing without any concern for meeting real-time requirements for suitable processes. Therefore, parameters can be configured on the GPU to maximise throughput without any concern to meet particular sample rate or latency requirements. For the default parameter configuration on a high-end desktop, the GPU had a speedup of 128× over the serial CPU version and 8.88× over the parallel CPU version. This highlights the significant improvement potential when using parallel processing in general, but also the massively parallel architecture of the GPU in comparison to the CPU. The population size of the ES has a significant impact on the execution time of all implementations. However, the GPU version was impacted to a lesser extent than the other implementations, suggesting that the proposed GPU design can process larger population sizes more rapidly, making it more suitable for practical use by music creators. The results show that the performance benefits apply to simple FM synthesis and extend to support more advanced arrangements such as double series and triple parallel FM synthesisers. Another controllable GPU parameter is the data block size; this was scaled and shown to harm the GPU processing time when block sizes

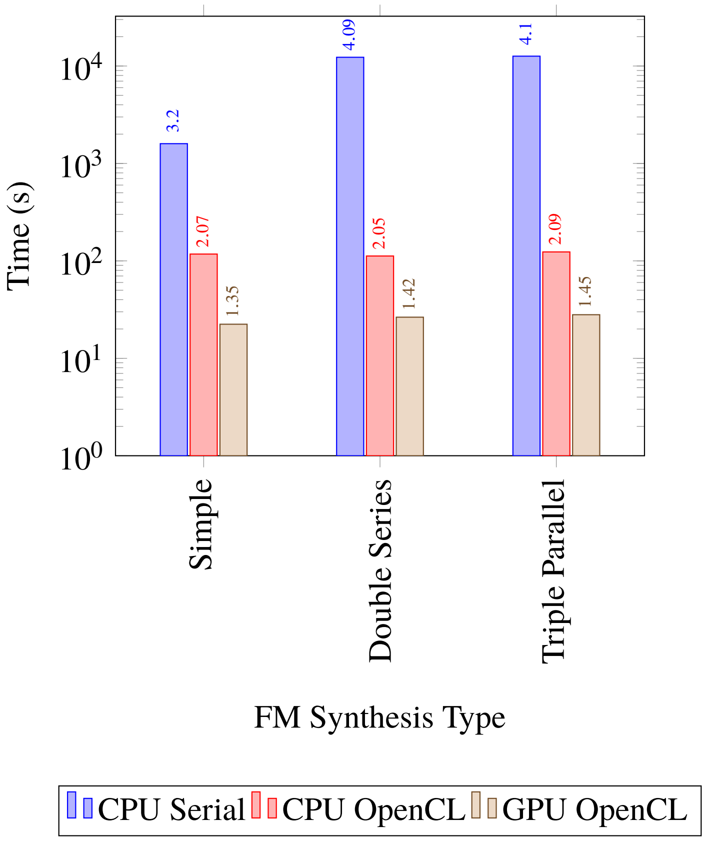


Figure 4.12: Execution time of three synthesis types for the CPU and GPU implementations. **Parameters**: Default, P+O=32768 **System** = High-end NVIDIA GeForce

below 128 are used. This is a weakness of the GPU design caused by the data transfer overhead between GPU and CPU. An optimised design needs to be adapted to better support this use case. The exclusive use of the GPU for offline audio processing ends in this chapter; the rest of this thesis is primarily concerned with contributing to real-time physical modelling audio synthesis.

Chapter 5

GPU Accelerated Physical Modelling

In Chapter 3, Section 3.1.3, the *complex buffer synthesis* test involved a physical modelling sound synthesis method. This chapter aims to describe the design of this physical model in more detail and then evaluate the performance of processing it on the CPU and the GPU. Two CPU versions are presented, a naive serial version to consider as the baseline and a parallel SSE vector processor version; these will be compared against three GPU versions. The contributions of this chapter draw from the publication (Paper [C]) "OpenCL vs: Accelerated Finite-Difference Digital Synthesis" published at the conference of the International Workshop on OpenCL and SYCL (Renney et al., 2019).

5.1 Physical Model Definition

This finite-difference physical model system will operate in a two-dimensional Cartisian grid. Considering a rectangular system with side lengths L_x and L_y (in m), the state is described by *u(t,x,y)*. The system operates in $t \geq 0$ and $(x,y) \in \mathscr{D}$ where the domain $\mathscr{D} \in [0,L_x] \times [0,L_y]$ is two-dimensional. As covered in Section 2.2.3, the state variable can be discretised to a two-dimensional function as $u(t,x,y) \approxeq u^n_{l,m}$ and independent variables discretised as $x = lh$, $y = mh$ and $t = nk$ where h is the size of the spatial step (For simplicity, in both x and y directions) and k is the size of the time step. Therefore, the temporal index $n \in \mathbb{N}^0$ and spatial indices $l \in \{0,\ldots,N_x\}$ and $m \in \{0,\ldots,N_y\}$ are used to index into the state function in time t and space x and y. N_x is the number of grid points in x dimension, N_y the number in the y dimension. The model is excited with a signal defined as $\delta(x_i - x, y_i - y)e$ where $x_i, y_i \in [0,1]$ identifies a position for an impulse signal to excite the physical model. Samples are then recorded from the model and loaded into the output audio buffer at a single *mic* position x_o, y_o where $x_o, y_o \in [0,1]$.

With the system variables established, the physical equation for simulating the audio acoustics in two dimensions can be defined. Taking the two-dimensional wave Equation (2.12), a simple form of damping can be added by including a first-derivative term (the same way friction enters a vibrating mechanical system (Langtangen, 2016*a*, p. 44)). Using these terms, the state variable $u^n_{l,m}$ can be defined as:

$$u_{tt} + bu_t = c^2 \Delta u - \delta(x_i - x, y_i - y)e \tag{5.1}$$

Where b is the friction coefficient that introduces damping ($0 < b < 1$), c is the speed of sound propagation. The state function can then be discritised using a second-order central difference for u_{tt}, a first-order central difference for u_t and after expanding Δu, a second-order central difference for u_{xx} and u_{yy}. This leads to the formation of the following finite-difference equation:

$$\delta_{tt} u^n_{i,j} + b\delta_{\cdot t} u^n_{i,j} = c^2(\delta_{xx} u^n_{i,j} + \delta_{yy} u^n_{i,j}) - j_{l,0}(x_i, y_i)E(t) \tag{5.2}$$

Where $j_{l,0}(x_i)E(t)$ is the discrete zeroth-order spreading function for exciting the model with the impulse signal E. This can then be rearranged to develop a recursively solvable explicit finite-difference form, similar to Equation (2.19), but including the central finite-difference for the damping term:

$$u^{n+1}_{x,y} = \frac{2u^n_{x,y} - (\mu - 1)u^{n-1}_{x,y} + \lambda^2(u^n_{x+1,y} + u^n_{x-1,y} + u^n_{x,y+1} + u^n_{x,y-1} - 4u^n_{x,y}) - j_{l,0}(x_i, y_i)E(t)}{1 + \mu} \tag{5.3}$$

Where two parameters $\mu = \frac{bk}{2}$ and $\lambda = \frac{ck}{h}$ are exposed. The stability condition ($\lambda \leq 0.5$) must be satisfied to maintain a stable system.

A Dirichlet-Neumann hybrid boundary condition is used at the grid points identified as boundary points. This is employed in the context of this physical model as:

$$u_{u,d,l,r} = \begin{cases} u^n \gamma, & if boundary \\ u^n_{u,d,l,r} & else \end{cases} \tag{5.4}$$

Where γ is used to transition between a fully clamped Dirichlet condition ($\gamma = 0$) to a free edge Nuemann condition ($\gamma = 1$)

```
for i = 1 to gridHeight
    for j = 1 to gridWidth
        ixy = i * gridWidth + j
        ixMy = i * gridWidth + j - 1;
        ixPy = i * gridWidth + j + 1;
        ixyM = (i-1) * gridWidth + j;
        ixyP = (i+1) * gridWidth + j;

        //Calculate based on boundary.
        leftNeighbour = modelGrid[ixMy] * (1-boundaryGrid[ixMy])
                        + modelGrid[ixy] * boundaryGrid[ixMy];
        //For all neighbours
        ...

        nPOne[idx] = timestep(n[idx], nMOne[idx], leftNeighbour,
                                rightNeighbour, upNeighbour,
                                downNeighbour)
    end for
end for
rotateGrids(nMOne, n, nPOne)
```

Figure 5.1: Serial processing of two-dimensional finite-difference simulation.

5.1.1 Serial CPU Version

The serial CPU version works by visiting each finite-difference point in the grid, iterating over $l \in \{0, \dots, N_x\}$ and $m \in \{0, \dots, N_y\}$, applying the explicit scheme update Equation (5.3). This is done in sequence, one point after another, each calculation independent of one another, making this suitable for SIMD type processing. Therefore, even though it is naively processed in serial, it is suitable for parallel processing.

Figure 5.1 presents the pseudocode for the recursive application of Equation (5.3) to all grid points one after another in serial. Two *for* loops are used to iterate over each grid point where all the necessary indices are calculated, and the neighbouring values are collected from the grid. The effect of the conditional statement checking the boundary modifies the neighbouring values that have the effect of the boundary condition from Equation (5.4) on the calculation of the grid point. The function *timestep* computes the explicit scheme Equation (5.3) using the grid data and coefficients not included in the code snippet. After the new state of the system is calculated, grids are rotated to consider the grids correctly for the next timestep.

Although this version does not take advantage of any parallel processing, all other possible optimisations are considered and applied in the serial case. This includes pointer switching, data cache alignment and avoiding redundant calculations by simplifying the parameters into coefficients. These optimisations are covered in detail within the context of GPUs later in Section 6.

5.1.2 Parallel CPU Version

The parallel CPU version uses available vector processors to process the finite-difference calculations in parallel. Intel's default AVX vector processing instruction set supports 128 bits[1] of register memory, enabling the same instructions to be applied in parallel to 4 different floats. Using the AVX instruction set requires the data to be handled differently and loaded into the vector processor registers. Figure 5.2 demonstrates the additional steps involved to process using AVX Intel intrinsic. Again, the grid is iterated over using 2 *for* loops for each dimension. Notice how the *for* loops stride over 4 grid points at a time, this is because the AVX vector registers can process 4 floats at a time. The Intel intrinsic function *_mm_load_ps* is used to load 4 floats from main memory into the AVX register *__m128*. Within *timestepVector*, the AVX Intel intrinsic functions for doing all mathematical operations must be used to operate on the vector register memory, such as *_mm_add_ps* and *_mm_mul_ps*.

It can be seen that the program source code becomes considerably more complicated to adhere to the CPU vector processors. Data must be loaded to registers in segments, compiler-specific intrinsic functions must be applied for calculations, and results must be written back from register to main memory to increment the timestep. Further, complications now arise at the boundaries, as the boundary now affects 4 grid points instead of 1 as a detected boundary applies the boundary effect to the whole vector register instead of a single grid point. This will become an issue as the environment being modelled becomes more complex, with multiple models with interacting boundaries. Supporting more advanced models would appear to break down using vector extensions.

In contrast, the GPU is a SIMT device, meaning there are threads running sets of identical instructions, but the instructions do not need to be the same across all the data as the threads of instructions can be assigned to process appropriate regions of the data in parallel with particular instructions. Furthermore, the GPU hardware and interfaces are designed to handle the thread instruction automatically and optimally. This means

[1] The original AVX instruction set has been used so that the benchmarking suite can support the widest range of hardware. However, a comprehensive suite would support and report on the more recent AVX2 and the latest AVX-512 SIMD instruction sets.

```
__m128 nVector;
__m128 nMOneVector;
__m128 nPOneVector;
__m128 leftNeighboursVector, rightNeighboursVector, upNeighboursVector,
       downNeighboursVector;
for i = 1 to gridHeight-4 when i += 4
    for j = 1 to gridWidth-4 when j += 4
        nVector = _mm_load_ps(n[i][j]);
        nMOneVector = _mm_load_ps(nMOne[i][j]);

        //Calculate based on boundary.
        leftNeighboursVector = calculateBoundaryVector(n[i][j],
                                                        n[i-1][j]);

        //For all neighbours
        ...

        nPOneVector = timestepVector(nVector, nMOneVector,
                                        leftNeighboursVector,
                                        rightNeighboursVector,
                                        upNeighboursVector,
                                        downNeighboursVector)
        _mm_store_ps(nPOne+i, nPOneVector);
    end for
end for
rotateGrids(nMOne, n, nPOne)
```

Figure 5.2: AVX vector processing of two-dimensional finite-difference simulation.

the GPU avoids the complications involved with purely SIMD processing units, like the vector processors on the CPU. The next version looks at the GPU design and how it can be used for processing physical models.

5.1.3 OpenCL Version

The OpenCL version conforms to the OpenCL standard kernel language. This language takes a general-compute format for describing processes. This is achieved by dispatching many workitems to the GPU that are then processed in a parallel way such that the workitem IDs are used for indexing into the grid memory. Figure 5.3 presents the pseudocode for the OpenCL GPU program. Here, no nested for loops are needed as each workitem the code describes obtains the indices from the two-dimensional workitem IDs.

```
int ixy = (get_global_id(1)) * get_global_size(0) + get_global_id(0);
int ixMy = (get_global_id(1)-1) * get_global_size(0) + get_global_id(0);
int ixPy = (get_global_id(1)+1) * get_global_size(0) + get_global_id(0);
int ixyM = (get_global_id(1)) * get_global_size(0) + get_global_id(0)-1;
int ixyP = (get_global_id(1)) * get_global_size(0) + get_global_id(0)+1;

//Calculate based on boundary.
leftNeighbour = modelGrid[ixMy] * (1-boundaryGrid[ixMy])
                + modelGrid[ixy] * boundaryGrid[ixMy]

//For all neighbours
...


nPOne[ixy] = timestep(n[ixy], nMOne[ixy], leftNeighbour, rightNeighbour,
                      upNeighbour, downNeighbour)

rotateGrids(nMOne, n, nPOne)
```

Figure 5.3: OpenCL finite-difference simulation calculating indices from OpenCL workitem ID.

Besides this alternative approach, the GPU general-compute code follows in a similar vein as the CPU code but can be processed by the GPU in parallel across the compute units. The *rotateGrids* step is not included in the GPU code as it is updated and pushed to the GPU from the CPU.

An additional OpenCL version is developed that utilises GPU local memory. With this approach, the idea is to load grid data from global memory to local memory once, then access the local memory grid within the workgroup for calculations. This aims to exploit the relatively faster GPU local memory over direct access to the global memory for calculations. Figure 5.4 demonstrates how a local memory grid *gridLocal* is created and used to hold intermediate calculations in faster access memory within workgroups.

5.1.4 OpenGL Version

The OpenGL version uses the design proposed by Zappi et al. (2017) where the physical modelling is processed using the graphics pipeline (Figure 2.5). This requires mapping the physical model and audio data structures into the graphics domain exposed by OpenGL. For instance, OpenGL expects pixel data in a framebuffer ready for rendering to a display. However, in this design, a custom texture framebuffer is created that remains on the GPU.

```
int ixyLocal =  get_local_id(1)     * get_local_size(0)
                    + get_local_id(0);
int ixMyLocal = get_local_id(1)     * get_local_size(0)
                    + get_local_id(0)-1;
int ixPyLocal = get_local_id(1)     * get_local_size(0)
                    + get_local_id(0)+1;
int ixyMLocal = (get_local_id(1)-1) * get_local_size(0)
                    + get_local_id(0);
int ixyPLocal = (get_local_id(1)+1) * get_local_size(0)
                    + get_local_id(0);

gridLocal[ixyLocal] = grid[ixy];
...
```

Figure 5.4: OpenCL calculating and loading global memory into local memory using local ID.

The physical model state is loaded into graphics textures, where the value of each grid point is contained within the texture's pixel data. The GPU multi-streaming processors execute in parallel, calculating each pixel value by applying Equation (5.3). Figure 5.5 illustrates the entire texture layout and the sections it consists of to support the physical modelling environment. Here, Tex0 and Tex1 pixels are N_xxN_y and hold the state of the system at n and $n-1$. For each timestep, *Tex0* and *Tex1* alternate between which of the timesteps they represent. As the values for the next timestep $n+1$ are calculated, they overwrite the $n-1$ pixels as they are not needed in future calculations. *Tex2* pixels is a single row that contains the audio sample buffer. From an output position in the physical model, samples are copied to the *Tex2* audio buffer region, and this region can then be transferred back to the CPU for audio playback. *Tex3* pixels is an empty and unused region of the texture that separates the model grids *Tex0* or *Tex1* from the audio buffer to prevent audio samples leaking back into the system when accessing neighbouring values $u_{u,d,l,r}$.

The OpenGL GPU program is coded using GLSL, where most of the processing takes place in the GPU fragment shader. Figure 5.6 provides the key content of the fragment shader used for the OpenGL version. Here, the textures must be loaded into vec4 types using the *texture* function. This takes the texture object and the coordinates of the fragments to retrieve from the texture. The *vec4* type is used as it can contain the 4 floats associated with textures in the RGBA format. This means physical model values are mapped to specific RGBA channels for further calculations. The neighbouring values $u_{u,d,l,r}$ are concisely identified and calculated in the assignment of *pLRUD*. With all of

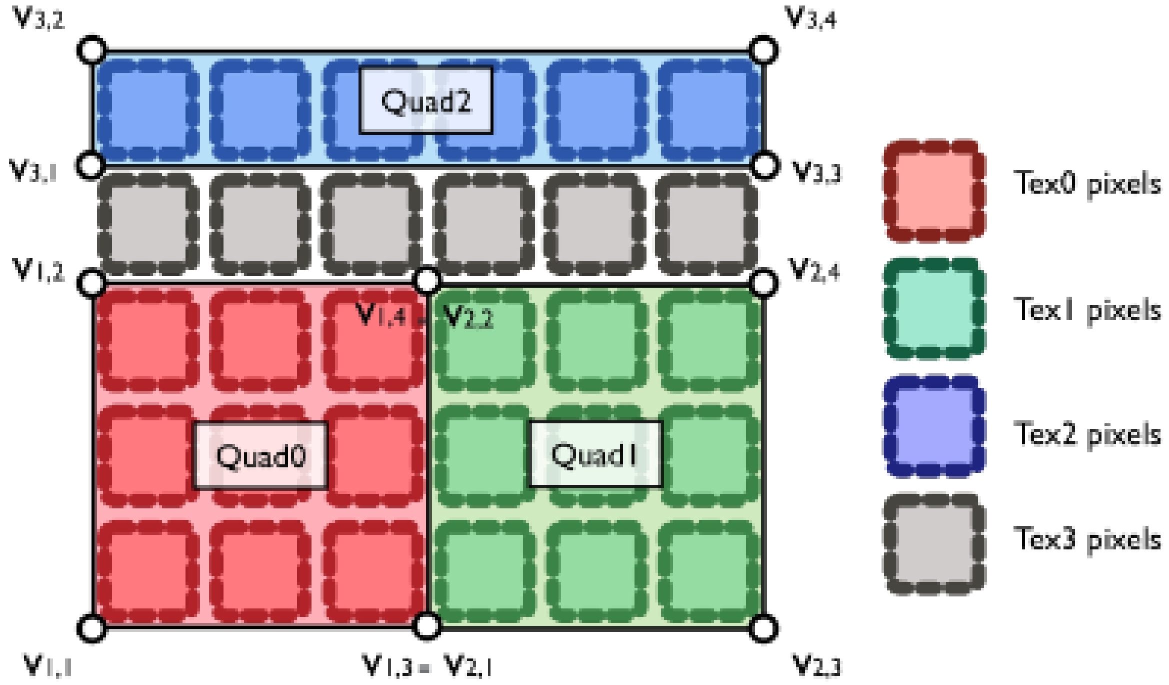


Figure 5.5: OpenGL full texture layout for audio physical model representation. (Zappi et al., 2017)

the relevant values collected into floats and *vec4* types, the calculation can be made and $n+1$ written back to the texture by *return*ing a correctly formed graphics fragment in *vec4*.

5.2 Results

In this chapter, the *High-end NVIDIA GeForce* from Table 3.3 is used to gather performance results. The operating system installed is windows 10 version 20H2[2], the OpenCL version used is from the CUDA SDK version used is 11.1.0 and all OpenGL programs use version 4.6[3]. When profiling the various versions of the physical model, the number of grid points N_x and N_y will be scaled at a doubling rate equal in both resolutions starting from 8 up to 512; therefore, $(N_x, N_y) = \{8, 16, 32, 64, 128, 256, 512\}$. Each version of the physical model will have the average execution time recorded for a buffer length of 256 over 172 buffers. The buffer length of 256 was selected as the results from Section 3.2.7 concluded that this buffer length displayed a good balance between overall performance and the

[2] https://learn.microsoft.com/en-us/windows/whats-new/whats-new-windows-10-version-20h2

[3] https://developer.nvidia.com/opengl-driver

```
vec4 frag_color  = texture(inOutTexture, tex_c);    //Loads pressure from texture.
vec4 p       = frag_color.rrrr;                    //Current centre point.
float p_prev = frag_color.g;                        //Previous centre point.

// Contains neighbour pressure and boundaries.
vec4 p_neigh;
vec4 b_neigh;

// Loads left neighbour from texture.
vec4 frag_l = texture(inOutTexture, tex_l);
p_neigh.r   = frag_l.r;
b_neigh.r   = frag_l.b;

//For all neighbours.
...

//Parallel computation of neighbours//
vec4 pLRUD = p_neigh*(1-b_neigh) + p*(b_neigh)*frag_color.b;

// Explicit update scheme calculation.
float p_next = 2*frag_color.r + (dampFactor-1) * p_prev;
p_next += (pLRUD.x+pLRUD.y+pLRUD.z+pLRUD.w - 4*frag_color.r) * propFactor;
p_next /= dampFactor+1;

// Return simulation data as graphics fragment.
return vec4(p_next,  p.r, frag_color.b, frag_color.a);
```

Figure 5.6: OpenGL fragment shader for finite-difference simulation adapted from Zappi et al. (2017)

selected real-time requirements. Further, the buffer length 256 is more widely supported by audio devices than the smaller buffer lengths tested (Liang et al., 2021). The chosen buffer length of 256 at 44.1KHz has a maximum audio buffer period of 5.805ms to satisfy the audio-sound latency requirement as seen in Table 2.1. The results in Table 5.1 record the average buffer execution time across 172 buffers across the enumeration of physical model resolutions. An additional column for each version measures the relative speedup of each measurement with relation to the naive serial CPU version. This highlights the differences between the default serial approach developers might use compared to the parallel versions (Bocchino et al., 2009). The first thing to notice is that the AVX CPU version that employs parallel vector processors on the CPU clearly improves performance over the serial version, with a consistent speedup between 2.4× and 3.4×. Therefore, basic finite-difference based physical modelling synthesis benefits from the SIMD parallel processing, even on the CPU.

Figure 5.7 plots the execution times recorded in Table 5.1. The results highlight a clear difference between the CPU and GPU versions as the resolutions scale from $(N_x, N_y) = 8$ up to $(N_x, N_y) = 1024$. For lower resolutions, between $(N_x, N_y) = 8$ to around $(N_x, N_y) = 32$, both of the CPU versions process the buffers considerably faster than the GPU counterparts. At $(N_x, N_y) = 8$, the CPU Serial version is at least 7× faster than the OpenCL Global and approximately 3× faster than OpenGL. This behaviour is expected, as the initial communication overhead of using the discrete GPU exceeds the execution time of the entire computation at these resolutions. This advantage will only favour the CPU further when buffer lengths less than 256 are used, as the GPU communication overhead increases further. However, as the resolutions scale, the ability for the GPU to fully utilise all parallel processors becomes clear and as shown in Figure 5.7, the GPU versions surpass the CPU serial at $(N_x, N_y) = 32$ and both CPU versions by $(N_x, N_y) = 64$ (indicated with a vertical dotted line). For $(N_x, N_y) > 32$, the GPU versions continue to handle the higher resolutions better than the CPU versions. By $(N_x, N_y) = 128$, even the parallel AVX CPU versions can not sustain the audio-sound latency maximum audio buffer period of 5.805ms, whilst the GPU versions can support this up to at least $(N_x, N_y) = 512$. To put the difference into perspective, $(N_x, N_y) = 128$ requires computing 16384 finite-difference points, while $(N_x, N_y) = 512$ requires 262144, 16× more computation. By $(N_x, N_y) = 1024$, OpenCL achieves a speedup of 125× and OpenGL 73× over the serial CPU version.

The difference between the OpenCL global and local versions is hard to detect on the graph in Figure 5.7, but the slight difference can be seen by analysing Table 5.1.

	Serial CPU		**AVX CPU**		**OpenCL Global**		**OpenCL Local**		**OpenGL**	
Resolution	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
8x8	0.262	1.0	0.099	2.6	1.900	0.1	1.947	0.1	0.872	0.3
16x16	0.448	1.0	0.186	2.4	1.884	0.2	1.936	0.2	1.000	0.4
32x32	1.604	1.0	0.471	3.4	1.884	0.9	1.924	0.8	0.988	1.6
64x64	5.883	1.0	2.081	2.8	1.953	3.0	1.988	3.0	1.122	5.2
128x128	20.436	1.0	8.384	2.4	1.895	10.8	2.186	9.3	1.215	16.4
256x256	78.791	1.0	33.209	2.4	2.215	35.6	3.075	25.6	1.732	45.5
512x512	317.732	1.0	134.389	2.4	3.151	100.8	3.232	98.3	4.994	62.2
1024x1024	1324.116	1.0	539.459	2.5	10.535	125.7	10.790	122.7	17.924	73.9

Table 5.1: Mean buffer compute time calculated over 172 buffers of length 256. System = High-end NVIDIA Desktop

The OpenCL version that utilises local memory appears to be slightly slower than the global version. It suggests that the overhead of moving data from the global memory to local memory is greater than the improvement local memory accesses provide. This is likely because, in this basic physical model, grid points are not accessed often in the computation. Moving the grid to local memory may be beneficial for more sophisticated physical models that require accessing grid values numerous times in the calculation. However, this hypothesis would require further experimentation to confirm. Therefore, for the remainder of this thesis the designs will default to using direct global grid memory accesses.

Considering the GPU versions, the execution time of OpenCL is consistent between $(N_x, N_y) = 8$ to $(N_x, N_y) = 128$, suggesting the majority of the time is made up of the GPU communication overhead, and the GPU processing is not fully utilised until larger amounts of data are processed past $(N_x, N_y) > 128$. In contrast, OpenGL involves similar GPU transfer overhead, but executes in about half the time of the openCL versions from $(N_x, N_y) = 8$ to $(N_x, N_y) = 128$. However, where the GPU begins to become useful past $(N_x, N_y) > 128$, OpenCL surpasses the performance of OpenGL.

5.3 Evaluation

Many programmers are most familiar with programming CPUs within a procedural language with a serial SISD execution pattern. Therefore, the serial CPU version is often the intuitive starting place for programming a finite-difference based physical model. Nevertheless, a clear and consistent speedup between 2.4× and 3.4× is observed when processing this linear finite-difference physical model using the outdated AVX SIMD instruction set. This suggests that simple linear models can be processed in parallel without any clear

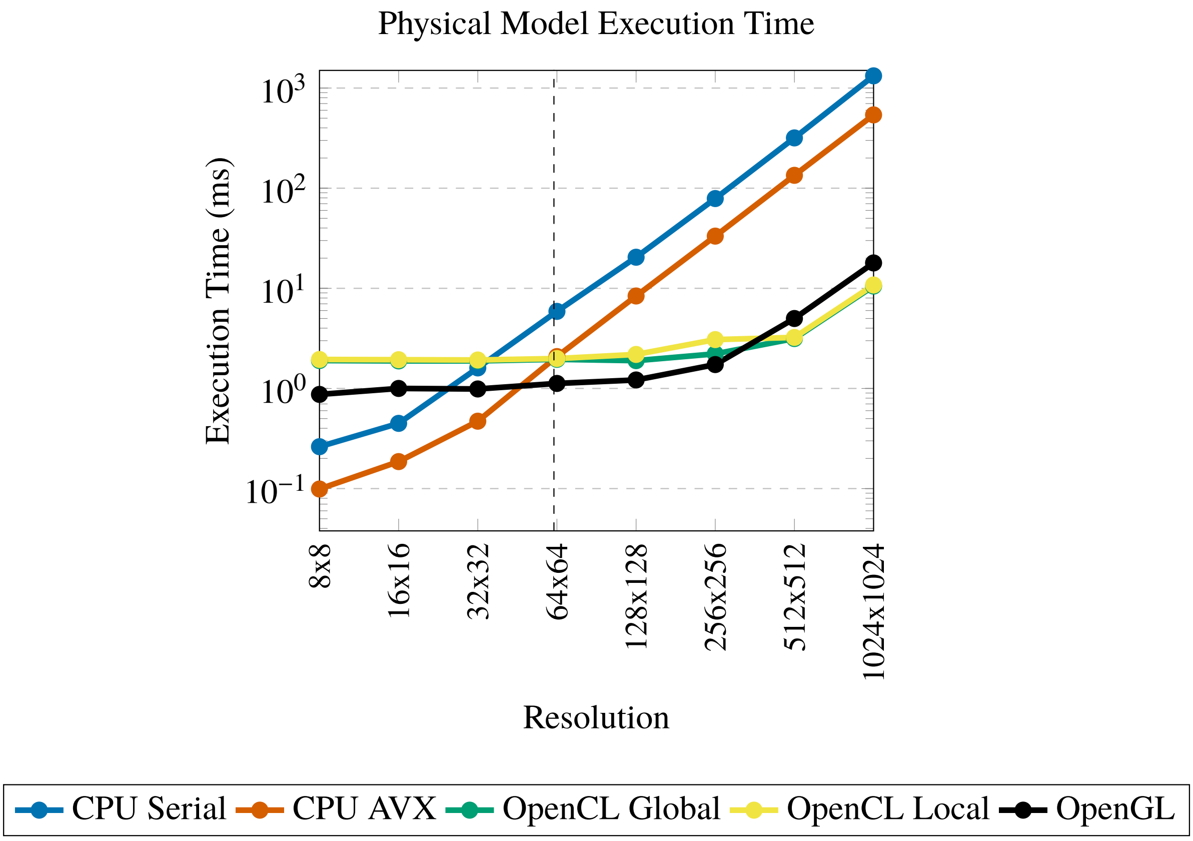


Figure 5.7: Average execution time for all physical model versions when scaling the model resolution. System = High-end NVIDIA Desktop

disadvantage whilst improving performance considerably. However, this does not necessarily extend to sophisticated physical models such as ones that must solve non-linear systems, as the limitations of parallel processing may affect these.

The results highlight the context when a linear finite-difference based physical model is more suited for the CPU or GPU. It appears, when scaling the resolution uniformly, the CPU is more suitable for handling resolutions $(N_x, N_y) < 64$, and then the GPU begins to outperform the CPU when $(N_x, N_y) > 64$. For this basic physical model, the CPU can support resolutions $(N_x, N_y) < 64$ at a sample rate of 44100Hz. For the same sample rate, the GPU can support a considerably larger range of resolutions up to $(N_x, N_y) < 512$. This suggests that the GPU can support higher resolution numerical physical models than the CPU can. Increasing the resolution of the simulated space has several advantages, including improved accuracy, more stable simulations and the space to create more sophisticated instruments. However, increasing the resolution proportionally increases the computation required to run the simulation. Designing a framework for GPU accelerated physical models will make the development of larger physical models more accessible.

Two GPU interfacing methods were compared to implement physical model synthesis on the GPU. A GPGPU approach using OpenCL and a graphics domain approach in OpenGL. The OpenCL approach is far more accessible as it provides a general compute framework that is designed to be quickly adopted, whilst the OpenGL approach requires a fundamental understanding of graphics rendering. This has been extensively demonstrated in the designs by Zappi et al. (2017) where the primary drawback of the OpenGL graphics pipeline is the sophisticated methods for adding interactions to models that requires using global shader variables called *uniforms* for updating parameters and input and output locations. In contrast, the general computation environment provided by OpenCL is more appropriate for the DMI developers, whilst the OpenGL design from Section 5.1.4 involves a more specialist approach that has difficulties supporting more advanced physical models. Furthermore, as covered in the results of Section 5.2, OpenCL was observed to perform better when scaling to larger grids. For these reasons, moving forward into Part 2 of this thesis, OpenCL will be used as the primary implementation method of the proposed designs.

The last key takeaway from this chapter is that using GPU local memory is not a guaranteed optimisation and appears to depend on the complexity of the physical model. Therefore, designs moving forward will default to using global memory and local memory optimisations considered as a possible future extension.

Part 1 of this thesis has demonstrated that the GPU is a powerful processing device for supporting processes that require high data throughput, such as parameter matching and physical modelling synthesis. With the correct configuration, the GPU can operate within real-time requirements whilst improving the range of resolutions supported for physical modelling synthesis. Part 2 of this thesis uses the fundamental knowledge established in this chapter and applies it to the design of a framework for facilitating the development of linear finite-difference based physical models for real-time audio synthesis.

Part II

Manually programming the various components of the physical modelling techniques covered in Part 1 takes a considerable amount of time and effort to successfully implement, as described in Chapter 5. To give some rough perspective, the isolated code for a minimal OpenCL physical model (without any audio callback or interaction code) created for this thesis involved approximately 1163 lines of code, and this only grows larger as the functionality and features of the physical model expands. Furthermore, a constant difficulty is that the more advanced components added to the physical model design, the more computation and resources are required, making it increasingly difficult to achieve real-time performance. Therefore, Part 2 of this thesis presents HyperModels, a framework for describing high-resolution, linear physical models that utilise the GPU hardware acceleration for real-time synthesis. This framework aims to improve the accessibility of real-time physical modelling to developers for interaction and performance. This could lead to unforeseen musical expression that artists can explore beyond the technical achievements that non-linear physical models can achieve offline (Cook, 1993) (Zappi, 2017). HyperModels aims to provide this environment by exposing a high-level interface for describing physics equations and an instrument's shape, e.g. the strings or membrane, that are automatically translated into optimised low-level code that utilises the real-time capabilities of modern GPUs. This approach enables the instrument designer to focus on the sound design aspect of a new instrument without necessarily requiring the advanced low-level architecture and programming knowledge often required to access parallel GPU programming. Building an instrument is demonstrated in Section 9.1, where the HyperModels framework is used to build a physically modelled drumhead. The instrument application from Figure 9.1a is built using the SVG description from Figure 9.1b and the finite-difference scheme for the two-dimensional wave equation. Part 2 begins by explaining all the GPU specific design components used for optimising the GPU design. This leads to the comprehensive definition of HyperModels and all the design details. A quantitative evaluation of the HyperModels framework is then presented using performance profiling and comparing the auto-generated HyperModel programs to manually written equivalents. Finally, the use of HyperModels to build actual instruments is demonstrated with two case study instruments that provide video demonstrations and source code.

Chapter 6

GPU Design Components

This chapter covers the GPU specific design components necessary for building the proposed physical modelling environment. This chapter draws from the foundational understanding gathered from Part 1 of this thesis to form the optimal design components here.

6.1 Memory Structure

The ideal memory arrangement on the GPU is to use a single flat-array memory buffer and offset into it to locate specific data. For two-dimensional physical modelling synthesis, a series of two-dimensional grids are used to store the state of the modelled system through space and time. Figure 6.1 provides a theoretical view of the state of the two-dimensional grid through time. Here, it can be seen that the value at each finite-difference grid point reflects the chosen function u. *Grid 0* contains the whole state of the system for all points $x = [0, N_x]$ to $y = [0, N_y]$ for timestep $t = 0$. These grids are progressively calculated using the explicit finite-difference scheme, using the initial values from *Grid 0* to calculate *Grid 1* and so on, to the latest timestep at $t = N_t$. Depending on the finite-difference equation formed, only a certain number of the grids need to be retained in memory for calculating the next timestep. For example, the two-dimensional wave scheme in Equation (2.13) requires $n-1$, n and $n+1$; this abstract two-dimensional view of the system can be dispatched to the GPU using OpenCL's abstract model. However, ultimately the GPU must suitably prepare memory as a one-dimensional flattened array representing the two-dimensional form. Therefore, space for three grids are needed in memory and must be mapped and accessed as a GPU suitable data structure. To do this, the GPU allocates $N_x * N_y$ of contiguous memory for the number of grids needed. For Equation (2.13), three grids of contiguous floating point memory must be allocated. Figure 6.2 visualises how the grid is represented as a one-dimensional flattened array and accessed using an index offset for $u^{n-1}_{x,y}$, $u^{n}_{x,y}$ and $u^{n+1}_{x,y}$ as the grid rotates.

6.1.1 Rotation Index

Explicit finite difference schemes that are solved recursively must keep track of the state of the system for a series of appropriate timesteps. Therefore, at the end of each iteration, the state of the system's timesteps must be updated. This can be achieved with the following mathematical form:

$$u^{n-1} := u^{n} \quad \text{and} \quad u^{n} := u^{n+1} \tag{6.1}$$

Considering u as the entire N-dimensional state of the system at time n, for each iteration the state considered for u^{n-1} becomes u^n and u^n becomes u^{n+1} for the next timestep. Programmatically, this might be done by using another temporary grid region of memory to copy the grids between them to the effect of timestepping. However, this needlessly copies data between memory when instead a pointer switching or rotation index method can be used. Pointer switching works by creating address pointers to the regions of memory containing the states of the system. Then, instead of copying the contents of the memory between the states to simulate timestepping, the pointers that indicate the memory regions are switched to point to the next state's region of memory. Referring back to 6.2, it can be seen that the timesteps $n-1$, n and $n+1$ can be switched around to point to the next region of memory when the whole state of $u^{n+1}_{x,y}$ has been calculated. To the same effect, an index into the 1-dimensional flattened array can be used called r.

The rotation index r is used to offset into the correct grid. The rotation index is multiplied by the grid size to offset the relevant grid for the current timestep. Therefore, the centre index into the flat array can be calculated as $I_o = r*N_x*N_y$, where I_o is the centre grid index offset, r is the rotation index pointing at the timestep being considered (like n), N_x and N_y are still the number grid points in the x-axis and y-axis respectively. Using I_o to offset into the correct grid, the indices for the grid points required for calculating $u^{n}_{x,y}$ must be collected. The GPGPU kernels provide methods for dispatching programs to the GPU that are considered in a two-dimensional workspace. Therefore, the x and y positions of each grid point being calculated inside the GPU program can be requested.

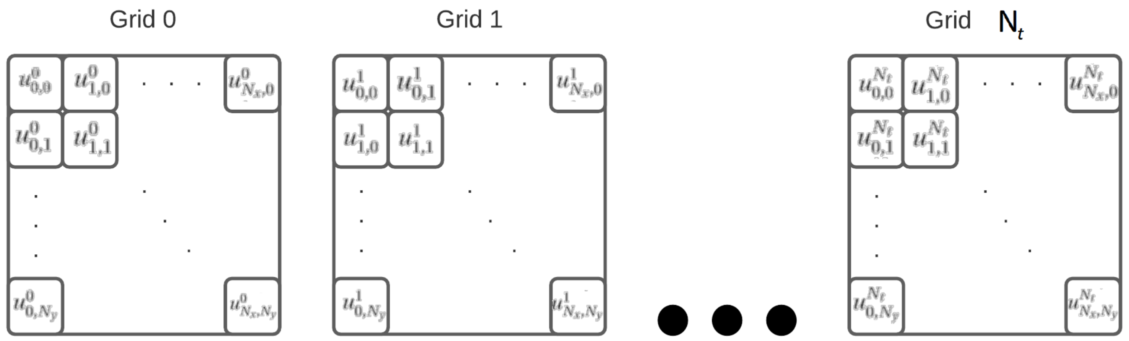


Figure 6.1: Theoretical view of the physical model state.

These can then be mapped into the 1-dimensional flat buffer using $I_c = y * N_x + x$, where I_c is the index of the centre point, x is the index on the x-axis and y for the y axis. Combining the grid offset and the grid point index provides $I_m = I_o + I_c$. Within the GPU program, the indices for accessing these points can be programmed using Algorithm 2.

Algorithm 2 Calculate Indices

```
▷ Calculate grid offset of current timestep.
int offsetRotation = gridSize * idxRotate;
int centreIdx = offsetRotation + (getWorkitemID(1)
        * getWorkitemSize(0) + getWorkitemID(0));
int leftIdx = offsetRotation + (getWorkitemID(1)
        * getWorkitemSize(0) + getWorkitemID(0) - 1);
▷ Definitions for all other neighbours right, up and down...
```

Here, *offsetRotation* implements $r*N_x*N_y$ as *gridSize* represents N_x*N_y and *idxRotate* is r. The functions *getWorkitemID* and *getWorkitemSize* are then used to calculate the indices such as the *centreIdx* $I_c = y * N_x + x$. When the GPU program is dispatched with the number of workitems equal to the number of grid points for the simulation, the indices for correctly accessing the flat buffer in Figure 6.2 is maintained.

The *boundary grid* is used to represent the boundary grid. The boundary grid identifies all points in the model space that are to be considered boundaries and then modify the standard state finite-difference function u to be replaced with the boundary condition function $u = \Psi$. The centre point offset I_c is used to access the boundary grid and identify if function u should be replaced with the boundary function $u = \Psi$.

The GPU memory allocations are positioned in different types of physical GPU memory. In order to maximise the access speed by the relevant device, the memory allocations

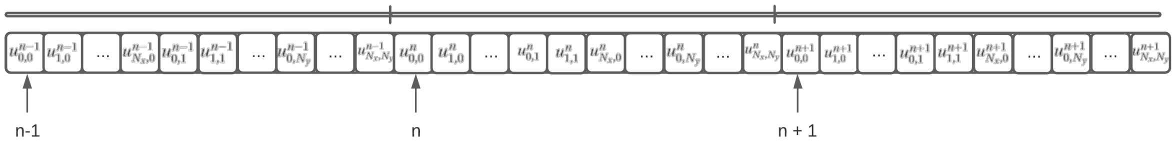


Figure 6.2: Physical GPU representation of physical model states shown in Figure 6.1.

Memory Type	Memory Names	OpenCL Flags
GPU Local Memory	System State Grids, ID grid, Boundary Grid	CL_MEM_HOST_NO_ACCESS & CL_MEM_READ_WRITE
CPU Mapped Memory	Input Buffer, Output Buffer	CL_MEM_HOST_WRITE_ONLY & CL_MEM_READ_ONLY & CL_MEM_HOST_READ_ONLY & CL_MEM_WRITE_ONLY
Push Constants	Rotation Index, Buffer Index, Physical Model Parameters	N/A

Table 6.1: Memory Allocation Types

should be in the appropriate locations. The system state, id grid and boundary grid are initialised once by the CPU and are therefore appropriately allocated to GPU local memory. The input sample buffer is written to by the CPU and read from the GPU. In contrast, the output buffer is written to by the GPU and read back to the CPU. These buffers should be optimised for the appropriate read and write accesses on the GPU and also need to be optimised for CPU accesses. The final memory types are not buffers but single data types that contain rotation, buffer indices and physical model parameter values. These are relatively small memory allocations that are updated from the CPU regularly, making them push constants and should therefore be optimised as such. The summary of all memory allocations and their types is given in Table 6.1 with a column using OpenCL specific memory allocations flags to demonstrate how these are implemented.

6.2 If-Conversion

Although the GPU architecture supports conditional branching, it has been documented as often being suboptimal to add branching (Rotem and Ben Asher, 2014). This is because of the SIMT structure of the GPU; groups of streaming processors execute in lock-step, meaning that all streaming processors on a compute unit execute the same set of instructions simultaneously. Therefore, when a program branches and separate sets of functions must be executed, the streaming processors that execute one branch must wait on the completion of the other branch. Under certain conditions, the GPU compiler can use optimisations such as predictive branching to avoid negative effects on performance (He and Zhang, 2010). For example, if the branch condition operates on a constant buffer the compiler can reliably predict the branch path. However, if the data is dynamic and either branch can execute, the compiler has to execute both sides and pick the best result or use proper branching. Modern GPU developments have seen some improvements supporting branching beyond predictive branching even with dynamic data and can be seen for example with fragment branching (Harris, 2005). Despite this, a conservative approach will be taken in this thesis that does not depend on these modern improvements that may not be supported by all GPUs. Therefore, control dependencies using dynamic data based on branching will be converted to data dependencies using arithmetic (Elkhouly et al., 2016).

As covered so far, linear finite-difference based physical modelling synthesis is highly suited for the GPU as each grid point can be calculated independent of one another, without synchronisation. However, alterations to the calculations need to be made when boundary points are identified. A common approach would be to use a conditional statement to check if a neighbouring cell is a boundary, calculate it using the boundary condition equation. This would lead to an inferior program based on branching on the GPU. Instead, branching can be avoided altogether using if-conversion. The pseudocode in Algorithm 4 illustrates this difference for a clamped Dirichlet boundary condition. The first function *boundaryConditional* uses a control dependency. It takes three arguments, *gridP* is the grid of the system state values, *gridB* is the grid identifying boundary points, and *idxB* is the index of the boundary point being considered. Using an if statement, *gridB* can be accessed at *idxB* to check if it is a boundary. If it is, then the *boundaryPressure* is set to the clamped value hardcoded here as 0. Otherwise, it is a normal grid point and set to the grid point value in *gridP*. Alternatively, the use of any control dependencies like an if statement can be avoided by if-conversion to use only data dependencies demonstrated in *boundaryArithmetic*. Here, the grid collected from *gridP* using *idxB* is multiplied by *1-gridB[idxB]*. The effect of this is the same as *boundaryConditional*, when *gridB[idxB]* is 1, it is a boundary point and therefore clamped to 0; otherwise, it does not affect the grid point value.

The proposed GPU optimised design will use if-conversion when generating the GPU programs. Although supporting if-conversions for more advanced boundary conditions, this will be the approach used in the current design that supports Dirichlet and Neumann boundary conditions.

Algorithm 3 Boundary Approaches Compared

```
function BOUNDARYCONDITIONAL(gridP, gridB, idxB)
    if gridB[idxB] then
        boundaryPressure = 0
    else
        boundaryPressure = gridP[idxB]
    return boundaryPressure

function BOUNDARYARITHMETIC(gridP, gridB, idxB)
    boundaryPressure = gridP[idxB] $*$ (1-gridB[idxB])
    return boundaryPressure
```

6.3 Buffering Input and Output

The buffering technique for transferring audio samples to and from the GPU will be a core configuration parameter of the HyperModels framework. The buffer length will be a controllable parameter of the synthesis design in order to be directly supported by audio playback devices and interfaces that externally determine audio buffer length. Input or excitation samples can then be written from the CPU into the GPU accessible memory and put into the physical model at the input position point. Samples generated from the physical model are then written to the output buffer on the GPU and read by the CPU for further processing or directly transferred to the audio playback device. The location where the input buffer is written and the output buffer is read from is controlled by two global variables. The following pseudocode achieves the input-output behaviour:

Algorithm 4 Boundary Approaches Compared

```
if idxCentre == outputPosition then
    outputBuffer[idxSample]= u[idxCentre]
if idxCentre == inputPosition then
    uPlusOne[idxCentre] += inputBuffer[idxSample];
```

Here, if the $idxCentre$ being the current point being considered is equal to the $output$-$Position$, the sample on the grid state u is written to the output buffer at the $idxSample$, the position of the buffer which is incremented every timestep. If the $idxCentre$ equal to $inputPosition$, then the sample is read from the input buffer and added to the model state for the next timestep $uPlusOne$.

6.4 Halo Cells

Following the memory structure and processing format laid out in Section 6.1, an issue arises when applying calculations to the edge of the memory grid. The finite-difference equations involve accessing neighbouring values in the memory grid; this involves calculating the indices of the neighbouring points and then using them to access the GPU memory buffer. However, when at the edge of the grid, the generated indices will attempt to access memory outside of the GPU memory buffer, causing undefined behaviour (Li et al., 2019). A sub-optimal solution is shown in Algorithm 5, where grid points on edge are identified using if-statements and ignored.

Algorithm 5 Sub-optimal Edge Detection

```
1: if workitemID[0] == 0 || workitemID[1] == 0 || workitemID[0] == workitemSize[0] ||
   workitemID[1] == workitemSize[1] then
2:     return
```

Here, a two-dimensional system where *workitemID* is an array containing each dimensions ID for the current work item and *workitemSize* is the number of workitems in each dimension. If there are workitems equal to every grid point in the two-dimensional system, this will ignore one cell around the entire grid and avoid accessing neighbours outside the GPU memory space. However, involving control dependencies on the GPU can reduce performance. An alternative solution is to use a halo of grid points around the simulation grid space (Kjolstad and Snir, 2010) (Millán et al., 2015). Figure 6.3 illustrates how a grid can be arranged to operate on the inner green cells, and an outer halo of cells are designated as part of the GPU memory buffer but are not involved in the simulation calculations. Using this method, the number of workitems dispatched for execution on the GPU equals the number of internal cells highlighted in green. The indices for accessing cells must then be offset in order to operate on the internal green cells, which can be programmed as shown in Algorithm 6.

```
int centreIdx = ((getWorkitemID(1) + haloOffset) * getWorkitemSize(0) +
                    getWorkitemID(0)+ haloOffset);
int leftIdx = ((getWorkitemID(1)) + haloOffset) * getWorkitemSize(0) +
                    getWorkitemID(0) - 1 + haloOffset);
//Definitions for all other neighbours...
```

Here, *haloOffset* is the size of the halo grid in grey surrounding the simulated space in green. By adding this onto the x and y axes, the workitems IDs can be used to access only the inner simulated grid points. This avoids the undefined behaviour that would be caused by accessing unallocated memory outside of the grid.

Algorithm 6 Optimised Halo Grid Edge Detection

```
▷ Calculate grid offset of current timestep using halo offset.
int offsetRotation = gridSize * idxRotate;
int centreIdx = (getWorkitemID(1) + haloOffset)
                * getWorkitemSize(0) + getWorkitemID(0) + haloOffset;
int leftIdx = (getWorkitemID(1) + haloOffset)
                * getWorkitemSize(0) + getWorkitemID(0) + haloOffset - 1;
▷ Definitions for all other neighbours right, up and down...
```

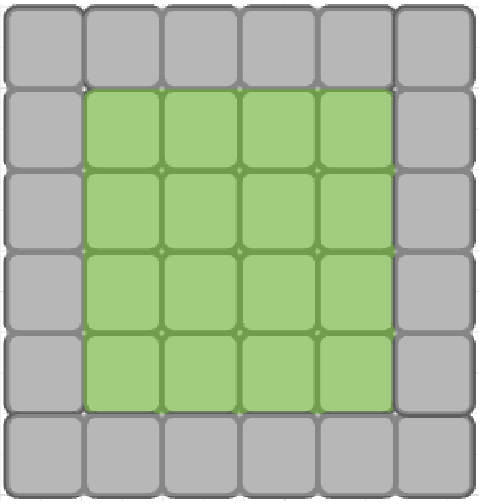


Figure 6.3: Two-dimensional grid space where the cells highlighted in green are part of the simulation space and grey cells are halo cells not operated on.

6.5 Constant Folding

Constant folding is the process of identifying and optimising constant expressions at compile time rather than repeatedly calculating them at runtime (Muchnick et al., 1997, p. 329). Terms in constant expressions are typically simple literals or constant variables in a calculation that can be reduced to a single number. For example, $i = 320 * 200 * 32$ can be folded down to simply $i = 2048000$. When forming explicit schemes with finite-difference equations, there are often parts to the equation that will contain redundant components if implemented directly in a program calculation. Consider Equation (2.13), there is a single controllable parameter λ. This is mathematically interesting because λ represents the wave speed propagation through the simulated material. However, if the GPU program includes the whole equation in the calculation, it redundantly processes λ^2 at every point. Instead, the calculation of $c_\lambda = \lambda^2$ can be done once on the CPU when the parameters are set, and then the coefficient c_λ can be loaded onto the GPU and replace λ^2. Including this optimisation requires the GPU interface to expose the original meaningful parameters like λ and then do the constant folding as an intermediate step to form c_λ, load it onto the GPU and include it in the calculation.

6.6 Summary

In this Chapter, a number of key GPU design techniques and optimisations have been covered. To summarise, the following will play a key role in the design of the GPU accelerated physical modelling design being presented:

- A grid that maps the model geometry to the GPU cores.
- Boundary grid for describing model's boundary points.
- System state grids for containing the state of the spatial systems for relevant timesteps.
- GPU optimised flattened memory Buffers.
- Rotation indices for efficient timestepping.
- Appropriate GPU Memory Types.
- Buffering CPU to GPU data transfers.

- Array containing instrument interconnection points.
- Constant folding to optimised runtime calculcations.
- Multiple input & output positions.

These techniques are used in the following chapter to present the design of the GPU optimised HyperModels framework.

Chapter 7

HyperModels Framework

There are many examples of frameworks that use DSLs for facilitating the development of applications that depend on PDEs. However, these DSLs are developed to target specific domains outside of digital audio processing. *Saiph* and *Liszt* are primarily designed for using defined PDEs in a DSL for solving fluid dynamics and many other similar DSLs exist for countless other domains (Macià et al., 2018) (DeVito et al., 2011). These DSLs have been demonstrated as being effective (Kieburtz et al., 1996), but they are limited to their targeted domain, meaning they are not easily adapted for creating digital audio synthesisers. There is one well-established digital audio processing DSL called Faust that supports the development of finite-difference based physical modelling synthesis (Russo et al., 2021). However, despite all these features in Faust, its original intention was not to specifically target the development of finite-difference based physical model synthesisers; This means Faust is missing more advanced features such as GPU acceleration (which finite-difference models are well known to benefit from (Moler, 1986)) and a visual approach for describing the geometry of the models. Instead, Faust is currently CPU bound and the physical model geometry must be defined within the written DSL.

Considering that there appears to be no DSL framework specifically designed for developing physical models for audio synthesis, this thesis aims to present a novel design explicitly addressing this. The HyperModels framework will have two key features that are missing from any existing DSL in this field, complete GPU acceleration of the finite-difference based physical model programs and a visual method for describing the physical model geometry. With these additional features that improve performance and accessibility, HyperModels may be in an ideal position to facilitate the development of new DMIs.

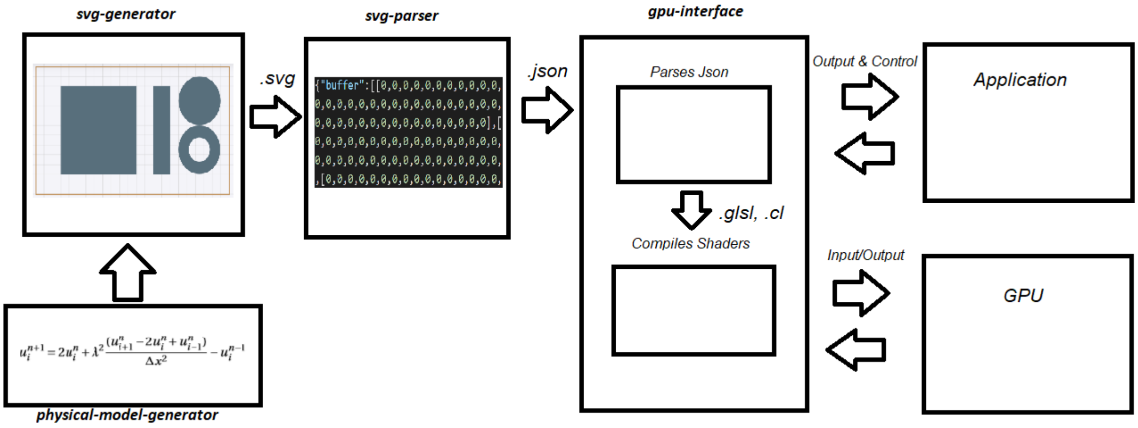
This chapter presents the HyperModels framework, a framework for automating the mapping of finite-difference based physical modelling synthesis into a GPU optimised parallel format for real-time processing. Initially, the framework aims to support modelling linear systems of recursively solvable explicit finite-difference schemes within a two-dimensional environment. The summary of the HyperModels framework has been accepted for publication in Paper [D] at the conference of New Instruments for Musical Expression (NIME) 2022 under the title "HyperModels - A Framework for GPU Accelerated Physical Modelling Sound Synthesis" (Renney, Willemsen, Gaster and Mitchell, 2022).

7.1 HyperModels Overview

The entire HyperModels framework is made up of the components arranged in Figure 7.1. This framework uses a component-based software design approach (Heineman and Councill, 2001). Each component functions independently of one another, such that provided the appropriate input and output, various implementations of each component can be developed and connected. This is a common and successful design approach as supported in the literature (Mahmood et al., 2005). There are four distinct software components: *physical-model-generator*, *svg-generator*, *svg-parser* and then the *gpu-interface*. The *physical-model-generator* generates the GPU program by mapping finite-difference equations defined in a DSL into a parallel format suitable for generating the appropriate GPU program. The *svg-generator* is used for creating the geometry of the physical model system inside an SVG. The SVG format (covered in Appendix B.1) must then be parsed into a bitmap style description of the same model for the finite-difference GPU program. This is where the *svg-parser* converts this into the suitable form captured inside of the JSON format (for JSON see Appendix B.2). With the GPU program and geometry contained inside the JSON object, the *gpu-interface* is used for loading the GPU program and geometry onto the GPU and interfacing with it. The interface can then be used to integrate the physical model into an application. Each of the components will first be covered in a platform-agnostic form before describing an implementation used in the subsequent chapter to validate the design.

7.2 Physical Model Generator

The *physical-model-generator* maps the definition of explicit finite-difference schemes into a parallel form appropriate for the GPU to process. The abstract design of the mapping is



```
{"buffer":[[0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

Figure 7.1: Relationship diagram of the GPU physical modelling tools.

described from which the details of the HyperModels DSL will be defined. The front-end of the DSL parses valid equations to form an AST that is used to generate the equivalent GPU program for a target GPU standard (such as OpenCL). An implementation of this design will be demonstrated using the Python programming language at the end of this section.

7.2.1 Model to GPU Mapping

Mapping finite-difference equations into a parallel processing environment requires defining an appropriate representation and program structure. Here, we define the GPU as system a with a grid of $C_x \times C_y$ PEs[1]. A single processing element is denoted by a_{c_x,c_y} where $c_x \in \{1,\ldots,C_x\}$ and $c_y \in \{1,\ldots,C_y\}$ are the PE indices in the horizontal and vertical direction of the GPU grid respectively. The update equation of one grid point can then be assigned to a single PE, such that these can be executed in parallel. Considering the 2D system presented in Equation (2.13), mapping a grid point of a scheme with spatial index (l,m) to a PE with index (c_x,c_y) is denoted by $a_{c_x,c_y} \Leftarrow v_{l,m}$.

[1] Refer back to Section 2.3.3 for the OpenCL terminology used for abstractly discussing GPU processes and cores.

The PEs of the GPU a must then be processed by recursively traversing the two-dimensional system a and calculating the values of a^{n+1} at each position depending on what models are assigned at each position. This process is repeated recursively and once all of a^{n+1} is calculated, time index n can be incremented. A serial representation of this process is formed using two nested *for* loops as shown in Algorithm 7.

Algorithm 7 Serial Representation

```
n = 0
while isSimulation do
    for x = 1 to gridX do
        for y = 1 to gridY do
            if id[x][y] == v then
                a[n+1][x][y]  = 2 * a[n][x][y] - a[n-1][x][y]
                              + $\lambda^2$*(a[n][x+1][y] + a[n][x-1][y] + a[n][x][y+1]
                              + a[n][x][y-1] - 4 * a[n][x][y])
    n = n + 1
```

Here, every position in the grid is visited by iterating over all possible values for x and y. These are first used to check if the position in the two-dimensional grid is inside the model v. If it is, then Equation (2.13) is used to calculate the value at $a^{n+1}_{c_x,c_y}$. In the equation, components such as $a^{n}_{c_x+1,c_y}$ require accessing neighbouring values of the currently considered position at c_x and c_y by adding 1 to x, leading to a[n][x+1][cy]. This is done for all neighbouring values. This serial approach can be rewritten as Algorithm 8, so that it is suitable for parallel processing.

Algorithm 8 Parallel Representation

```
n = 0
while isSimulation do
    cx = getGridX()
    cy = getGridY()
    if id[cx][cy] == v then
        a[n+1][cx][cy]= 2 * a[n][cx][cy] - a[n-1][cx][cy]
                      + $\lambda^2$*(a[n][cx+1][cy] + a[n][cx-1][cy] + a[n][cx][cy+1]
                      + a[n][cx][cy-1] - 4 * a[n][cx][cy])
    n = n + 1
```

Here the nested *for* loops are replaced by the implicit dispatching of gridX by $gridY$ workitems. The loop's indices are represented through the identifiers getGridX() and getGridY(), respectively, where $C_x \times C_y$ process streams are dispatched to the processor. Thus, instead of iterating over two nested for loops, the ID of each process stream is used to check which model's equation to use, such as v for accessing a to calculate the next timestep a^{n+1}. The state of the system a is contained in global GPU memory as it is only directly accessed once by each PE.

This mapping supports multiple equations inside of system a depending on the position of c_x and c_y coordinates. For example, as shown in Algorithm 9, one can add the 1D wave equation described in Equation (2.19) as a second model to the GPU. This one-dimensional equation can be mapped into the system a according to $a_{c_x,c_y} \Leftarrow u_l$. The code then includes an additional conditional statement checking if the coordinate (c_x, c_y) identifies u or v. This mapping is used by the HyperModels DSL to map equations defined in the DSL into the GPU parallel program code.

Algorithm 9 Parallel Representation Two Equations

```
n = 0
while isSimulation do
    cx = getGridX()
    cy = getGridY()
    if id[cx][cy] == v then
        a[n+1][cx][cy]= 2 * a[n][cx][cy] - a[n-1][cx][cy]
                        + $\lambda^2$*(a[n][cx+1][cy] + a[n][cx-1][cy] + a[n][cx][cy+1]
                        + a[n][cx][cy-1] - 4 * a[n][cx][cy])
    if id[cx][cy] == u then
        a[n+1][cx][cy]= 2 * a[n][cx][cy] - a[n-1][cx][cy]
                        +$\lambda^2$* (a[n][cx+1][cy] - 2 * a[n][cx][cy]+ a[n][cx-1][cy] )
    n = n + 1
```

7.2.2 Front-End - DSL Definition

The grammar of the HyperModels DSL will be formally defined using BNF (Section 2.4.2). First, the basic fundamental numerical data types including whole and decimal numbers must be defined:

```
int                ::=  digit { digit }*
float              ::=  pointfloat | exponentfloat
```

```
pointfloat      ::=  [intpart] fraction | intpart "."
exponentfloat   ::=  (intpart | pointfloat) exponent
fraction        ::=  "." digit { digit }*
exponent        ::=  ("e" | "E") ["+" | "-"] digit { digit }*
```

The *int* type is used for matching all whole numbers $\mathbb{N}$ as a sequence of one or more *digit*s. The *float* type is used for matching real numbers $\mathbb{R}$ and can take one of two forms, it can be either *pointfloat*, á la *1.2*, or *exponentfloat* that supports scientific notation such as *5.9736e24*. Next, all the letters supported for naming variables must be defined:

```
letter          ::= [a-z A-Z]
```

This set for *letter*s includes all capital and lower case letters from the english alphabet. Using the *letter* type, variables for coefficients and constants in the variable can be defined:

```
coeff           ::= letter { letter | digit }*
const           ::= "const " coeff
```

Here, a *coeff* is at least one letter, followed by one or more *letter*s or *digit*s, like *lambda1*. A *const* is the literal sequence "const ", followed by a *coeff*. The last expression for representing values is for expressing finite-difference points u in the unique form:

```
u ::= $u^{int}_{int,*}$
```

u is used to indicate a finite-difference point from the simulation grid. The *int* in the superscript of u is a single *int* for indicating the timestep relative to the current finite-difference point. The subscript is one or more comma separated *int*s indicating the spatial step from the current finite difference point. This pattern enables referencing a point on the finite-difference grid through space and time. With these types defined, operators for forming calculations with them are defined as:

```
unop            ::= + | -
binop           ::= ^ | * | / | + | -
```

unop is used to reference unary operators, these will be operators that take a single argument preceding it, like *-0.5* or *-lambda*. *binop* is for binary operators, these are operators that take two arguments for a calculation, including 9+10, 5^4 and $2*lambda$. All of these value types and operators are collated together to form the recursive expression:

```
expr            ::= u | num | float | coeff | const | ( expr ) | unop expr
                    | expr binop expr
```

The equation expression *expr* ranges over the possible valid set of expressions defined in the grammar. *expr* recursively matches *binop* operators with further expressions either side which can be terminating expressions, like u, or further binary operators to continue the recursive behaviour, building up the finite-difference equation.

7.2.3 Back-End - GPU Program Generation

The Haskell programming language will be used to present a generalised mapping from the DSL equations to the GPU parallel representation. Although the Meta-Language (ML) (Pierce, 2002) can be used to formally present this design with proofs, Haskell is used as it provides a closer representation of the final implementation (Jones, 1999).

The HyperModels BNF grammar formalised in the previous section provides the design of the front-end parser for converting the finite-difference equations defined in the DSL into a form ready for the GPU program generation. The Haskell programming language can match the *expr* with the following code:

```haskell
data Op = Plus | Mult | Div | Pow
    deriving (Show)

data Expr = EU Int [Int] | EInt Int | EVar String | BinaryOp Op Expr Expr
    deriving (Show, Eq)
```

This captures the HyperModels BNF grammar in Haskell as *Expr*. Any sequences separated by a " | " can be matched to identify a valid expression that can be parsed to build an AST that can be used to generate GPU code. A significant programmatic difference from the BNF grammar is that the super and subscript notation used for defining relative finite-difference points used in the explicit schemes $u^{int}_{int,*}$ is matched instead with the following *EU Int [Int]*. Here, an integer for the timestep is followed by a list of integers for relative spatial steps in the finite-difference scheme. This format has been chosen over more sophisticated types to conform to the universal ASCII standard that all text editors support. The *Op* type is used for matching binary operators that are optimised using constant folding in the *eval* function that is defined as:

```haskell
eval                            :: Expr -> Int
eval (EInt i)                   = i
eval (EVar var)                 = var
eval (UnaryOp Pos x)            = eval +x
eval (UnaryOp Neg x)            = eval -x
eval (BinaryOp Plus x y)        = eval x + eval y
eval (BinaryOp Minus x y)       = eval x - eval y
eval (BinaryOp Mult x y)        = eval x * eval y
eval (BinaryOp Div x y)         = eval x * eval y
eval (BinaryOp Pow x 0)         = 1
eval (BinaryOp Pow x y)         = eval x * Pow x (y-1)
```

The *eval* function can be recursively called on the AST formed from Expr to fold all constants and remove redundant calculations. Once *eval* has no more effect, the recursive constant folding halts.

The AST representation of the equation is formed by matching *Expr*. Once the AST has been optimised using *eval*, the AST can be used by the HyperModels compiler to generate GPU program code. The compiler will target a high-level, human-readable GPU programming standard; available target GPU standards include GLSL shaders, OpenCL and CUDA kernels. The GPU programs will be generated using template code for the target standard and the code injection technique (Section 2.4.4). A key part of generating code involves searching for data types contained in the AST. The following Haskell code is used for searching for a particular data type:

```haskell
findX :: [Expr -> Bool] -> Expr -> [Expr]
findX fs (BinaryOp _ e e') = findX fs e ++ findX fs e'
findX fs e = concatMap (\f -> if f e then [e] else []) fs
```

Here, the function *findX* is used for searching the AST for a particular type and collecting them into a list. This is important for collecting all the relevant information about the equations stored in the AST to inject the generated code snippets into the template code. For example, using *findX(AST, EU)* will collect all finite-difference expressions into a list. The list can then be filtered to find all neighbouring values that need to be accessed to generate code for accessing all relevant indices in the GPU grid memory. This code can then be injected into the template, making the relevant indices available for accessing memory throughout the GPU program. Using this approach, a set of functions for generating all the necessary code snippets must be defined. These functions are implementation-specific and depend on the GPU standard being targeted. Therefore, the details of these functions will be covered in the implementation section and here will just be presented as declarations:

```haskell
generateHeader      :: AST -> String
generateHeader()
generateIndices     :: AST -> String
generateIndices()
generateVariables   :: AST -> String
generateVariables()
generateBoundary    :: AST -> String
generateBoundary()
generateEquation    :: AST -> String
generateEquation()
```

The String outputs of these functions can then be injected into the template code using regular expressions (Karttunen et al., 1996). The following code provides an example for the GPU code header:

```haskell
import Text.Regex (mkRegex, subRegex)

```

```
-- Generate code for header and insert into template.
headerSnippet = generateheader(AST)
strProgram = subRegex (mkRegex "$insertHeader$") template headerSnippet
-- ... For all code snippets.
```

Here, the code snippet for the header is generated from the AST using *generateheader*. This is then injected into the *template* code in place of *$insertHeader$*. This process is repeated for all sections of the template that have code injected into them in place of the tokens.

7.2.4 Implementation Python

This section presents an implementation of the HyperModels DSL using the Python programming language. This implementation will support OpenCL as the target GPU standard, although the implementation can be extended to support other targets by adding additional template code and code generation functions for each target.

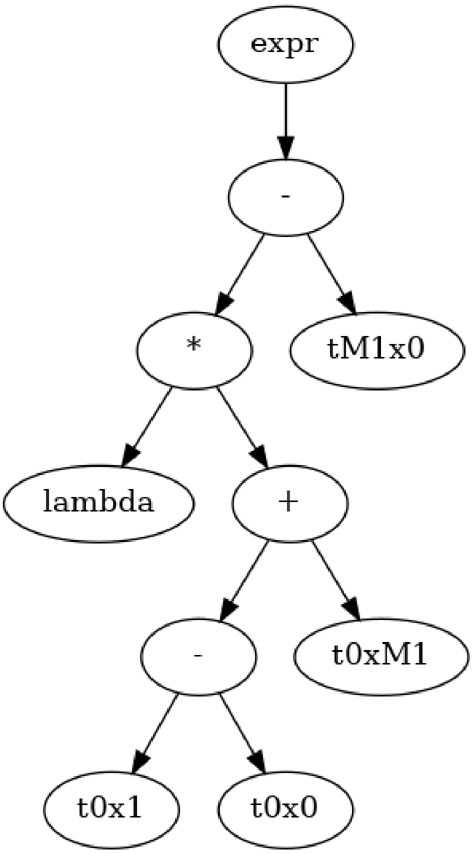


Figure 7.2: Visualisation of an abstract syntax tree of Equation (2.19)

PyParsing[2] is used for implementing the HyperModels DSL grammar from Section 7.2.2. The parsing functionality is contained in the independent front-end python file *PhysicalModelParser.py*. Leaving the GPU code generation in a separate back-end python file, *PhysicalModelGenerator.py*. The *defineGrammar()* function below is used to pythonicaly define and prepare a grammar object that can be used to parse the DSL:

```python
def defineGrammar():
    # Numbers - 0.75, 5, 1000.0
    number = pyparsing_common.fnumber
    number.setParseAction(Constant)

    # Coefficients - mu, lambda
    coefficient = Word(alphas)
    coefficient.setParseAction(Variable)

    # Timesteps - u(0)(0,0), u(-1)(0,0), u(0)(2,-1)
    u = Group(Literal("u") + Suppress('(') + number + Suppress(')') +
            Suppress('(') + delimitedList(number) + Suppress(')'))
    u.setParseAction(Timestep)

    # Arithmetic - 5 * 5, mu * mu, lambda * (u(0)(0,0) - u(-1)(0,0))
    signop = oneOf("+ -")
    multop = oneOf("* /")
    plusop = oneOf("+ -")
    expop = Literal("^")
    arithmetic = infixNotation(
        u | coefficient | number,
        [
            (signop, 1, opAssoc.RIGHT, SignOp),
            (expop, 2, opAssoc.LEFT, BinOp),
            (multop, 2, opAssoc.LEFT, BinOp),
            (plusop, 2, opAssoc.LEFT, BinOp),
        ],
    )
    arithmetic.setParseAction(PhysicalModelAST)

    # Collect all rules in grammar.
    grammar = Forward()
    grammar << arithmetic
    grammar.ignore('#' + restOfLine)

    return grammar
```

Example	Description
u(0)(1,0)	Value of point in current timestep and x+1, y in spatial dimensions
u(-1)(0,0)	Value of point in previous timestep and centre position x, y
u(0)(-1,1)	Value of point in current timestep and x-1, y+1 spatial dimensions
u(0)(-1,1,-1)	Value of point in current timestep and x-1, y+1, z-1 spatial dimensions

Table 7.1: Example of representing discrete steps in a physical model using the DSL.

[2] https://github.com/pyparsing/pyparsing

This implements the HyperModels grammar, where finite-difference points represented in the BNF as u, take the form given in examples shown in Table 7.1. The grammar returned from the function *defineGrammar* can then be used to parse a string of input using the PyParsing function *parseStr(input, grammar)*. A string object containing the finite-difference equation can be given to the function, along with the defined grammar object to parse the equation and produce the appropriate abstract syntax tree. This generates an AST as output that can be used for generating the GPU program code. The following example is given to describe the process of forming the AST using the python parser on a valid definition of the one-dimensional wave equations explicit scheme (2.19):

```
grammar     = defineGrammar():
strEquation = "u(1)(0) = 2 * u(0)(0) + lambda^2 * ( u(0)(1) - u(0)(0)"
                "+ u(0)(-1) ) - u(-1)(0))"
AST         = parseStr(strEquation, grammar)
```

This will create an AST as visualised in Figure 7.2 (removing multiplications with 2 for conciseness), where the equation is structured in a way that prepares it for convenient and optimized traversal during the back-end code generation stage. For example, all leaf nodes can be searched to provide a list of all finite-difference points and coefficients required for the program. Further, operations here such as "*" and "-" are arranged in the tree to naturally give correct precedence to mathematical operations in the calculation. This is demonstrated when starting at the lowest level of the abstract syntax tree, "u0x0" should be taken away from "u0x1" first, before adding it to "u0xM1" which is a level higher up the tree.

The GPU programs are then generated by searching the AST for information, generating appropriate code snippets and injecting them into the target GPU standard template. In the python implementation, all data types are given a definition for __str__ that returns a string representing the datatype. An implementation for a generic class that can handle one-dimensional models as in the example given above but also supports two-dimensional definitions is provided below:

```
class U():
    ...
    def __str__(self):
        ret = "t"
        t = self.__timestep.get()
        ret += (("M" + str(abs(t))) if t < 0 else str(t))
        x = self.__x.get()
        y = self.__y.get()
        ret += "x"
        ret += (("M" + str(abs(x))) if x < 0 else str(x))
        ret += "y"
        ret += (("M" + str(abs(y))) if y < 0 else str(y))
        return ret
```

So when the AST is searched for all unique finite-difference points in the equations, they can be used to generate a list of the required indices by getting the string representation of all unique points and appending the index code onto it. The example of this implementation in Python targeting OpenCL is:

```
def generateIndicesCL(AST):
    strIndices = ""
    # Form code snippet template for generating OpenCL indices.
    strIndex = "int $indexName$ = $insertRotationIndex$
                                  + ((get_global_id(1)+$xAxis$)
                                  * get_global_size(0) + get_global_id(0)
                                  +$yAxis$);"

    # Search AST
    for eq in AST:
        #Get unique timesteps.
        timesteps = eq.value.find(U)    # The Class U represents timesteps.
        for t in timesteps:
            # Build index code snippet.
            strTempIndex = re.sub("\$indexName\$", t.getIndex() + "Idx", strIndex)
            strTempIndex = re.sub("\$insertRotationIndex\$", t.getRotationIndex(),
                                    strTempIndex)
            strTempIndex = re.sub("\$xAxis\$", str(t.getX()), strTempIndex)
            strTempIndex = re.sub("\$yAxis\$", str(t.getY()), strTempIndex)

            # Append index code snippet to indices.
            strIndices += strTempIndex
    return strIndices
```

This function uses a code snippet template *strIndex* and goes through all unique timesteps, generates a unique and relevant index and then appends it to the list of indices. The entire code snippet returned from this function includes all the indices needed in the GPU program for processing the finite-difference equation. Continuing to use the one-dimensional wave scheme as an example, the AST generated from the scheme in Figure 7.2 given to the function *generateIndicesCL* would return:

```
int t0x0y0Idx = rotation0   + ((get_global_id(1)+0) * get_global_size(0)
                            + get_global_id(0));
int t0x1y0Idx = rotation0   + ((get_global_id(1)+1) * get_global_size(0)
                            + get_global_id(0));
int t0xM1y0Idx = rotation0  + ((get_global_id(1)-1) * get_global_size(0)
                            + get_global_id(0));
int tM1x0y0Idx = rotationM1 + ((get_global_id(1)+0) * get_global_size(0)
                            + get_global_id(0));
```

This code snippet would then be inserted into the OpenCL GPU code template in place of the *$insertIndices$* token. For this implementation, the following OpenCL code template is used:

```
int rem(int x, int y)
{
    return (x % y + y) % y;
}

__kernel
void $insertHeader$ //Header informtion.
{
        //Rotation Index into model grid.
        int gridSize = get_global_size(0) * get_global_size(1);
        $insertRotationIndices$

        //Get relevant indices.
        $insertIndices$
        int t1x0y0Idx = rotation1 +
                        (get_global_id(1)) * get_global_size(0) +
                        get_global_id(0);

        float t1x0y0;
        $insertBC$

        //Calculate the next pressure value.
        t1x0y0 = $insertEq$;

        //Read sample at outputPosition.
        if(centreIdx == outputPosition)
        {
                output[samplesIndex]= t0x0y0;
        }

        //Input excitation at inputPosition.
        if(centreIdx == inputPosition)
        {
                t1x0y0 += input[samplesIndex];
        }

        modelGrid[t1x0y0Idx] = t1x0y0;
}
```

The token *\$insertIndices\$* would then be replaced with the result of *generateIndicesCL*. This process is repeated for replacing all tokens in the template with generated code snippets. The final GPU code is built using the generated code snippets inserted into the template code; the target GPU standard can then compile this to create the HyperModels physical modelling program[3].

7.3 Vector Based Representation

The HyperModels framework proposes using vector graphics based representation for describing the shape of models in the simulation. The shape and dimensions of a physical model have a considerable influence of the generated acoustics. This is demonstrated in

[3] The full implementation of the standalone Python parser and code generator can be found in the open-source repository at: `https://github.com/Harri-Renney/HyperModels-physical-model-generator`

studies modelling speech using physical models of glottal passage, where the shape of each glottal section determines the output sound (Scherer et al., 2010; Cummings et al., 1995). HyperModels uses an annotated form of the SVG protocol for describing models using vector graphics. The standard SVG format is extended to include additional information needed for mapping the shapes into the physical model GPU memory a. Best explained with an example, two rectangles and a circle inside a 12x12 environment as visualised in the SVG in Figure 7.3 would be described with the following HyperModels annotated SVG:

```
<svg viewbox='0 0 12 12' height='12' width='12' connections='4 3 10 10'
    physics_program='...' interface_device='custom'>
    <rect id='1' interface_osc_address='' interface_type='pad'
        interface_osc_args=' 'width='3' height='10' x='3' y='2'
        style='fill:rgb(88,111,124);'/>
    <circle id='2' interface_osc_address='' interface_type='pad'
        interface_osc_args=' 'r='2' cx='8' cy='2'
        />
    <rect id='3' interface_osc_address='' interface_type='pad'
        interface_osc_args=' 'width='4' height='4' x='9' y='8'
        style='fill:rgb(88,111,124);'/>
</svg>
```

Inside the SVG *svg* tags, the *viewbox*, *width* and *height* are all important components as they describe the resolution of the entire physical modelled environment. Defining these as *viewbox='0 0 12 12' height='12' width='12'*, describes the environment as a 12x12 grid environment. This controls the resolution of the entire simulated environment a, dictating the possible space to draw detailed shapes inside. The *connections* field contains a list of tuples containing coordinates that are connected together between models. The list of coordinates are iterated through in the GPU program to apply connection forces between the two points. In the example above, the position x=4, y=3 on rect ID 1 will have a connection to position x=10, y=10 on the rect with ID 3. The *physics_ program* contains the generated GPU program from the *physical-model-representation* stage. The SVG format enables convenient changes to these values to tune the resolution of the application. A final attribute used in the *svg* tags is the *interface_ device*. This attribute is used for conveniently setting a default interface resolution size. The example above uses a custom interface such that a 12x12 resolution grid can be defined. However, the convenience of this tag can be explained in the case where a Sensel Morph touch pad is used for interacting with the model, the model dimensions will be conveniently set to use the Sensel Morphs resolution of 185x105.

The support of the standard SVG shape tags is maintained so that the SVGs can be rendered as coloured shapes, along with additional attributes for describing the physical

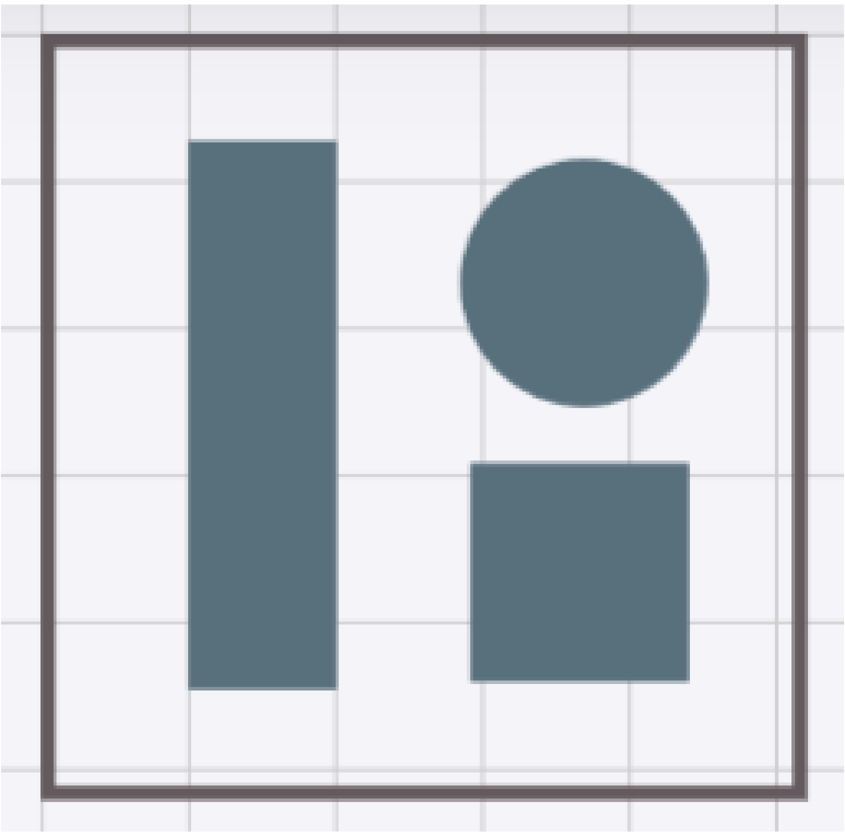


Figure 7.3: Rendering of the SVG description above.

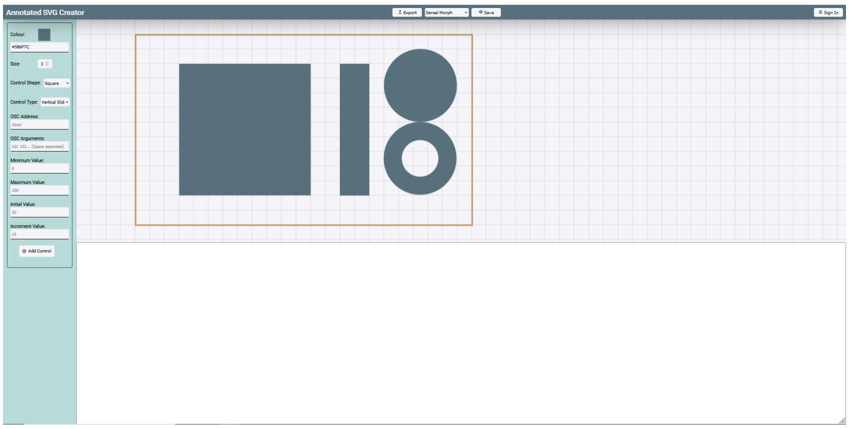
model in each shape. The most important being *id* and *physics_ program*. The *id* is the unique identifier for each shape; this is used to fill in the *id* of the shape when generating the two-dimensional grid in the *svg-parser*. Another attribute is *interface_ osc_ args*, which contains a list of any optional arguments that can be used to set constants in the *gpu-interface API* stage. The intended use of these is open-ended, but they enable the physical model developer to include useful information about the shape in the final developed application. For example, the ranges of acceptable parameter values can be defined in here (like $\lambda \leq \frac{1}{\sqrt{2}}$ for the two-dimensional wave equation) which an application developer can reference using *gpu-interface* to ensure they are creating an unstable simulation.

7.3.1 Implementation Javascript Web Application

The *svg-generator* design has been implemented as a javascript web application[4]; a screenshot is given in Figure 7.4. The Javascript web application supports drawing shapes on

[4] Code available from here: `https://github.com/Harri-Renney/HyperModels-svg-generator`

a two-dimensional canvas. From the placement of shapes, the application can generate and store it as SVG data in a file that populates the HyperModels attributes. The physical model equations are entered into a text box and sent to a local Flask[5] server containing the Python implementation of the *physical-model-generator*. This generates the target GPU program, sends it back to the svg-generator and is added to the SVG *physics_program* attribute. The tool also supports drawing connections between shapes that populates the *connections* attribute with tuples of connected coordinates.



Sensel Morph

Colour: #586F7C

Size: 3

Control Shape: Square

Control Type: Vertical Slid

OSC Address: /stop/

OSC Arguments: 101 102... (Space separated)

Minimum Value: 0

Maximum Value: 100

Initial Value: 50

Increment Value: 10

Figure 7.4: The modelling tool demonstrating various shapes inside an enclosed envrionment.

7.4 svg-parser

The *svg-parser* is used for parsing the extended SVG format covered in the *svg-generator*. Simulating the physical models using finite-difference based methods requires a Cartisian grid of points. Therefore, the SVG vector image format must be mapped to a two-dimensional array using rendering techniques. The design presented here is based on the SVG parser work by Gaster et al. (2019), with modifications making it appropriate for the physical modelling framework[6]. The process of parsing the SVG follows a similar method of rendering graphics (Figure 2.5). The SVG description must first be tessellated; this is the process of producing a set of vertices from triangles that, when considered holistically, forms the original shape defined in the SVG. The set of vertices then undergoes rasterisation, where a grid of points is generated containing the ID of the current shape inside the vertices.

[5] https://flask.palletsprojects.com/en/2.1.x/

[6] Source-code of the svg-parser implementation available at: `https://github.com/Harri-Renney/HyperModels-svg-parser`

Referring back to the 12x12 example in Figure 7.3, a 3x10 rectangle, 4x4 square and a circle with a radius of 2 are described. These are placed inside the environment that is a 12x12 grid as described in the first SVG tag. The SVG parser will first need to generate a grid reflecting the size of the environment, in this case, a 12x12 grid. Then, each shape will need to be processed to determine the cells to set to the shape's ID. The attributes belonging to each shape will also need to be stored with the shape ID, including the physical model GPU program. Finally, the shapes must be tessellated and rasterised to map the shapes into a grid space. The tessellation of each shape requires interpreting the shape type and the attributes included in it that describe the shape. This requires slightly different approaches depending on the shape parsed, but all draw from well-established tessellation techniques (Sellers et al., 2013).

Processing the rectangle SVG tags requires basic tessellation. Figure 7.5 presents the basic tessellation and rasterisation pipeline using the rectangle as an example. The rectangle position x and y are used to locate the first vertex, using the *width* and *height*, the other 5 vertices are generated to form two triangles. Each triangle can then undergo basic rasterisation (Hersch, 1989) to fill the triangle with values, in this case, the shape ID. Each triangle's vertices are taken and used to form a bounding box; this bounding box can be used to set the value of all grid points falling inside it to the current shape's ID.

Forming a circle can be done using a tessellation technique known as a triangle fan (Shreiner et al., 2009, Chapter 11). First, the SVG circle attributes *cx* and *cy* are used to find the centre point of the circle, then using the radius contained in r, 4 equally spaced vertices located on the top, bottom, left and right of the circle. Then, using the same process, 4 new vertices evenly spaced along the circle circumference between these 4 original vertices can be generated. This is repeated up to the required resolution of the circle that can be calculated based on the circumference length. Figure 7.6 demonstrates how increasing the number of triangles for the circle creates a smoother circumference. This repeated doubling of the vertices along the circumference makes this a process suitable for recursion.

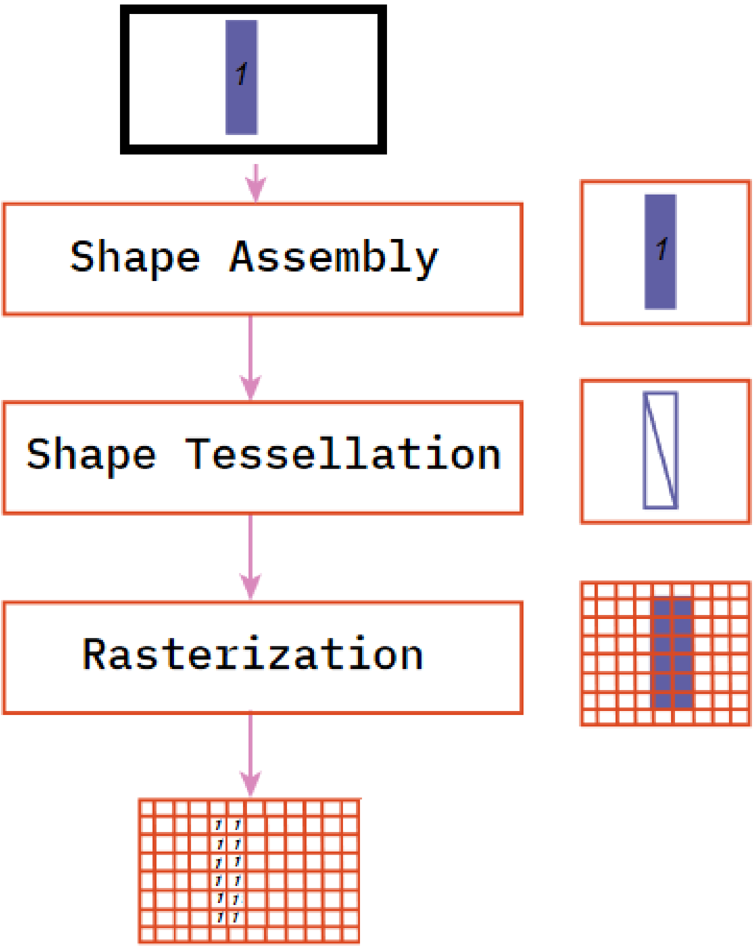


Figure 7.5: Interface tessellation and rasterization of single rectangle.

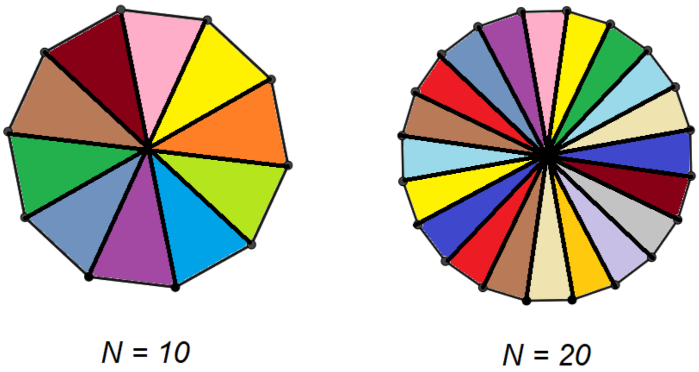


Figure 7.6: Effect of increasing vertices for a circle to improve smoothness of circle circumference.

7.4.1 JSON Representation

With these shapes, the SVG structure can be used to generate the model environment and shapes inside of it. The parser can then be used to capture this information inside of a JSON object for universal accessibility. Using the SVG example from Section 7.3, the following JSON representation can be formed:

```
{
    "shapes" : [
    {
        "id": 1,
        "rgb": "rgb(217,137,188)",
        "args": [
        0.39
        ]
    },
    . . .
    ],
    "environment": [
        [
            H, H, H, H, H, H, H, H, H, H, H, H, H, H,
            H, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 0, 2, 2, 0, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 2, 2, 2, 2, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 2, 2, 2, 2, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 0, 2, 2, 0, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 3, 3, 3, 3, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 3, 3, 3, 3, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 3, 3, 3, 3, 0, H,
            H, 0, 0, 1, 1, 1, 0, 0, 3, 3, 3, 3, 0, H,
            H, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, H,
            H, H, H, H, H, H, H, H, H, H, H, H, H, H,
        ]
    ] ,
    "physics_program": "...",
    "connections": [4, 3, 10, 10],
    "interface" : "custom"
}
```

Here, there are a list of *shapes* that contain an *id*, this maps the rest of the attributes inside the shape with the grid points inside the *environment* buffer. Inside the *environment* buffer the grid of *id* cells are surrounded by the boundary halo cells indicated by the symbol H[7]. Each shape also contains any constant coefficients, RGB colour and the physical model GPU program in *physics_program*. The connections take the same form as the SVG, listing numbers representing the connected coordinates.

It is in this stage that the halo cells are generated to surround the grid such that shapes can be supported close to the edge of the model and the calculations will not attempt to access memory outside of the grid in undefined memory. The thickness of the

[7] The symbol H is used here for illustration purposes. In the implementation H is replaced with a reserved number for representing halo cells, in this case it is the maximum 8-bit value 255.

halo cell boundary is equal to the size of the largest finite-difference equation's stencil. The stencil of a finite-difference equation is the distance of neighbouring cells that are accessed by the equation to calculate the next timestep.

All the necessary physical model equations and geometry and defined inside this JSON schema. A C++ framework presented in the next section can now be used to unpack and load the physical model programs and data structures representing the grids onto the GPU.

7.5 gpu-interface API

The *gpu-interface API* component needs to support the following interfacing functionality:

- Compilation of the GPU program.
- Loading the GPU program onto the GPU.
- Requesting the executing the GPU program.
- Sending excitation as input to the GPU.
- Receiving generated samples as output from the GPU.
- Updating physical model parameters.

To meet these requirements, the necessary GPU standard and API needs to be supported. The *gpu-interface API* is designed as an abstract class so that various implementations can be made supporting different targets, such as OpenCL, Vulkan, CUDA, etc. Provided the virtual functions defined here are implemented, the details of the interfacing API are dependant on the target GPU standard.

7.5.1 Implementation C++ GPU Interfacing API

In this implementation, the *gpu-interface-api* component has been written in C++ using an abstract class *GPU_Accelerated_PM* which currently supports the OpenCL standard in the *GPU_Accelerated_PM_OpenCL* class. The support of further standards can be introduced by inheriting the abstract class *GPU_Accelerated_PM*. The function declarations that meet these capabilities are shown here:

```
void createModel(const std::string aPath);

void step();
void fillBuffer(float* input, float* output, int numSteps);
void updateCoefficient(std::string aCoeff, int aIndex, float aValue);

void setInputPosition(int numInputs, int aInputs[]);
void setOutputPosition(int numOutputs, int aOutputs[]);
```

createModel() takes a file path to the JSON file on the system. This JSON file must conform to the form described in the SVG-Parser in Section 7.4. The GPU program for the target GPU interfacing API can then be loaded onto the GPU. *step()* is a private function that is used for advancing the physical model one timestep. This is used by *fillBuffer* to recursively advance the physical model and at each time step extract sampels from the output position to fill the *output* buffer. updateCoefficient() is used to change the value contained in a named coefficient, defined originally in the DSL. The input and output positions for excitation and sample generation respectively can be moved using *setInputPosition()* & *setoutputPosition()*. Both of these functions take an argument for the number of inputs or outputs and then an array containing the coordinates for each of the inputs or outputs. When these functions are used, they alter the state of the physical model to determine where input and output audio signals are directed when processing the model using *fillBuffer*. When the GPU program is executed, the input signal is added to all the input positions set by *setInputPosition* and samples are recorded onto the output audio buffer from the physical model for all the output positions set by *setoutputPosition*.

Chapter 8

Evaluation

This chapter evaluates the performance of the GPU programs generated by the HyperModels framework using the same benchmarking strategy from Section 3. The *High-end NVIDIA Desktop* from Table 3.3 is used with windows 10 version 21H2[1] and the OpenCL version used is from the CUDA SDK version used is 11.4.2.

Using this system, the performance results between the automatically generated physical model GPU programs will be compared to functionally identical manually developed physical models [2]. The purpose of this comparison is to highlight any performance differences from automatically generated programs to manually written ones. To create a controlled benchmarking environment for the purpose of this thesis, only the GPU program of the physical model will be altered between the auto-generated and manually developed programs. The C++ framework (Section 7.5) and FDTD grid representations (Section 7.2.1) will remain consistent between versions. The performance of the autogenerated and manual GPU programs will also be compared to parallel CPU versions.

8.1 Comparative Tests

The benchmarking tests follow the same real-time profiling technique from Section 3.1. A sample rate is set, and all the physical models require bidirectional memory transfers. Each test is executed and profiled through an enumeration of sample buffer lengths and physical model dimensions. The buffer length again begins at 1 and increases in powers

[1] https://learn.microsoft.com/en-us/windows/whats-new/whats-new-windows-10-version-21h2

[2] Benchmarking suite available to build from source-code at: https://github.com/Harri-Renney/HyperModels-benchmarking-suite

of 2 up to 1024. The physical model dimensions begin with 64x64 and increases uniformly for both axes in powers of 2 up to 1024. Four tests have been defined that have been designed first to test the basic functionality of the physical models and reveal fundamental differences in performance between the automatic and manually written GPU physical model programs. The tests that involve more complex equations and multiple models aim to expose further limitations of the automated process over the manual approach. The four tests are defined as follows:

- *Simple Single Model* - A single two-dimensional wave equation square physical model. (Geometry shown in Figure 8.1)
- *Simple Multiple Models* - Ten 1-dimensional wave equations on separate strings. (Geometry shown in Figure 8.3)
- *Complex Single Model* - A single two-dimensional circle model that uses a complex linear plate equation including general and frequency dependant damping. (Geometry shown in Figure 8.5)
- *Complex Multiple Models* - Two two-dimensional square physical models connected by a single string. (Geometry shown in Figure 8.7)

The manually written GPU programs provide opportunities for a competent developer to exploit specific types of optimisations and take advantage of contextual elements that automated tools either have difficulty identifying or impossible to accurately or safely implement. There are a couple of common optimisations used in the manually written shaders that are currently absent from the autogenerated versions. The first is constant folding and the second is grouping all similar equations into a single identifiable ID. Both of these optimisations are supported in the HyperModels design, and can be added to the implementation of the *physical-model-generator* with further work.

The theoretical estimated FLOPs for each test will be calculated and compared in the results to the actual execution time to demonstrate the effectiveness expected from optimisations. In this chapter, the estimated FLOPs is extended slightly from Equation 3.1 to:

$$FLOPS = N_x * N_y * r_s * N_o * N_u \tag{8.1}$$

Where a new variable N_u is define as the utilisation space of the simulation which is the percentage of the simulation space occupied by calculations. N_x, N_y, r_s and N_o

continue to be the simulation resolution, the sample rate and the estimate number of operations executed per utilised grid point.

8.1.1 Simple Single Model

The simple single model test is designed to profile the performance for a single basic physical model equation, simple clamped boundary condition and square geometry. The finite-difference equation used is based on the simple two-dimensional wave equation with the general damping component, used previously and defined in Equation (5.3)[3]. The Dirichlet boundary condition evaluates points at the edge of the shape and is defined as:

$$u^i_{b_x,b_y} = 0 \quad \forall \quad b_x \& b_y, \tag{8.2}$$

Where b indicates a boundary grid that indicates x and y coordinates of boundary points with b_x and b_y respectively. Therefore, whenever an identified boundary point is identified at the border of the square physical model, the value at $u^i_{b_x,b_y}$ will be clamped to 0. This physical equation will operate within a simple rectangle taking up the majority of the simulated environment space as shown in Figure 8.1. This test involves most of the simulated space being involved in calculations at approximately 88% utilisation.

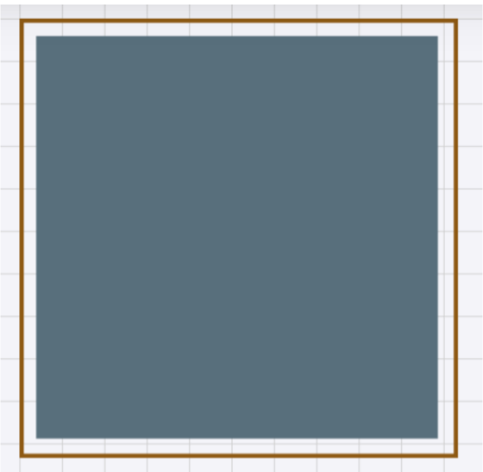


Figure 8.1: SVG representation of *Simple Single Model* geometry.

This simple test does not involve many opportunities to optimise the manual version over the auto-generated one. The most significant difference in the manual version is the

[3] See Figure A.1 in the appendix for the HyperModels DSL definition of Equation 5.3

comprehensive constant folding such that all redundant calculations are pre-calculated once on the CPU and passed to the GPU as coefficients. For example, the redundant calculations λ^2, $\mu-1$ and $\mu+1$ in Equation (5.3) are removed as demonstrated in the code comparison below:

```
// Auto-generated version with redundant calculations.
t1x0y0 = ((2*t0x0y0)+((mu-1.0)*tM1x0y0)+
            (lambda*lambda*(t0x0y1+t0x0yM1+t0x1y0+t0xM1y0-(4*t0x0y0))))/(mu+1.0)

// Manual optimised version.
t1x0y0 = (((2*t0x0y0)+((muOne)*tM1x0y0)+
            (lambdaOne*(t0x0y1+t0x0yM1+t0x1y0+t0xM1y0-(4*t0x0y0))))*(muTwo));
```

Here it can be seen that the manual version pre-calculates λ^2, $\mu-1$ and $\mu+1$ on the CPU and whilst these remain constant, they are sent to the GPU as coefficients labelled *lambdaOne*, *muOne* and *muTwo* respectively.

8.1.1.1 Results

Figure 8.6 presents the performance results between the auto-generated and manually developed GPU programs. The *Simple Single Model* only involves a single, simple finite-difference equation and therefore there is not much room for manually optimising it. Therefore, this test only includes one optimisation, constant folding. This means the difference in performance between the manual and auto-generated version primarily depends on the inclusion of constant folding. The graph plots the total execution time to accumulate 44100 samples for a second of audio at the minimum accepted sample rate. For grid resolutions between $(N_x, N_y) = [64, 256]$, there is comparatively insignificant differences between the performance. From $(N_x, N_y) > 256$, the disparity begins to emerge, the manually written program begins to perform increasingly better. This enables the manually written version to operate at higher resolutions up to around $(N_x, N_y) = 700$, while the auto-generated version is limited to $(N_x, N_y) = 512$.

The only clear optimisation used in the manual version is the removal of redundant calculations using constant folding. The calculations are involved across 88% of the simulated space. To demonstrate the difference in theoretical performance using FLOP estimates, the auto-generated version at $(N_x, N_y) = 512$ is estimated to require $512*512*44100*14*0.88 = 142.425$GFLOPS whilst the manual version has a reduced estimated requirement of $512*512*44100*11*0.88 = 111.906$GFLOPS because of the constant folding. In the results, the difference is observed to be 957.0859ms for the GPU auto-generated version down to the 474.387ms of the manual equivalent giving the manual version a considerable 101% speedup over the auto-generated version. This demonstrates

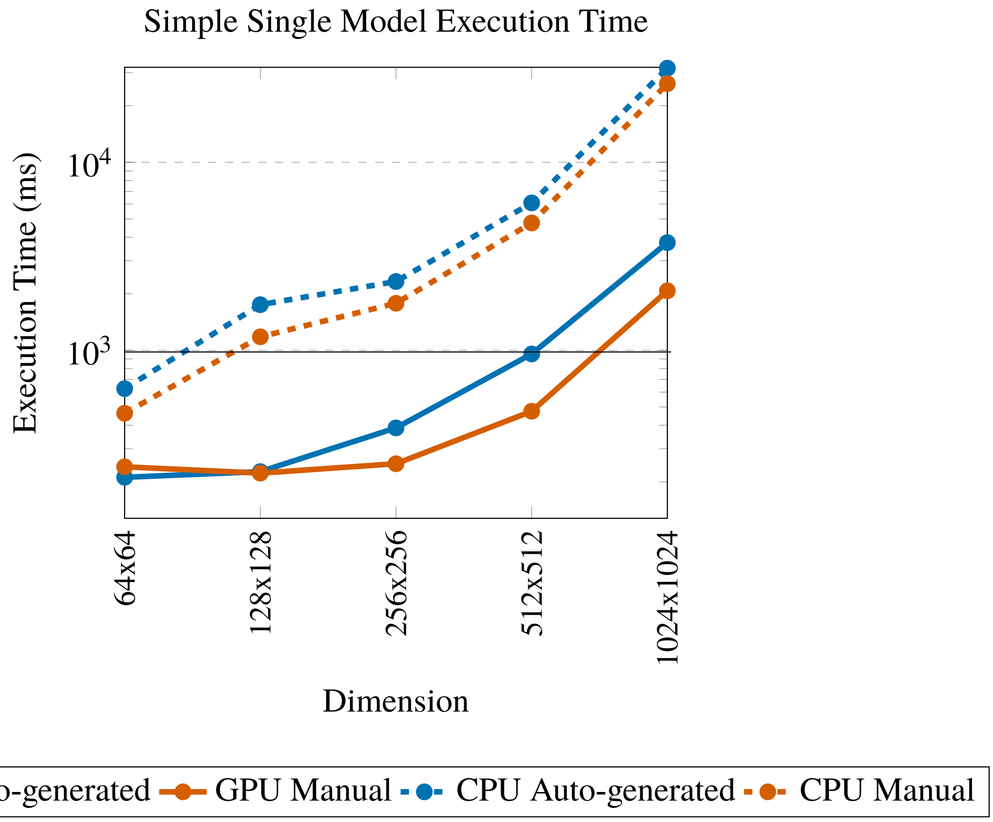


Figure 8.2: Execution time for a second's worth of sample at 44.1KHz with the *simple single model test*. **Spec**: **GPU** = NVIDIA RTX 2080

that the reducation of operations from constant folding is highly effective, especially for simulated spaces that have a high utilisation such as 88%.

8.1.2 Simple Multiple Model

The *simple multiple model* test is designed to continue testing simple equations involving a limited number of physical components but scales up the number of separate models. This is achieved by creating an environment of ten strings with identical equations and sharing the same set of parameters. This is a common arrangement for some string instruments with the same type and thickness but varying lengths. This test will arrange ten horizontal strings as shown in Figure 8.3. These strings take up considerably less space than two-dimensional shapes at a significantly smaller 1.8% grid utilisation.

The system of multiple string equations uses the notation introduced in Section 2.2.8.1 and is expressed as:

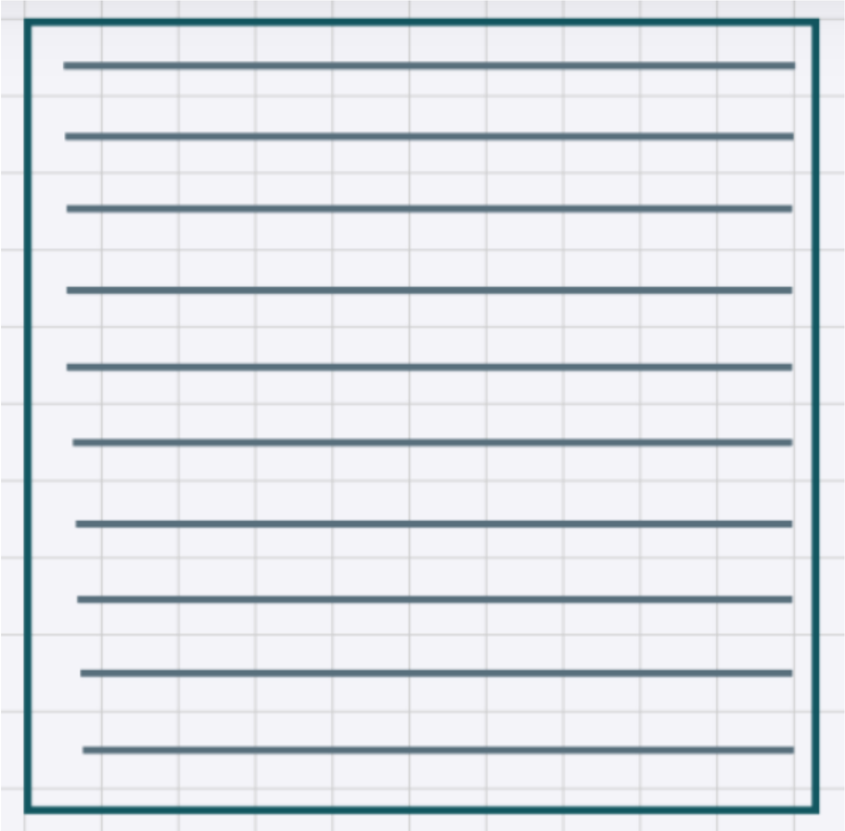


Figure 8.3: SVG representation of *Simple Multiple Model* geometry.

$$u_{tt}^{q} + \mu u_{t}^{q} = \gamma^{2} u_{xx}^{q} \quad \text{for} \quad q = 1, \dots, 10 \tag{8.3}$$

Where γ and μ are universal (meaning they apply to all 10 equations) string parameters for wave propagation and general damping, respectively. Using centered second-order finite-differences for u_tt and u_xx and a backward finite-difference for u_t, the following recursively solvable explicit scheme can be formed[4]:

$$u_{i}^{q,n+1} = \frac{2u_{i}^{q,n} + \lambda^{2}(u_{i-1}^{q,n} + u_{i+1}^{q,n} - 2u_{i}^{q,n})}{1+\mu} \tag{8.4}$$

Here, each string through $q = 1, \dots, M$ can be solved with the same universal parameters λ and μ. Therefore, the manual version can take advantage of this, and all strings can be captured as a group to apply the same equation and parameters as if they are the same shape ID. The auto-generated program creates the following lines of code for applying physics equations to the environment:

[4] See Figure A.2 in the appendix for the HyperModels DSL definition of Equation 8.4

```
if(idGrid[centreIdx] == 0) {
    t1x0y0 = 0.0;
} else if(idGrid[centreIdx] == 1) {
        t1x0y0 = (((2*t0x0y0)+((lambda*lambda)*(t0x1y0-(2*t0x0y0)
                    +t0xM1y0))-tM1x0y0) * (1.0/(mu+1.0)));
}
//...Repeated up to idGrid[centreIdx] == 10
```

Here, although the same equation with the same parameters are applied to all 10 strings, ten separate branching conditional statements are used for the same calculation. This is unnecessary and as covered in Section 6.2, branching is detrimental for GPU performance. Therefore, when manually writing the program, the branching can be avoided by grouping the conditional statement into:

```
if(idGrid[centreIdx] > 0) {
    t1x0y0 = (((2*t0x0y0)+(stringLambda*((modelGrid[t0x1y0Idx] *
    (1-boundaryGrid[rightIdx]))-(2*t0x0y0)+(modelGrid[t0xM1y0Idx] *
    (1-boundaryGrid[leftIdx]))))-tM1x0y0)*stringMu);
}
```

Here, the same equation and parameters *stringLambda* and *stringMu* are used, therefore, if any of the IDs between 1-10 are read from *idGrid*, then they can use this same calculation without branching unnecessarily. This example is only concerned with measuring performance and therefore uses the same parameters *stringLambda* and *stringMu* across all strings. However, this is not often desirable in practical applications but having a list of separately controllable *stringLambda* and *stringMu* for each string can be readily supported.

8.1.2.1 Results

The results for the *simple multiple model* test are provided in Figure 8.4. The results follow the same trend as the *simple single model* test, the GPU versions can support much higher resolutions of around $(N_x, N_y) < 800$, whilst the CPU versions can only reach $(N_x, N_y) = 128$ at most. Although this test includes 10 different physical models, this test is far less intensive with only 1.8% of the simulation space being utilised, meaning it can support the higher resolution range of $(N_x, N_y) < 800$ whilst the *simple single model* test can support up to $(N_x, N_y) < 700$. This is a significant difference that highlights the simulation space utilisation is an important component effecting performance.

The FLOPS estimated between the GPU auto-generated version at $(N_x, N_y) = 512$ is estimated to require $512*512*44100*11*0.018 = 2.288$GFLOPS and the manual version is reduced to an estimated $512*512*44100*8*0.018 = 1.664$GFLOPS. In the results at

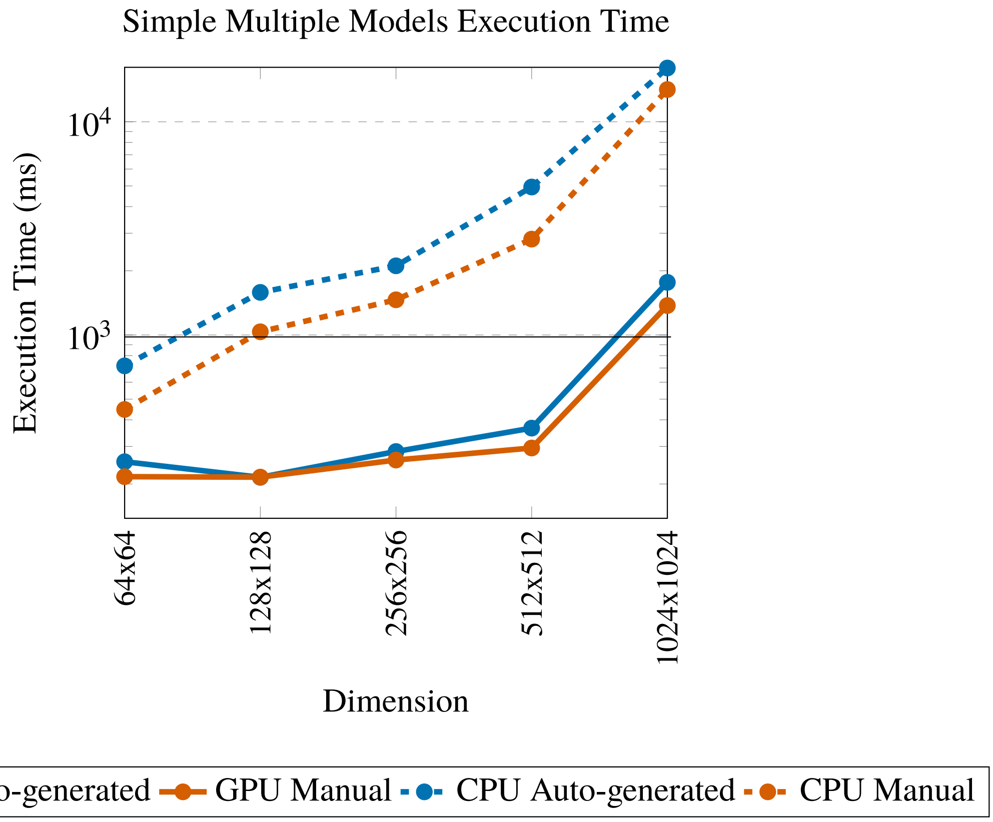


Figure 8.4: Execution time for a second's worth of sample at 44.1KHz with the *simple multiple model test*.
Spec: **GPU** = NVIDIA RTX 2080

$(N_x, N_y) = 512$, the difference is observed as 365.655ms for the GPU auto-generated and 295.0761ms for the manual version giving it roughly 23% speedup.

Here, the performance difference between the auto-generated and manual version exists but is less significant than in the 101% speedup seen in *simple single model* test. This is likely because the optimal calculation optimisation applies only to 1.8% of the points. However, it is not clear if the optimisation grouping models into a single if-statement has a significant effect on the performance over the auto-generated version, which does not have the foresight to do this. Note that this optimisation has limits and can not always be used in simulations with multiple models.

8.1.3 Complex Single Model

Willemsem et al. extend the Kirchoff thin plate model from Equation (2.24) with a general damping component as (Willemsen et al., 2017, p. 5):

$$u_{tt} = -\kappa^2 \Delta\Delta u - \sigma u_t \tag{8.5}$$

Here, $u = u(x, y, t)$ is the transverse plate deflection and is defined for $x \in [0, L_x]$, $y \in [0, L_y]$ are horizontal and vertical plate dimensions respectively. κ^2 is the referred to as the plate stiffness parameter:

$$\kappa^2 = \frac{EH^2}{12\rho(1-v^2)} \tag{8.6}$$

Where ρ is a material density, H is the plate thickness, and the constant E and v are Young's modulus and Poisson's ratio. By using a discrete bi-harmonic operator for approximating $\Delta\Delta u$, a second-order central difference for u_{tt} and first-order central difference for u_t, the following finite-differences are arranged:

$$\delta_{tt} u = -\delta_{\Delta\boxplus,\Delta\boxplus} u - \sigma\delta. \tag{8.7}$$

By expanding the finite-differences to their constituent components, the following recursively solvable explicit scheme can be formed (Willemsen et al., 2017, p. 13)[5]:

$$\begin{aligned}(1+\sigma T)u^{n+1}_{l,m} = 2u^n_{l,m} - (1-\sigma T)u^{n-1}_{l,m} - \mu^2(u^n_{l+2,m} + u^n_{l-2,m} + u^n_{l,m+2} + u^n_{l,m-2})\\ -2\mu^2(u^n_{l+1,m+1} + u^n_{l+1,m-1} + u^n_{l-1,m+1} + u^n_{l-1,m-1})\\ +8\mu^2(u^n_{l+1,m} + u^n_{l-1,m} + u^n_{l,m+1} + u^n_{l,m-1}) - 20\mu^2 u^n_{l,m}\end{aligned} \tag{8.8}$$

This explicit scheme will be used for a circular shape inside the environment illustrated in Figure 8.5. Here, the circle has moderate grid utilisation at 61%, which is less than the 88% of *simple single model* but involves a more advanced physics scheme.

8.1.3.1 Results

The results for the *complex single model* test are presented in Figure 8.6. In the graph, the auto-generated and manual versions execute at similar speeds, but the manual version is slightly faster than the auto-generated version. For example, at $(N_x, N_y) = 512$, the auto-generated measures in at 747ms and manual at 730ms, giving an insignificant speedup of approximately 2% to the manual version. In the broad scope, this is a negligible difference that does not enable the manual version to support any higher resolutions at any point, unlike in the *single simple model* and *multiple simple model* tests. As this test only involves a single model, removing redundant calculations is the primary optimisation,

[5] See Figure A.3 in the appendix for the HyperModels DSL definition of Equation 8.8

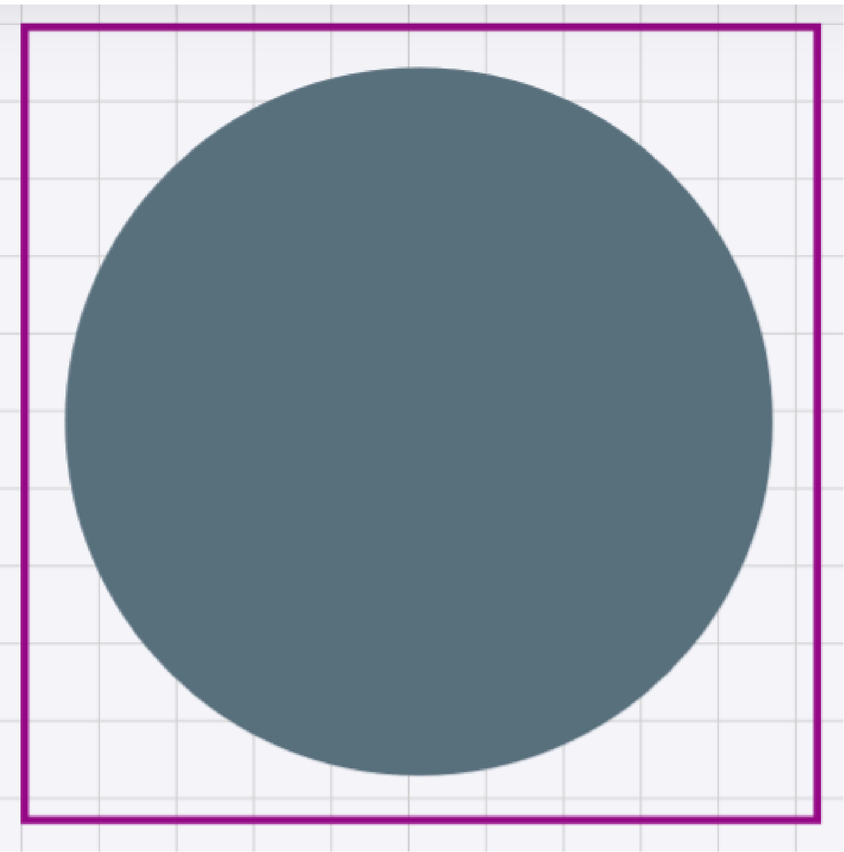


Figure 8.5: SVG representation of *Complex Multiple Model* geometry.

and there is no opportunity for grouping equations like in *simple multiple model*. The FLOPS estimated between the GPU auto-generated version at $(N_x, N_y) = 512$ is estimated to require $512*512*44100*29*0.61 = 204.506$GFLOPS and the manual version is reduced to an estimated $512*512*44100*23*0.61 = 162.194$GFLOPS. Oddly, these results have a negligible 2% difference despite having a greater difference in the number of estimated FLOPs than the *Simple Single Model* test. This could be because the theoretical floating point estimation is considerably inaccurate meaning the constant folding optimisation will have unexpected results or there are some other nuanced differences between manual and auto-generated versions not accounted for.

In similar work, the same differential equation with an additional frequency dependant component was used in Bilbao and van Walstijn (2005) to simulate a linear plate for audio synthesis. Although executing a grid of 26x32 points in their implementation, it was reported to take 22.6 seconds to output 1 second of sound at a sample rate of 44100Hz. Although the exact details of the implementation are not known, it was CPU based and possibly executed serially. Their conclusion discusses how the finite-difference scheme is

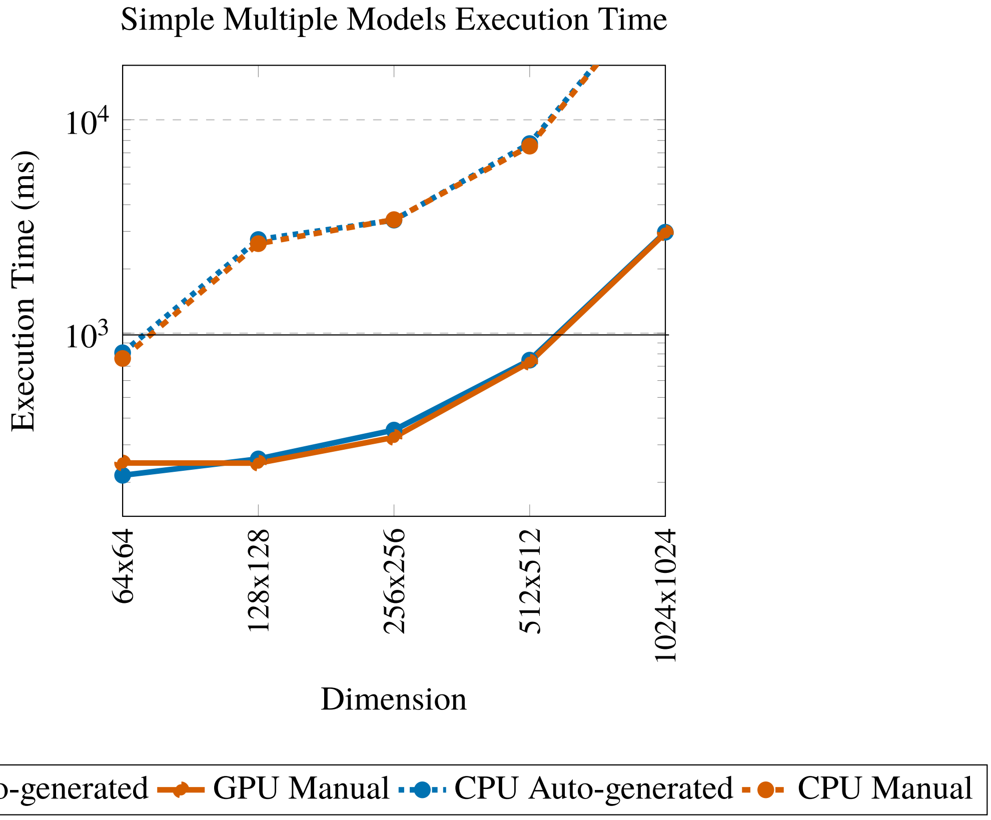


Figure 8.6: Execution time for a second's worth of sample at 44.1KHz with the *complex single model test*. **Spec**: **GPU** = NVIDIA RTX 2080

not computationally cheap, comparing it to having calculation costs of the same order of magnitude as modal synthesis. However, the auto-generated GPU accelerated physical modelling program enables far greater resolutions up to at least 512x512 that can operate in real-time, significantly improving over a 26x32 grid in 22.6 seconds.

8.1.4 Complex Multiple Model

The *complex multiple model* test uses a combination of all the equations used previously in the other tests. The system is made up of three separate models as shown in Figure 8.7, a rectangle with state function u, a 1-dimensional string v and circle w, resulting in a grid utilisation of approximately 51%. This test adds the inclusion of connection points between all three models. The rectangle u uses the two-dimensional wave equation from Equation (5.1) and has a one-way connection to the string. The string v uses Equation (8.3) and has a one-way connection to the circle w which uses Equation (8.5). Following the previous recursively solvable explicit schemes lead to the following definitions for u, v and w[6]:

$$u_{x,y}^{n+1} = \frac{2u_{x,y}^{n} - (\mu - 1)u_{x,y}^{n-1} + \lambda^2(u_{x+1,y}^{n} + u_{x-1,y}^{n} + u_{x,y+1}^{n} + u_{x,y-1}^{n} - 4u_{x,y}^{n})}{1+\mu} \tag{8.9}$$

$$v_i^{n+1} = \frac{2u_i^n + \lambda^2(v_{i-1}^n + v_{i+1}^n - 2v_i^n)}{1+\mu} \tag{8.10}$$

$$\begin{aligned}(1+\sigma T)w_{l,m}^{n+1} = 2w_{l,m}^{n} - (1-\sigma T)w_{l,m}^{n-1} - \mu^2(w_{l+2,m}^{n} + w_{l-2,m}^{n} + w_{l,m+2}^{n} + w_{l,m-2}^{n})\\ -2\mu^2(w_{l+1,m+1}^{n} + w_{l+1,m-1}^{n} + w_{l-1,m+1}^{n} + w_{l-1,m-1}^{n})\\ +8\mu^2(w_{l+1,m}^{n} + w_{l-1,m}^{n} + w_{l,m+1}^{n} + w_{l,m-1}^{n}) - 20\mu^2 w_{l,m}^{n}\end{aligned} \tag{8.11}$$

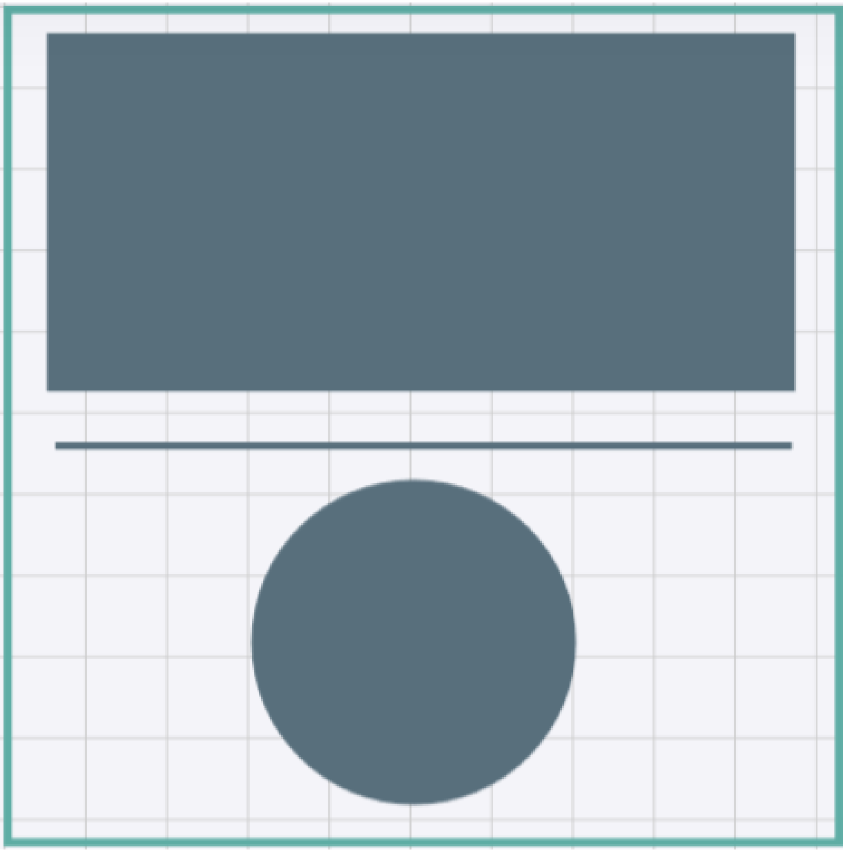


Figure 8.7: SVG representation of *Complex Multiple Model* geometry.

[6] See Figure A.4 in the appendix for the HyperModels DSL definition of Equation 8.9

8.1.4.1 Results

The results of the *Complex Multiple Model* test shown in Figure 8.8 are similar to the *Complex Single Model* test. Both support up to $(N_x, N_y) < 512$, even though *Complex Multiple Model* utilises less of the grid at 51% and *Complex Single Model* at 61%. Despite 10% lower grid utilisation, the test involves handling two connection points between the models. The manual versions are slightly faster but appear to be a mostly negligible improvement. With lower grid utilisation, the optimised calculations provide less benefit than those seen to accumulate in *simple single model test.* In the results at $(N_x, N_y) = 512$, the difference is observed as 649.684ms for the GPU auto-generated and 614.943ms for the manual version giving it roughly 5.6% speedup. The results suggest that involving a couple of connection points (and therefore conditional components) does not significantly impact performance. This is promising as, in theory, adding conditional components risks reducing performance, and the inclusion of connection points is a powerful fundamental feature of the framework. A deeper investigation would need to be made in order to determine how connection points effect performance. For example, how would this scale to hundreds of connection points?

8.1.5 Discussion

The results presented in this chapter expose important performance information about the auto-generated physical model programs related to manually developed versions. The difference between auto-generated and manual performance was contextual and depended on the arrangement in each test. This can be seen in the inconsistency between the speedup observed between the auto-generated and manual versions that ranges from as low as 2% in test *Complex Single Model* to as high as 101% in test *Simple Single Model.* The *Simple Single Model* isolated and highlighted the most effective manual version's optimisation was the use of constant folding that removed redundant calculations in the equations. Whilst the HyperModels design supports constant folding, the implementation does not currently include this feature. Therefore, with further development of the code generation, this optimisation can be added to the auto-generated programs. Considering that the results gathered only highlighted constant folding as a significant optimisation, this suggests that adding constant folding to this implementation of HyperModels would achieve similar performance to the manual version.

Aside from constant folding, the primary benefit reflected on by the author is that the manually developed versions is that the source code is more readable, making it more

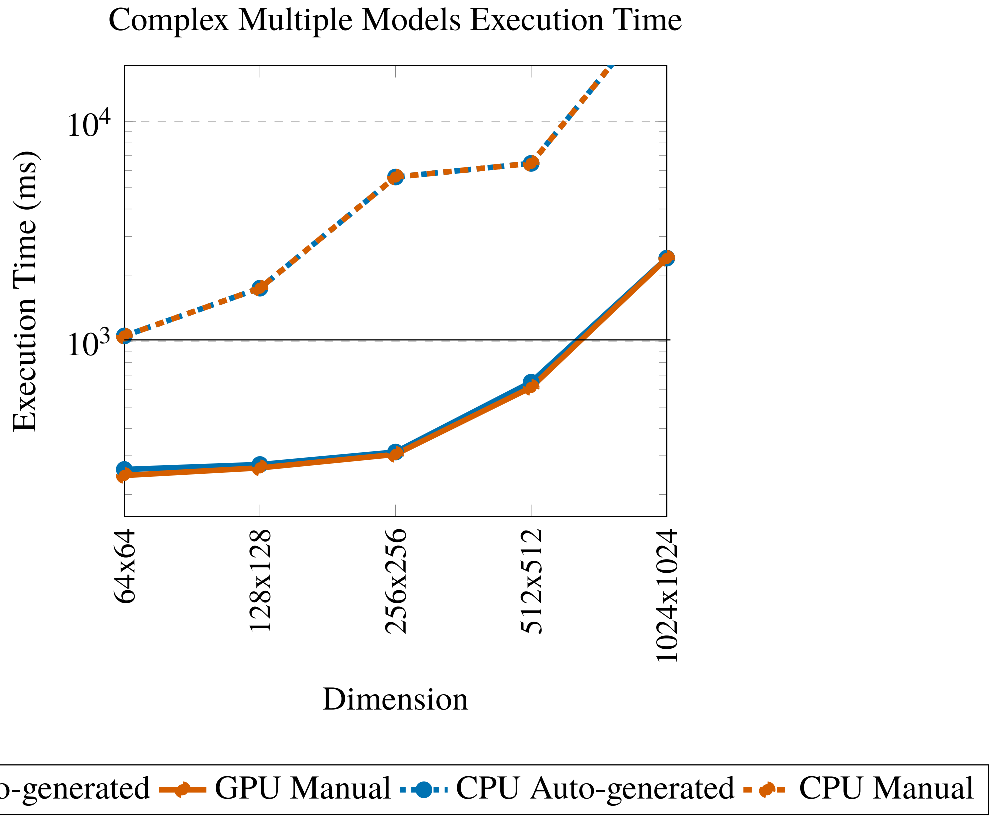


Figure 8.8: Execution time for a second's worth of sample at 44.1KHz with the *complex multiple model test*.

Spec: **GPU** = NVIDIA RTX 2080

manageable if the user makes modifications. The auto-generated programs were typically harder to read and understand, which could present issues if a user intends to extend their programs generated by HyperModels with their own modifications. The readability of HyperModel's auto-generated programs is related to the overall usability of HyperModels and must be rigorously evaluated in the future with organised user studies. One of the focuses of the user study would be to determine if the auto-generated programs can be comfortably modified to meet the user's needs.

Further performance differences were observed between the parallel CPU and GPU versions of the physical modelling. Whilst the CPU versions struggled to support tests for $(N_x, N_y) = 64$, the GPU auto-generated versions could support up to $(N_x, N_y) = 512$, a simulation space $64\times$ bigger. This reinforces the output from Part 1 that the GPU is more suitable than the CPU for simulating high-resolution finite-difference schemes based on linear equations. This provides further evidence for answering research question 5 showing that the GPU is more suitable for higher resolution simulations.

Analysing the results revealed another key component other than the model resolution affecting simulation performance, simulation space utilisation. The amount of simulation space being involved in the calculations affects performance significantly. For example, a simulation utilisation of 1.8% can be $1.28\times$ times faster than a simulation with the same resolution but with 81% utilisation. The performance impact of connection points was evaluated and shown to have no considerable effect when 3 connections were used. Further experiments scaling the number of connections up to hundreds will be required to understand how this scales.

The theoretical floating point estimations of each program showed that the expected improvements from the number of operations in the calculations reduced using constant folding could not be reliably predicted. This is likely because either the number of FLOPs being estimated is considerably inaccurate or that there are other nuanced differences between the auto-generate and manually written versions that disrupt the performance more than the number of operations in the calculations.

Ultimately, this leads to the conclusion that there is room to improve the HyperModel's framework with further performance analysis with a different set of goals. These goals would be targeted at evaluating if the programs are compute or bandwidth bound and then using Roofline analysis (Yang et al., 2020) to determine how close to peak performance. The new knowledge generated from such an investigation could be used to subsequently improve the HyperModels framework to reach maximum potential.

Chapter 9

Case Studies

This chapter demonstrates the expressiveness of HyperModels in practice by developing two real-time instruments using the framework. The first is the *Hyper Drumhead*[1], previously presented by Zappi et al. in hand optimised GPU code (Zappi et al., 2017; Zappi, 2017). The second is a variant of Willemsen et al. hammered dulcimer (Willemsen, Andersson, Serafin and Bilbao, 2019*a*), that originally operated with a plate model of 17x6 points running on the CPU, that has been ported to the GPU using HyperModels to support resolutions up to 256x256. Both instruments will operate within the system a defined in the HyperModels mapping with a resolution of (C_x, C_y) where models can be defined to operate at specific positions determined by the vector-based description of the model shapes.

9.1 Instrument 1: Twin Drumhead Membrane

Instrument 1 implements the *Hyper Drumhead* using the HyperModels Framework. The original design involves a two-dimensional simulation of the wave equation accelerated on the GPU using the graphics pipeline. On the tested systems, the *Hyper Drumhead* was reported to support resolutions up to 320x320. In their implementation, Zappi et al. mapped the audio domain of the physical model directly into the graphical domain using the OpenGL graphics rendering API. By conforming to the graphical domain, this design requires an additional field of knowledge and imposes some limitations. For Instrument 1, the *Hyper Drumhead* will be ported to the HyperModels framework with an extended resolution of $C_x = C_y = 512$. The source code and recordings demonstrating this

[1] The name of this instrument inspired the naming of the HyperModels framework

instrument are publicly available online [2].

The PDE used in the *Hyper Drumhead* is based on the two-dimensional wave equations with a frequency independent damping component (Renardy and Rogers, 2006). Equation (2.13) can be extended to include damping, and the recursively solvable explicit scheme used for the q^{th} model v_q where $q = 1, 2$ will be defined as[3]:

$$\begin{aligned}(1+\sigma_q)v^{n+1}_{q,l,m} &= 2v^{n}_{q,l,m} - (1-\sigma_q)v^{n-1}_{q,l,m} \\ &\quad + (\lambda_q)^2(v^{n}_{q,l+1,m} + v^{n}_{q,l-1,m} + v^{n}_{q,l,m+1} + v^{n}_{q,l,m-1} - 4v^{n}_{q,l,m})\end{aligned} \tag{9.1}$$

where the Courant number λ_q is defined as in Equation (2.13), and σ_q is the frequency independent damping coefficient (in s^{-1}). To maintain stability inside the model, the conditions $\lambda \leq \frac{1}{\sqrt{2}}$ and $0 < \sigma < 1$ must be maintained. Continuing to use the grid of PEs a from the HyperModels description in Section 7.2.1, v_q is mapped into a following the geometry illustrated as an SVG in Figure 9.1b.

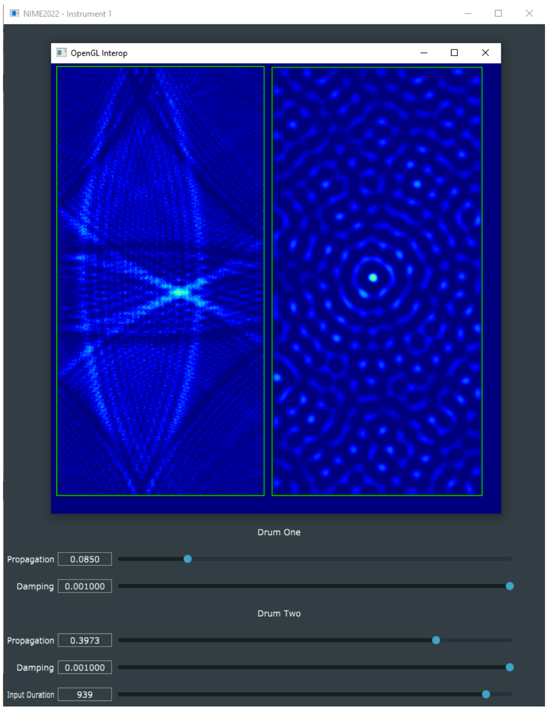
The details of the physical model described so far are contained in the *instrument* component visualised in Figure 9.2. Here, the CPU application program is written in the JUCE[4] audio framework interfaces with the GPU *instrument* program requesting 44100 samples per second. However, the input/output samples between CPU and GPU uses a buffering technique with data transfers at a rate of $\left\lceil \frac{r_s}{b_s} \right\rceil$ where r_s is the sample rate and b_s is the buffer length. So for a buffer length $b_s = 256$ at $r_s = 44100$, there are $\lceil \frac{44100}{256} \rceil = 173\text{Hz}$ data transfers per second. The state of the system a remains in the GPU global memory and is not transferred back to the CPU. Instead, for the on-screen visualisation of the model, an OpenGL graphics program is called at a rate of 15Hz. This efficiently maps the state of the instrument into coloured pixels that are then sent directly to a display device. Interactions with the instrument are controlled using two Sensel Morphs [5]. The Sensel Morph is a high-resolution pressure sensor that detects the position and amount of pressure of contacts. The two Sensel morph's have been connected to the application, one mapping to model v_1 and the other to v_2 by adding excitation to the system a_{c_x,c_y} at a position c_x and c_y that is detected by the Sensels. The Sensels are polled for contacts at a rate of 150Hz as this provides a maximum detection latency of 6.6ms (Willemsen, Andersson, Serafin and Bilbao, 2019*a*). A screenshot of Instrument 1's application GUI is shown in Figure 9.1a.

[2] `https://github.com/Harri-Renney/-NIME2022---InstrumentOne`

[3] See Figure A.5 in the appendix for the HyperModels DSL definition of Equation 9.1

[4] `https://juce.com`

[5] `https://github.com/sensel/sensel-api`



Drum One

Propagation 0.0850

Damping 0.001000

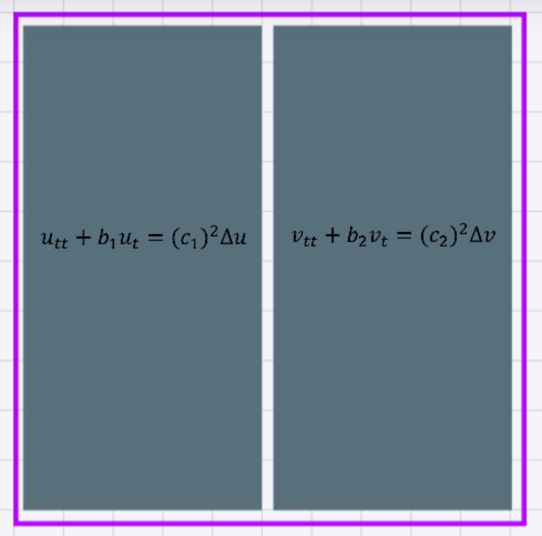
Drum Two

Propagation 0.3973

Damping 0.001000

Input Duration 939

(a) Screenshot of *Hyper Drumhead* application.


$u_{tt} + b_1 u_t = (c_1)^2 \Delta u$
$v_{tt} + b_2 v_t = (c_2)^2 \Delta v$

(b) Vector graphics and physical model description for u and v in system a.

Figure 9.1: Instrument 1 *Hyper Drumhead*

9.2 Instrument 2: String connected plates

Instrument 2 demonstrates how the GPU accelerated framework can represent more advanced, interconnected linear models. In Willemsen, Andersson, Serafin and Bilbao (2019*a*) the instrument designs use strings and plate models that are connected to form instruments such as the sitar, hammered dulcimer and Hurdy Gurdy. In their implementation, the instruments were executed on the CPU and operated for small resolutions, with strings involving a maximum of 50 points and plates of 20x10. The GPU accelerated implementation developed using HyperModels will support higher resolution models with multiple strings of 280 points and a plate of 236x121 inside an environment a with $C_x = C_y = 256$. A plate model v and multiple strings u_q will be defined where the subscript q is used to identify each string between $q = 1, \ldots, 13$.

The 13 strings are defined using the stiff string equation from Equation (2.21) along with frequency independent and frequency dependant components (Bensa et al., 2003) leading to the recursively solvable explicit scheme[6]:

[6] See Figure A.6 in the appendix for the HyperModels DSL definition of Equation 9.2

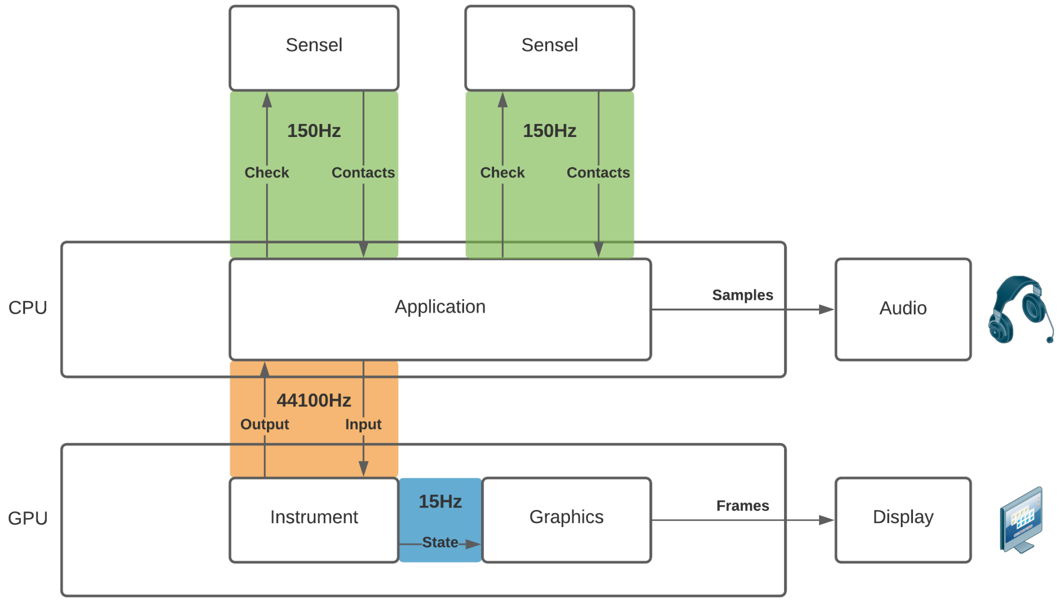


Figure 9.2: Application overview for Instrument 1: *Hyper Drumhead.*

$$
\begin{aligned}
(1+\sigma_{q,0}T)u_{q,l}^{n+1} = &\left(2-2(\lambda_q)^2-6(\mu_q)^2\frac{4\sigma_{q,1}T}{X^2}\right)u_{q,l}^{n} \\
&\left((\lambda_q)^2+4(\mu_q)^2+\frac{2\sigma_{q,1}T}{X^2}\right)(u_{q,l+1}^{n}+u_{q,l-1}^{n}) \\
&-(\mu_q)^2(u_{q,l+2}^{n}+u_{q,l-2}^{n}) \\
&+\left(-1+\sigma_{q,0}T+\frac{4\sigma_{q,1}T}{X^2}\right)u_{q,l}^{n-1} \\
&-\frac{2\sigma_{q,1}T}{X^2}\left(u_{q,l+1}^{n-1}+u_{q,l-1}^{n-1}\right)
\end{aligned}
\tag{9.2}
$$

with $\mu_q = \frac{\kappa_q T}{X^2}$, stiffness coefficient κ_q, frequency independent damping $\sigma_{q,0}$ and frequency dependant damping $\sigma_{q,1}$.

The linear plate equation (Morse and Ingard, 1986) uses a similar description of physics as the stiff string, including frequency independent and dependant components, but is extended to operate across two-dimensions. Using the explicit form from Equation (2.25) with damping components, the following recursively solvable explicit scheme is formed[7]:

[7] See Figure A.7 in the appendix for the HyperModels DSL definition of Equation 9.3

$$\begin{aligned}(1+\sigma_0 T)v_{l,m}^{n+1} = {} & (2-20\mu^2-4S)v_{l,m}^{n} \\ & + (8\mu^2+S)(v_{l+1,m}^{n}+v_{l-1,m}^{n}+v_{l,m+1}^{n}+v_{l,m-1}^{n}) \\ & - 2\mu^2(v_{l+1,m+1}^{n}+v_{l+1,m-1}^{n}+v_{l-1,m+1}^{n}+v_{l-1,m-1}^{n}) \\ & - \mu^2(v_{l+2,m}^{n}+v_{l-2,m}^{n}+v_{l,m+2}^{n}+v_{l,m-2}^{n}) \\ & + (\sigma_0 T-1+4S)v_{l,m}^{n-1} \\ & - S(v_{l+1,m}^{n-1}+v_{l-1,m}^{n-1}+v_{l,m+1}^{n-1}+v_{l,m-1}^{n-1})\end{aligned} \tag{9.3}$$

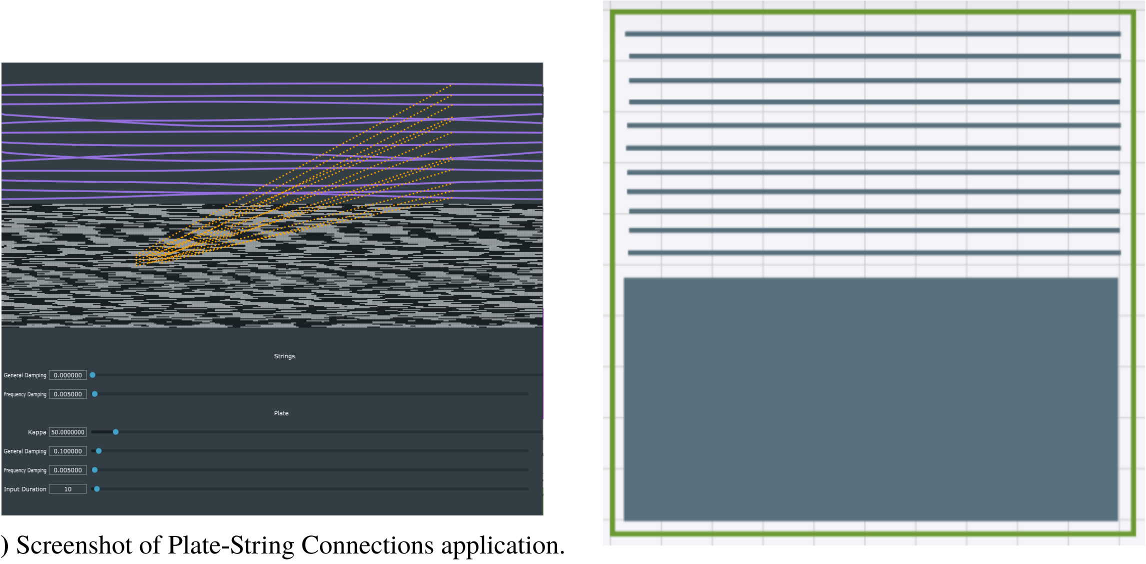
Where parameters are the same as in Equation (9.2) but with the additional coefficient $S = \frac{2\sigma_1 T}{X^2}$.

The geometry for instrument 2 is displayed as an SVG in Figure 9.3b. Again, a JUCE application running on the CPU interfaces with the instrument on the GPU at a sample rate of 44100Hz as shown in the instrument overview in Figure 9.4. The Sensel Morphs are used as input, one being mapped to pluck across the strings u^q using a triangle signal and the other to strike the plate v using an impulse. A key difference between Instrument 2 and 1 is that Instrument 2 generates the visualisation of the instrument on the CPU using the JUCE framework and needs to load it back to the GPU, accruing some additional overhead. Therefore, the state of the grid must be transferred from the GPU to the CPU to update the JUCE graphical components to then load onto the GPU at a frame rate of 15Hz. A screenshot of the Instrument 2 application is shown in Figure 9.3a; the demonstration recordings and source code for Instrument 2 are openly available online [8].

9.3 Case Studies Summary

This chapter presented two real-time finite-difference based physical modelled instruments developed using the thesis' proposed HyperModels framework. The first instrument adapts the design of the *Hyper Drumhead* to avoid the low-level graphics interface and instead use the abstracted, high-level framework of HyperModels. The HyperModels DSL is used to describe the finite-difference schemes and generates the optimised GPU simulation program. The C++ interface supports the integration of the GPU simulation into the JUCE audio playback and integrated interactions using an input device like the Sensel Morph. The second instrument based on the Hammered Dulcimer demonstrates that HyperModels can support more sophisticated models involving more advanced equa-

[8] https://github.com/Harri-Renney/NIME2022---InstrumentTwo



Strings

General Damping 0.000000

Frequency Damping 0.005000

Plate

Kappa 50.0000000

General Damping 0.100000

Frequency Damping 0.005000

Input Duration 10

(a) Screenshot of Plate-String Connections application.

(b) Vector graphics and physical model description for u_q and v in system a.

Figure 9.3: Instrument 2 Plate-String Connections

tions (particularly stiff string and plate equations) and interconnected models. These two instruments are just examples of what can be developed using the standardised and automated approach proposed by HyperModels. Future developments of instruments using HyperModels will be needed to demonstrate the full utility of the framework.

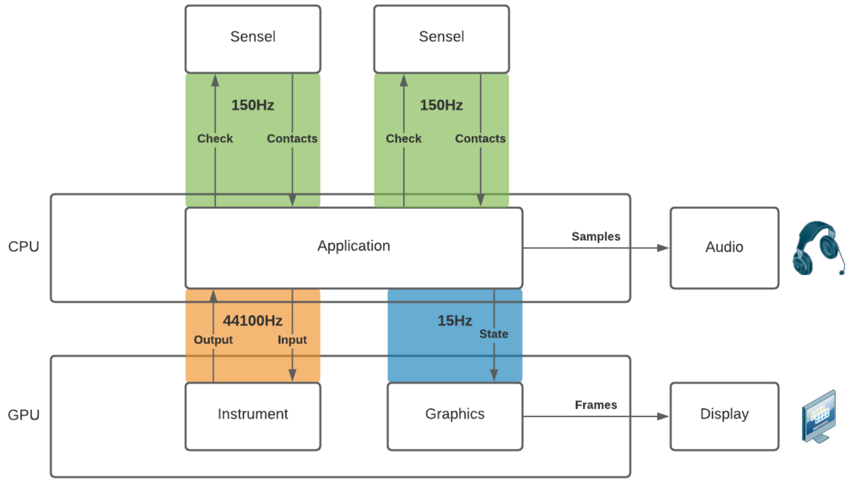


Figure 9.4: Application overview for Instrument 2: String-Plate Connections application.

Chapter 10

Conclusion

This chapter concludes this work by providing a summary of the thesis, along with some perspective on future contributions and avenues of research.

10.1 Summary

This thesis has presented the evaluation of the GPU as a hardware accelerator for digital audio and presented tools for facilitating the development of GPU accelerated physically modelled musical instruments. Part 1 built a series of performance benchmarking suites targeting offline and real-time digital audio requirements. For an offline evolutionary sound matching application, the GPU was shown to significantly improve offline performance by 5× over the equivalent parallel CPU version. However, offline physical modelling synthesis has a vast body of literature supporting it. Therefore, the thesis focuses on meeting real-time digital audio requirements. The results from the real-time benchmarking suite suggest that the buffer length of samples dispatched to the GPU was essential for meeting real-time requirements, being within the range of 32 to 512 samples to meet appropriate real-time audio-sound latency and sonic interaction requirements. The most significant improvement of the GPU observed was for real-time physical modelling synthesis. The GPU was capable of supporting 4× higher resolution two-dimensional models in real-time than the equivalent parallel CPU version. Part 2 presents the design of the HyperModels framework, a high-level framework for describing GPU accelerated, high-resolution physical model synthesisers. Beginning with detailed descriptions of all the physical model specific GPU optimised design components, these are then used to define the platform-agnostic design of HyperModels. An implementation of each component of HyperModels is given such that the performance of the framework can be evaluated with comparison to manually written equivalents. Finally, two complete instruments built using the HyperModels framework have been given to demonstrate the capability of the tools.

The HyperModels framework aims to bring support for real-time physical models to digital luthiers [1]. Whether realistic and traditional or novel and extraordinary, numerous new instruments can be developed using HyperModels, but the effectiveness will need to be refined and improved with future adoption.

10.2 Contributions

Chapter 3 summarised paper [A] where the practicality of the GPU within the domain of digital audio was tested and evaluated. This chapter provides the reader with quantifiable evidence presented in a way to give an understanding of the GPU when used for audio processing and synthesis. The primary contribution of this chapter is the evidence demonstrating that the GPU can reliably meet a set of real-time requirements, provided a particular range of buffer lengths is used. The buffer range of 32-512 reliably meets the core audio-sound latency and sonic interaction real-time requirements for discrete GPUs. Buffer lengths 8 and lower were shown to force the program to be bandwidth bound as the minimum latency of data transfers accumulate. Then when intensive audio processing is added, the synchronisation stages between each frame of execution needed for physical models means only a fraction of the GPUs theoretical peak performance is utilised. Some secondary findings where also highlighted as contributions including the suitability for integrated GPUs to be used for shorter buffer lengths below 32 and the considerable execution overhead for the first execution of a GPU program.

Chapter 4 involved the content of paper [B] and built off from the fundamental tests from [A] to develop a GPU accelerated offline evolutionary sound matching application. The primary contribution of this chapter is the proposed design for a GPU accelerated sound matching program that is shown to reliably outperform parallel CPU equivalents by approximately 8.88× for the configurations tested. The results provide evidence that the GPU is suitable for processing evolutionary algorithms when applied to advanced FM synthesis algorithms. This chapter also provides evidence that using lookup tables for trigonometric values instead of calling trigonometric functions improves performance by approximately 4×.

[1] Digital lutherie, a term coined by Jorda Jordà (2004), refers to the specialised domain Hirschfeld and Gelman (1994) and diverse community that is concerned with the creation of technology for music.

Chapter 5 uses the results from paper [C] to evaluate the effectiveness of the GPU for real-time physical modelling synthesis. This chapter's primary contribution is the extensive evidence supporting the GPU for large scale physical model processing. The results show that the CPU was more efficient at processing physical models with resolutions up to $(N_x, N_y) < 64$ but beyond this resolution the GPU outperformed the CPU and could support real-time requirements up to resolutions $(N_x, N_y) < 512$.

Paper [D] provides a brief overview of Part 2 of this thesis, where the HyperModels framework is described, tested and used to build example instruments. The primary contribution of Part 2 is the proposed design of the Hypermodels framework that provides a novel solution that facilitates the development of finite-difference based physical modelling for audio synthesis and DMI development. By specifically targeting this domain, HyperModels provides GPU acceleration and a visual method for describing physical model geometry which are both key features that are missing from the closest existing DSL called Faust. HyperModels was shown produce GPU programs with performance similar to manually written equivalent versions and was then shown to work in practice when building two DMIs from existing designs published by other researchers.

10.3 Perspectives and Future Work

Naturally, the content of this thesis opens up multiple avenues to explore in future work. The first part of this work highlights the strengths and weaknesses of the GPU in the field of digital audio processing. This provides a basis where researchers and designers can further explore the integration of the GPU as an audio processor. For example, after reading the test results, a designer may consider a real-time additive synthesiser that scales the number of oscillators further by using a GPU in their design. However, whilst the benchmarking suites have provided detailed insight into the comparison of software implementations and how the CPU and GPU compare for similar audio processes, a rigorous performance analysis is needed across the various available GPUs. Some targeted analysis between discrete and integrated GPUs has already been given, but these are broad categories and there are still a number of GPU specifications that need to be analysed, such as core count, core speed and memory bandwidth (McIntosh-Smith and Curran, 2014). It is already well understood in the existing literature (Sosnick and Hsu, 2010) (Skare and Abel, 2019) that these components of the GPU are important for audio synthesis, but quantifiable results are needed to produce an informed discussion on this topic. A thorough investigation can be made in a new study by assembling a collection of GPUs covering the range of existing modern GPU architectures and profiling their performance using the benchmarking suites available in this thesis. Analysing the results should lead to answering what GPU specifications influence the real-time audio performance the most? And ultimately, are the same components that impact graphics processing performance the same that determine audio processing performance? A study gathering evidence to answer these questions would assist readers in their choices when selecting the right GPU hardware for their audio applications.

The HyperModels design has only been through a short phase of development and no evidence for its usability has been gathered yet. Therefore, the next logical step is to understand the current usability of HyperModels with user studies that focus on evaluating the effectiveness of HyperModels to provide a GPU accelerated environment for building DMIs. The user studies could be framed to compare a user's own approach to developing GPU accelerated DMIs to the guided approach provided by HyperModels. This could provide qualitative feedback from users to understand some of the following:

1. Is HyperModels an effective approach for building DMIs?
2. Does the additional accessibility of the GPU provided by HyperModels improve the user's experience developing DMIs.
3. What stages of HyperModels blocks or negatively impacts the user's progress?

By analysing the feedback, contributions surrounding the field of DMI development and the accessibility of GPUs can be answered. Further, the usability of HyperModels can be improved by reflecting and revising the design based on the user feedback and experiences using it.

Once the usability of HyperModels has been evaluated and improved to an acceptable standard, improvements can be made to some of the technical details of HyperModels. HyperModels involves an amalgamation of concepts, ranging from DSLs, GPUs and finite-difference equations; therefore, the design and implementation would benefit from multiple experts in each respective field evaluating and suggesting technical improvements. The design for HyperModels is still relatively restricted to support linear systems that can be simulated using recursively solvable explicit schemes. These linear systems although highly suited for parallel processing and meeting real-time requirements, are fundamentally limited. However, the framework could be extended to support certain non-linear systems using iterative methods such as Newton Rhapson method (Bilbao et al., 2019). Further, other advance features can be added; for example, coupled first-order systems that use leap-frogging techniques to calculate single timesteps with multiple equations, such as those used in the state-of-the-art brass models in Bilbao and Harrison (2016) and Harrison-Harsley (2018). The boundary conditions were restricted to two basic forms; however, extending the DSL to enable users to define their own intricate boundary conditions would greatly improve the potential of the DSL. Furthermore, whilst HyperModels has been designed with the application of digital audio in mind, it could be revised to support physical models in general. For example, to simulate heat diffusion (Richter et al., 2013) or electro-magnetic waves through various materials (Wei et al., 2014). Whilst the development of DSLs in these fields have been extensively explored, the audio domain influence when using HyperModels could provide artistic value. With this in mind, it could provide novel experimental contributions by facilitating the exploration of different domains in ways not seen before within the context of digital audio.

With enough collective interest in HyperModels, further developments on the design will be explored and documented with the potential to continue expanding the audio landscape through the advancement of physical modelling audio synthesis.

Appendix A

HyperModels DSL Representations

All of the recursively solvable explicit finite-difference schemes used for building instruments in HyperModels have their respective HyperModels DSL definitions documented here.

```
1 = (2 * u(0)(0)(0) - (mu -1) * u(-1)(0)(0)
    + lambda^2 * (u(0)(1)(0) + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1)
    - 4 * u(0)(0)(0))) / (1 + mu)
```

Figure A.1: HyperModels DSL representation of Equation 5.3.

```
1 = (2 * u(0)(0) + lambda^2 * ( u(0)(-1) + u(0)(1) - 2 * u(0)(0)))
    / (1 + mu)

...

10 = (2 * u(0)(0) + lambda^2 * ( u(0)(-1) + u(0)(1) - 2 * u(0)(0)))
    / (1 + mu)
```

Figure A.2: HyperModels DSL representation of Equation 8.4.

```
1 = (2 * u(0)(0)(0) - (1 - \sigma * T) * u(-1)(0)(0)
    - mu^2 * (u(0)(2)(0) + u(0)(-2)(0) + u(0)(0)(2) + u(0)(0)(-2))
    - 2 * mu^2 * (u(0)(1)(1) + u(0)(-1)(1) + u(0)(1)(-1) + u(0)(-1)(-1))
    + 8 * mu^2 * (u(0)(1)(0) + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1))
    - 20 * mu^2 * u(0)(0)(0))
    / (1 + \sigma * T)
```

Figure A.3: HyperModels DSL representation of Equation 8.8.

```
1 = (2 * u(0)(0)(0) - (mu-1) * u(-1)(0)(0)
    + lambda^2 * (u(0)(1)(0) + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1)
    - 4 * u(0)(0)(0)))
    / (1 + mu)

2 = (2 * u(0)(0) + lambda*lambda * (u(0)(-1) + u(0)(1) - 2 * u(0)(0)))
    / (1 + mu)

3 = (2 * u(0)(0)(0) - (1 - sigma * T) * u(-1)(0)(0)
    - mu^2 * (u(0)(2)(0) + u(0)(-2)(0) + u(0)(0)(2) + u(0)(0)(-2))
    - 2 * mu^2 * (u(0)(1)(1) + u(0)(1)(-1) + u(0)(-1)(1)
    + u(0)(-1)(-1))
    + 8 * mu^2 * (u(0)(1)(0)
    + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1))
    - (20 * mu^2 * u(0)(0)(0)))
    / (1 + sigma * T)
```

Figure A.4: HyperModels DSL representation of Equation 8.9.

```
1 = (2 * u(0)(0)(0) - (muOne -1) * u(-1)(0)(0)
    + lambdaOne^2 * (u(0)(1)(0) + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1)
    - 4 * u(0)(0)(0))) / (1 + muOne)

2 = (2 * u(0)(0)(0) - (muTwo -1) * u(-1)(0)(0)
    + lambdaTwo^2 * (u(0)(1)(0) + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1)
    - 4 * u(0)(0)(0))) / (1 + muTwo)
```

Figure A.5: HyperModels DSL representation of Equation 9.1.

```
1 = ((2 - 2 * lambdaOne^2 - 6 * muOne^2 * ((4 * sigmaOneOne * T) / X^2))
    * u(0)(0)
    + (lambdaOne^2 + 4 * muOne^2 + ((2 * sigmaOneOne * T) / X^2))
    * (u(0)(1) + u(0)(-1))
    - mu^2 * (u(0)(2) + u(0)(-2))
    + (-1 + sigmaOneZero * T + ((4 * sigmaOneOne * T) / X^2)) * u(-1)(0)
    - ((2 * sigmaOneOne * T) / X^2) * (u(-1)(1) + u(-1)(-1)))
    / (1 + sigmaOneZero * T)

...

13 = ((2 - 2 * lambdaThirteen^2 - 6 * muThirteen^2 * ((4 * sigmaThirteenOne * T)
    / X^2)) * u(0)(0)
    + (lambdaThirteen^2 + 4 * muThirteen^2 + ((2 * sigmaThirteenOne * T) / X^2))
    * (u(0)(1) + u(0)(-1))
    - mu^2 * (u(0)(2) + u(0)(-2))
    + (-1 + sigmaThirteenZero * T + ((4 * sigmaThirteenOne * T) / X^2))
    * u(-1)(0)
    - ((2 * sigmaThirteenOne * T) / X^2) * (u(-1)(1) + u(-1)(-1)))
    / (1 + sigmaThirteenZero * T)
```

Figure A.6: HyperModels DSL representation of Equation 9.2.

```
1 = ((2 - 20 * mu^2 - 4 * S) * u(0)(0)(0)
    + (8 * mu^2 + S) * (u(0)(1)(0) + u(0)(-1)(0) + u(0)(0)(1) + u(0)(0)(-1))
    - 2 * mu^2 * (u(0)(1)(1) + u(0)(1)(-1) + u(0)(-1)(1) + u(0)(-1)(-1))
    - mu^2 * (u(0)(2)(0) + u(0)(-2)(0) + u(0)(0)(2) + u(0)(0)(-2))
    + (sigmaZero * T - 1 + 4 * S) * u(-1)(0)(0)
    - S * (u(-1)(1)(0) + u(-1)(-1)(0) + u(-1)(0)(1) + u(-1)(0)(-1)))
    / (1 + sigmaZero * T)
```

Figure A.7: HyperModels DSL representation of Equation 9.3.

Appendix B

Data Formats

Two particular data structures and formats are used in this thesis to capture the descriptions of the physical models. The SVG format is used for describing the geometry in a readily accessible and supported format, whilst JSON is used to capture the comprehensive physical model descriptions, from the physical model program to the finite-difference grid.

B.1 SVG

Scalable Vector Graphics (SVG) is an XML based two-dimensional graphic file format Eisenberg and Bellamy-Royds (2014). SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text. There are two ways of representing graphics: raster graphics, as already covered in Section 2.3.4 and vector graphics. In vector graphics, an image is described as a series of geometric shapes with appropriate attributes for describing them. Rather than containing a two-dimensional set of coloured pixels (a la raster format), an SVG contains commands for drawing the specified shape. The commands are instructions for drawing lines and curves to make the shapes, capturing the entire object this way is powerful for modifying and adapting graphics when displaying them in different environments. Because SVGs are not stored as pixels yet, they are scalable, meaning when changing the resolution, the quality of the image rendered is not impacted. SVG is an XML application and therefore brings the advantages of openness, transportability and interoperability Ferraiolo et al. (2000). This makes it a powerful format for gaining widespread support and conforming to a robust standard.

SVGs follow XML syntax and therefore opens with the standard XML processing instructions and DOCTYPE declaration. This is defined inside <*svg*> tags and contains the width and height of the finished graphic in pixels. Further attributes include the SVG namespace in *xmlns* and <*title*> tags to give the SVG a meaningful title. Shapes are then added within the SVG using shape tags, such as <*circle*>. <*circle*> requires attributes for the centre of the circle's x and y coordinates with *cx* and *cy*, along with the circle's radius *r* about this point. General presentation of shapes are set inside the *style* attribute, such as "fill:rgb(255,0,0);" to set the colour of the shape to red. A basic example of an SVG using the ideas covered is given in Code B.1 and rendered in Figure B.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
     width="120" height="120">
  <circle cx="60" cy="60" r="40" style="fill: red; stroke: black;"/>
</svg>
```

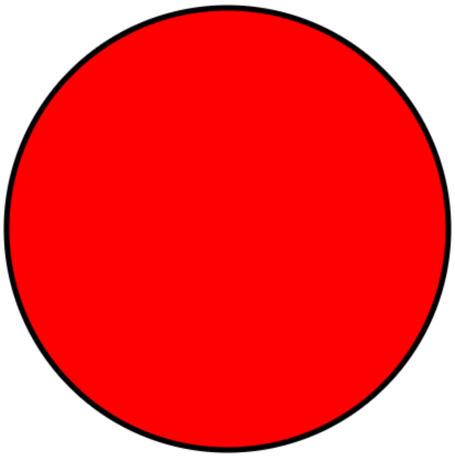


Figure B.1: SVG rendered in browser using Code B.1.

SVGs are a powerful way of describing two-dimensional graphics as they provide meaningful identities to shapes and relationships between them, unlike bitmaps storing pixels where the groups of pixels don't have a standard awareness of being a shape. This means SVG shapes can be assigned special attributes that can affect the whole shape directly. For example, an SVG can be rendered in the browser and then when clicked on, the SVG colour attribute could be modified to change blue and move to the other side of the screen. All this requires is modifying three attributes in the <circle>

shape, as opposed to working out a way to clear the pixels, and draw them to the other position without complications like aliasing. A further benefit gained form being a markup language is that meta data such as descriptions can be assigned to shapes, capturing further identities for the shapes Good (2005).

B.2 JSON

Javascript Object Notation (JSON) is an open standard file and data interchange format that uses human-readable text to store data Bray (2017). Data is stored in plain-text arrays and attribute-value pairs. JSON is considered a fast, resource efficient format that provides a language-independent medium for data to move through between incompatible software Nurseitov et al. (2009). For these reasons, JSON is commonly used to store data as it provides a standard format that applications can readily support.

Attribute-value pairs is a concept where a named attribute, such as "animal" is assigned an appropriate value, like "cat". The attribute must be a name defined as a string, values can take many different forms of data type such as values, arrays and objects. This enables JSON to store complex nested objects and arrays that can be handled conventionally. An example of a JSON file containing data about animals is shown in Code B.2. The JSON object here contains the name "animals" that has a value of an array of objects that include attribute-value pairs that hold data on animals.

```
{
    "animals": [
        {
            "name": "cat",
            "arms": 0,
            "legs": 4
        },
        {
            "name": "monkey",
            "arms": 2,
            "legs": 2
        }
    ]
}
```

Much like the SVG format, JSON conforms to a widely adopted standard, meaning it can be easily adopted into other software tools and exchanged between unrelated platforms.

Bibliography

Ahmad, S. and Ambrosetti, A. (2019), *Differential Equations: A first course on ODE and a brief introduction to PDE*, Walter de Gruyter GmbH & Co KG.

Ahmed, K. and Schuegraf, K. (2011), 'Transistor wars', *IEEE Spectrum* **48**(11), 50–66.

Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. (2020), *Compilers: principles, techniques and tools.*

Albano, J. (2016), 'An overview of logic pro x's powerful synths', `https://macprovideo.com/article/audio-software/an-overview-of-logic-pro-x-s-powerful-synths`. Accessed: 2020-05-03.

Alford, R., Kelly, K. and Boore, D. M. (1974), 'Accuracy of finite-difference modeling of the acoustic wave equation', *Geophysics* **39**(6), 834–842.

Allalen, M., Gray, A., Ilieva, N., Sjöström, A., Codreanu, V. and Weinberg, V. (2017), *Best Practice Guide – GPGPU.*

Annett, M., Ng, A., Dietz, P., Bischof, W. F. and Gupta, A. (2014), How low should we go? understanding the perception of latency while inking, *in* 'Proceedings of Graphics Interface 2014', pp. 167–174.

Ascher, U. M., Ruuth, S. J. and Spiteri, R. J. (1997), 'Implicit-explicit runge-kutta methods for time-dependent partial differential equations', *Applied Numerical Mathematics* **25**(2-3), 151–167.

Backus, J. (1979), 'The history of fortran i, ii and iii', *Annals of the History of Computing* **1**(1), 21–37.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B. et al. (1963), 'Revised report on the algorithmic language algol 60', *The Computer Journal* **5**(4), 349–367.

Baer, J.-L. (1980), *Computer systems architecture*, Vol. 11, Computer Science Press Rockville, Maryland.

Bailey, D. H. (1988), 'A high-performance fft algorithm for vector supercomputers', *The International Journal of Supercomputing Applications* **2**(1), 82–87.

Beauchamp, J. W. and Horner, A. (2003), 'Error metrics for predicting discrimination of original and spectrally altered musical instrument sounds', *The Journal of the Acoustical Society of America* **114**(4), 2325–2325.

Bensa, J., Bilbao, S., Kronland-Martinet, R. and Smith III, J. O. (2003), 'The simulation of piano string vibration: From physical models to finite difference schemes and digital waveguides', *The Journal of the Acoustical Society of America* **114**(2), 1095–1107.

Berg, R. E. and Stork, D. G. (1990), *The physics of sound*, Pearson Education India.

Bertrand, M. (1992), 'The cordic method for faster sin and cos calculations', *C Users Journal* **10**(11).

Beyer, H.-G. and Schwefel, H.-P. (2002), 'Evolution strategies–a comprehensive introduction', *Natural computing* **1**(1), 3–52.

Bilbao, S. D. (2009), *Numerical sound synthesis*, Wiley Online Library.

Bilbao, S., Hamilton, B., Torin, A., Webb, C., Graham, P., Gray, A., Kavoussanakis, K. and Perry, J. (2013), Large scale physical modeling sound synthesis, *in* 'Proceedings of the Stockholm music acoustic conference (SMAC2013), Stockholm', pp. 593–600.

Bilbao, S. and Harrison, R. (2016), 'Passive time-domain numerical models of viscothermal wave propagation in acoustic tubes of variable cross section', *The Journal of the Acoustical Society of America* **140**(1), 728–740.

Bilbao, S., Perry, J., Graham, P., Gray, A., Kavoussanakis, K., Delap, G., Mudd, T., Sassoon, G., Wishart, T. and Young, S. (2019), 'Large-scale physical modeling synthesis, parallel computing, and musical experimentation: the ness project in practice', *Computer Music Journal* **43**(2-3), 31–47.

Bilbao, S., Torin, A., Graham, P., Perry, J. and Delap, G. (2014), Modular physical modeling synthesis environments on gpu, *in* 'ICMC'.

Bilbao, S. and van Walstijn, M. (2005), A finite difference scheme for plate synthesis, *in* 'Proceedings of the International Computer Music Conference', pp. 119–122.

Bilbao, S. and Webb, C. (2012), Timpani drum synthesis in 3d on gpgpus, *in* 'Proc. of the 15th Int. Conference on Digital Audio Effects (DAFx-12), York, United Kingdom'.

Blake, G., Dreslinski, R. G. and Mudge, T. (2009), 'A survey of multicore processors', *IEEE Signal Processing Magazine* **26**(6), 26–37.

Blythe, D. (2008), 'Rise of the graphics processor', *Proceedings of the IEEE* **96**(5), 761–778.

Bocchino, R. L., Adve, V., Adve, S. and Snir, M. (2009), 'Parallel programming must be deterministic by default', *Usenix HotPar* **6**.

Bolin, M. (2010), *Closure: The definitive guide: Google tools to add power to your JavaScript*, " O'Reilly Media, Inc.".

Bray, T. (2017), 'The JavaScript Object Notation (JSON) Data Interchange Format', `https://rfc-editor.org/rfc/rfc8259.txt`.

Bristow-Johnson, R. (1996), Wavetable synthesis 101, a fundamental perspective, *in* 'Audio Engineering Society Convention 101', Audio Engineering Society.

Broesch, J. D. (2008), *Digital signal processing: instant access*, Elsevier.

Butcher, J. C. and Goodwin, N. (2008), *Numerical methods for ordinary differential equations*, Vol. 2, Wiley Online Library.

Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D. and McDonald, J. (2001), *Parallel programming in OpenMP*, Morgan kaufmann.

Chatterjee, S., Tasırlar, S., Budimlic, Z., Cave, V., Chabbi, M., Grossman, M., Sarkar, V. and Yan, Y. (2013), Integrating asynchronous task parallelism with mpi, *in* '2013 IEEE 27th International Symposium on Parallel and Distributed Processing', IEEE, pp. 712–725.

Chattopadhyay, D. (2006), *Electronics (fundamentals and applications)*, New Age International.

Chazarain, J. and Piriou, A. (2011), *Introduction to the theory of linear partial differential equations*, Elsevier.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K. (2009), Rodinia: A benchmark suite for heterogeneous computing, *in* '2009 IEEE international symposium on workload characterization (IISWC)', Ieee, pp. 44–54.

Cheng, J., Grossman, M. and McKercher, T. (2014), *Professional CUDA c programming*, John Wiley & Sons.

Chowning, J. and Bristow, D. (1986), 'Fm theory and applications', *By Musicians for Musicians* .

Chowning, J. M. (1973), 'The synthesis of complex audio spectra by means of frequency modulation', *Journal of the audio engineering society* **21**(7), 526–534.

Cook, P. R. (1993), 'Spasm, a real-time vocal tract physical model controller; and singer, the companion software synthesis system', *Computer Music Journal* **17**(1), 30–44.

Corporation, N. (2009), 'Nvidia fermi compute architecture whitepaper'.

Cosnard, M. and Trystram, D. (1994), *Parallel algorithms and architectures*, Thomson Learning.

Cummings, K. E., Maloney, J. G. and Clements, M. A. (1995), Modelling speech production using yee's finite difference method, *in* '1995 International Conference on Acoustics, Speech, and Signal Processing', Vol. 1, IEEE, pp. 672–675.

Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V. and Vetter, J. S. (2010), The scalable heterogeneous computing (shoc) benchmark suite, *in* 'Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units', ACM, pp. 63–74.

Das, S. and Suganthan, P. N. (2010), 'Problem definitions and evaluation criteria for cec 2011 competition on testing evolutionary algorithms on real world optimization problems', *Jadavpur University, Nanyang Technological University, Kolkata* pp. 341–359.

Davidson, A., Tarjan, D., Garland, M. and Owens, J. D. (2012), *Efficient parallel merge sort for fixed and variable length keys*, IEEE.

De Poli, G. and Prandoni, P. (1997), 'Sonological models for timbre characterization', *Journal of New Music Research* **26**(2), 170–197.

Demir, A. O. (2015), Mephisto: a source to source transpiler from pure data to faust, Master's thesis, Middle East Technical University.

Denning, P. J. and Lewis, T. G. (2017), 'Exponential laws of computing growth'.

Desell, T., Waters, A., Magdon-Ismail, M., Szymanski, B. K., Varela, C. A., Newby, M., Newberg, H., Przystawik, A. and Anderson, D. (2010), Accelerating the milkyway@home volunteer computing project with gpus, *in* R. Wyrzykowski, J. Dongarra, K. Karczewski and J. Wasniewski, eds, 'Parallel Processing and Applied Mathematics', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 276–288.

DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K. et al. (2011), Liszt: a domain specific language for building portable mesh-based pde solvers, *in* 'Proceedings of 2011 international conference for high performance computing, networking, storage and analysis', pp. 1–12.

Ducceschi, M. and Bilbao, S. (2016), 'Linear stiff string vibrations in musical acoustics: Assessment and comparison of models', *The Journal of the Acoustical Society of America* **140**(4), 2445–2454.

Duff, I. S., Heroux, M. A. and Pozo, R. (2002), 'An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum', *ACM Transactions on Mathematical Software (TOMS)* **28**(2), 239–267.

Duncan, R. (1990), 'A survey of parallel computer architectures', *Computer* **23**(2), 5–16.

Durran, D. R. (2013), *Numerical methods for wave equations in geophysical fluid dynamics*, Vol. 32, Springer Science & Business Media.

Eichenberger, A. E., Wu, P. and O'brien, K. (2004), Vectorization for simd architectures with alignment constraints, *in* 'Acm Sigplan Notices', Vol. 39, ACM, pp. 82–93.

Eisenberg, J. D. and Bellamy-Royds, A. (2014), *SVG essentials: Producing scalable vector graphics with XML*, " O'Reilly Media, Inc.".

Elkhouly, R., Kimura, K. and El-Mahdy, A. (2016), 'If-conversion optimization using neuro evolution of augmenting topologies', *arXiv preprint arXiv:1603.01112* .

Evans, G., Blackledge, J. and Yardley, P. (2012), *Numerical methods for partial differential equations*, Springer Science & Business Media.

Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M. and Zugal, S. (2009), Declarative versus imperative process modeling languages: The issue of understandability, *in* 'Enterprise, Business-Process and Information Systems Modeling', Springer, pp. 353–366.

Felleisen, M. (1991), 'On the expressive power of programming languages', *Science of Computer Programming* **17**(1), 35 – 75.
URL: *http://www.sciencedirect.com/science/article/pii/016764239190036W*

Ferraiolo, J., Jun, F. and Jackson, D. (2000), *Scalable vector graphics (SVG) 1.0 specification*, iuniverse Bloomington.

Ferrer, P. J. M., Buttay, R., Lehnasch, G. and Mura, A. (2014), 'A detailed verification procedure for compressible reactive multicomponent navier–stokes solvers', *Computers & Fluids* **89**, 88–110.

Flynn, M. (2011), 'Flynn's taxonomy', *Encyclopedia of parallel computing* pp. 689–697.

Flynn, M. J. (1972), 'Some computer organizations and their effectiveness', *IEEE transactions on computers* **100**(9), 948–960.

Folland, G. B. (2020), *Introduction to partial differential equations*, Princeton university press.

Freedman, M. D. (1967), 'Analysis of musical instrument tones', *The Journal of the Acoustical Society of America* **41**(4A), 793–806.

Frigo, M. and Johnson, S. G. (1998), Fftw: An adaptive software architecture for the fft, *in* 'Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)', Vol. 3, IEEE, pp. 1381–1384.

Fukuda, Y. (1985), *Yamaha DX7 digital synthesizer*, Music Sales Corp.

Gaster, B., Howes, L., Kaeli, D. R., Mistry, P. and Schaa, D. (2012), *Heterogeneous computing with openCL: revised openCL 1.*, Newnes.

Gaster, B. R., Renney, N. and Parraman, C. (2019), Fun with interfaces (svg interfaces for musical expression), *in* 'Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design', FARM 2019, Association for Computing Machinery, New York, NY, USA, p. 25–36.

Gates, A., Bradshaw, J. L. and Nettleton, N. C. (1974), 'Effect of different delayed auditory feedback intervals on a music performance task', *Perception & Psychophysics* **15**(1), 21–25.

Gerver, M. L. (1970), 'Inverse problem for the one-dimensional wave equation', *Geophysical Journal International* **21**(3), 337–357.

Good, J. (2005), 'The benefits and practicalities of using extensible markup language (xml) for the interfacing and control of object-oriented simulations'.

Gordon, M. I., Thies, W. and Amarasinghe, S. (2006), 'Exploiting coarse-grained task, data, and pipeline parallelism in stream programs', *ACM SIGARCH Computer Architecture News* **34**(5), 151–162.

Gregg, C. and Hazelwood, K. (2011), Where is the data? why you cannot debate cpu vs. gpu performance without the answer, *in* '(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software', IEEE, pp. 134–144.

Group, K. O. W. et al. (2013), 'The opencl spec v2. 0'.

Hambric, S. A. (2006), 'Structural acoustics tutorial—part 1: vibrations in structures', *Acoustics Today* **2**(4), 21–33.

Hamilton, B. and Webb, C. J. (2013), 'Room acoustics modelling using gpu-accelerated finite difference and finite volume methods on a face-centered cubic grid', *Proc. Digital Audio Effects (DAFx), Maynooth, Ireland* pp. 336–343.

Har, J. and Tamma, K. K. (2012), *Advances in computational dynamics of particles, materials and structures*, Wiley Online Library.

Harris, F. J. (1978), 'On the use of windows for harmonic analysis with the discrete fourier transform', *Proceedings of the IEEE* **66**(1), 51–83.

Harris, M. (2005), Mapping computational concepts to gpus, *in* 'ACM SIGGRAPH 2005 Courses', pp. 50–es.

Harrison-Harsley, R. L. (2018), 'Physical modelling of brass instruments using finite-difference time-domain methods'.

Hassan, S. A., Hemeida, A. and Mahmoud, M. M. (2016), 'Performance evaluation of matrix-matrix multiplications using intel's advanced vector extensions (avx)', *Microprocessors and Microsystems* **47**, 369–374.

Hassani, S. (2009), Dirac delta function, *in* 'Mathematical methods', Springer, pp. 139–170.

Havlicek, L. L. (1968), 'Effect of delayed auditory feedback on musical performance', *Journal of Research in Music Education* **16**(4), 308–318.

He, B., Govindaraju, N. K., Luo, Q. and Smith, B. (2007), Efficient gather and scatter operations on graphics processors, *in* 'Proceedings of the 2007 ACM/IEEE conference on Supercomputing', ACM, p. 46.

He, L. and Zhang, G. (2010), Parallel branch prediction on gpu platform, *in* 'High Performance Computing and Applications', Springer, pp. 153–160.

Heineman, G. T. and Councill, W. T. (2001), 'Component-based software engineering', *Putting the pieces together, addison-westley* **5**.

Hellingman, C. (1992), 'Newton's third law revisited', *Physics Education* **27**(2), 112.

Hersch, R. D. (1989), 'Introduction to font rasterization', *André and Hersch [AH89]* pp. 1–13.

Hestness, J., Keckler, S. W. and Wood, D. A. (2014), A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior, *in* '2014 IEEE International Symposium on Workload Characterization (IISWC)', IEEE, pp. 150–160.

Hirschfeld, L. A. and Gelman, S. A., eds (1994), *Mapping the mind: domain specificity in cognition and culture*, Cambridge University Press, Cambridge ; New York.

Histibe (2016), 'How to create electronic music sample packs', `https://www.macprovideo.com/article/audio/how-to-create-electronic-music-sample-packs`. Accessed: 2022-01-31.

Hopgood, F. R. A., Hubbold, R. J. and Duce, D. (1986), *Advances in computer graphics II*, Springer Science & Business Media.

Horner, A. (1998*a*), 'Nested modulator and feedback fm matching of instrument tones', *Speech and Audio Processing, IEEE Transactions on* **6**, 398 – 409.

Horner, A. (1998*b*), 'Nested modulator and feedback fm matching of instrument tones', *IEEE Transactions on Speech and Audio Processing* **6**(4), 398–409.

Hsu, B. and Sosnick-Pérez, M. (2013), 'Realtime gpu audio: Finite difference-based sound synthesis using graphics processors', *Queue* **11**(4), 40–55.

Hutton, G. (2016), *Programming in haskell*, Cambridge University Press.

Ifeachor, E. C. and Jervis, B. W. (2002), *Digital signal processing: a practical approach*, Pearson Education.

Iwai, H. and Ohmi, S. (2002), 'Silicon integrated circuit technology from past to future', *Microelectronics Reliability* **42**(4-5), 465–491.

Iwaniec, H. and Kowalski, E. (2004), *Analytic number theory*, Vol. 53, American Mathematical Soc.

Jack, R. H., Mehrabi, A., Stockman, T. and McPherson, A. (2018), 'Action-sound latency and the perceived quality of digital musical instruments: Comparing professional percussionists and amateur musicians', *Music Perception: An Interdisciplinary Journal* **36**(1), 109–128.

Jacobs, C. T., Jammy, S. P. and Sandham, N. D. (2017), 'Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures', *Journal of Computational Science* **18**, 12–23.

John, F. (1941), 'The dirichlet problem for a hyperbolic equation', *American Journal of Mathematics* **63**(1), 141–154.

Jones, M. P. (1999), Typing haskell in haskell, *in* 'Haskell workshop', Vol. 7.

Jones, S. P., Gordon, A. and Finne, S. (1996), Concurrent haskell, *in* 'POPL', Vol. 96, pp. 295–308.

Jordà, S. (2004), 'Instruments and Players: Some Thoughts on Digital Lutherie', *Journal of New Music Research* **33**(3), 321–341.
URL: *http://www.tandfonline.com/doi/abs/10.1080/0929821042000317886*

Junger, M. C. and Feit, D. (1986), *Sound, structures, and their interaction*, Vol. 225, MIT press Cambridge, MA.

Kahane, J.-P. (1991), 'Jacques hadamard', *The Mathematical Intelligencer* **13**(1), 23–29.

Karttunen, L., Chanod, J.-P., Grefenstette, G. and Schille, A. (1996), 'Regular expressions for language engineering', *Natural Language Engineering* **2**(4), 305–328.

Ketchum, W., Amerio, S., Bastieri, D., Bauce, M., Catastini, P., Gelain, S., Hahn, K., Kim, Y., Liu, T., Lucchesi, D. et al. (2012), 'Performance study of gpus in real-time trigger applications for hep experiments', *Physics Procedia* **37**, 1965–1972.

Khokhar, A. A., Prasanna, V. K., Shaaban, M. E. and Wang, C.-L. (1993), 'Heterogeneous computing: Challenges and opportunities', *Computer* **26**(6), 18–27.

khronos (2017), 'Rendering pipeline', `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`. Accessed: 2019-08-11.

Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I. and Walton, L. (1996), A software engineering experiment in software component generation, *in* 'Proceedings of the 18th international conference on Software engineering', IEEE Computer Society, pp. 542–552.

Kilgariff, E. and Fernando, R. (2005), The geforce 6 series gpu architecture, *in* 'ACM SIGGRAPH 2005 Courses', ACM, p. 29.

Kirby, R. (2009), 'A comparison between analytic and numerical methods for modelling automotive dissipative silencers with mean flow', *Journal of Sound and Vibration* **325**(3), 565–582.

Kjolstad, F. B. and Snir, M. (2010), Ghost cell pattern, *in* 'Proceedings of the 2010 Workshop on Parallel Programming Patterns', pp. 1–9.

Kos, T., Kosar, T., Knez, J. and Mernik, M. (2010), Improving end-user productivity in measurement systems with a domain-specific (modeling) language sequencer., *in* 'ADBIS (Local Proceedings)', Citeseer, pp. 61–76.

Kramer, D., Clark, T. and Oussena, S. (2010), Mobdsl: A domain specific language for multiple mobile platform deployment, *in* '2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications', IEEE, pp. 1–7.

Kusama, Y. and Saito, K. (2021), A study on graphic input gui for em simulation, *in* '2021 International Conference on Electrical, Computer and Energy Technologies (ICECET)', IEEE, pp. 1–6.

Langtangen, H. P. (2016*a*), 'Finite difference methods for wave motion', *Department of Informatics, University of Oslo, preliminary version edition* .

Langtangen, H. P. (2016*b*), 'Solving nonlinear ode and pde problems', *Center for Biomedical Computing, Simula Research Laboratory and Department of Informatics, University of Oslo* .

Larsen, S. H. (2019), Futhark Vulkan Backend, PhD thesis, Master's thesis, University of Copenhagen, 01 2019.

Lavry, D. (2004), 'Sampling theory for digital audio', *Lavry Engineering, Inc.* . Accessed: 2022-03-15.

Leach, PGL, G. K. B. T. and Schei, C. (1993), 'The ubiquitous time-dependent simple harmonic oscillator', *South African Journal of Science* **89**(3), 126–130.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P. (2010), 'Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu', *SIGARCH Comput. Archit. News* **38**(3), 451–460.

Li, W., Sun, J. and Chen, H. (2019), 'Detecting undefined behaviors in cuda c', *IEEE Access* **7**, 182559–182572.

Liang, C.-C., Huang, C.-C. and Liou, C.-F. (2021), The impact of hardware buffer size settings on digital audio production: The model example of the avid pro tools digital audio workstation, *in* 'Smart Design, Science and Technology', CRC Press, pp. 40–43.

Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J. (2008), 'Nvidia tesla: A unified graphics and computing architecture', *IEEE micro* **28**(2), 39–55.

Lisper, B. (1996), Data parallelism and functional programming, *in* 'The Data Parallel Programming Model', Springer, pp. 220–251.

Liu, C. R., Gibbs, C. and Coady, Y. (2005), Sonar: System optimization and navigation with aspects at runtime, *in* 'Dynamic Aspects Workshop (DAW05)', p. 57.

LLC, S. (2021), 'Laplace operator', `https://www.definitions.net/definition/Laplace+operator`. Accessed: 2021-03-10.

Lomont, C. (2011), 'Introduction to intel advanced vector extensions', *Intel white paper* **23**.

Loveman, D. B. (1977), 'Program improvement by source-to-source transformation', *Journal of the ACM (JACM)* **24**(1), 121–145.

Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M. and Buck, I. (2006), Gpgpu: general-purpose computation on graphics hardware, *in* 'Proceedings of the 2006 ACM/IEEE conference on Supercomputing', ACM, p. 208.

Macià, S., Ferrer, P. J. M., Ayguadé, E. and Vicenç, B. (2022), 'Assessing saiph, a task-based dsl for high-performance computational fluid dynamics', *Available at SSRN 4220649* .

Macià, S., Mateo, S., Martínez-Ferrer, P. J., Beltran, V., Mira, D. and Ayguadé, E. (2018), Saiph: Towards a dsl for high-performance computational fluid dynamics, *in* 'Proceedings of the Real World Domain Specific Languages Workshop 2018', pp. 1–10.

MacKenzie, I. S. and Ware, C. (1993), Lag as a determinant of human performance in interactive systems, *in* 'Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems', pp. 488–493.

Mahmood, S., Lai, R., Kim, Y. S., Kim, J. H., Park, S. C. and Oh, H. S. (2005), 'A survey of component based system quality assurance and assessment', *Information and Software Technology* **47**(10), 693–707.

Marroquim, R. and Maximo, A. (2009), Introduction to gpu programming with glsl, *in* '2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing', IEEE, pp. 3–16.

Matteson, S. (2009), The acoustic simple harmonic oscillator: Experimental verification and applications, *in* 'APS Texas Sections Spring Meeting Abstracts', pp. M3–008.

McClanahan, C. (2010), 'History and evolution of gpu architecture', *A Survey Paper* **9**.

McCracken, D. D. and Reilly, E. D. (2003), Backus-naur form (bnf), *in* 'Encyclopedia of Computer Science', pp. 129–131.

McIntosh-Smith, S. and Curran, D. (2014), Evaluation of a performance portable lattice boltzmann code using opencl, *in* 'Proceedings of the International Workshop on OpenCL 2013 & 2014', pp. 1–12.

McPherson, A. (2017), 'Bela: An embedded platform for low-latency feedback control of sound', *The Journal of the Acoustical Society of America* **141**(5), 3618–3618.

Mebrate, B. (2015), 'Numerical solution of a one dimensional heat equation with dirichlet boundary conditions', *American Journal of Applied Mathematics* **3**(6), 305–311.

Meehan, M., Razzaque, S., Whitton, M. C. and Brooks, F. P. (2003), Effect of latency on presence in stressful virtual environments, *in* 'IEEE Virtual Reality, 2003. Proceedings.', IEEE, pp. 141–148.

Mei, G. and Tian, H. (2016), 'Impact of data layouts on the efficiency of gpu-accelerated idw interpolation', *SpringerPlus* **5**(1), 1–18.

Micikevicius, P. (2012), Gpu performance analysis and optimization, *in* 'GPU technology conference', Vol. 3.

Millán, E. N., Bederian, C. S., Piccoli, M. F., Garino, C. G. and Bringa, E. M. (2015), 'Performance analysis of cellular automata hpc implementations', *Computers & Electrical Engineering* **48**, 12–24.

Mitchell, J. C. and Apt, K. (2003), *Concepts in programming languages*, Cambridge University Press.

Mitchell, T. (2012), 'Automated evolutionary synthesis matching', *Soft Computing* **16**(12), 2057–2070.

Mitchell, T. J. (2010), An exploration of evolutionary computation applied to frequency modulation audio synthesis parameter optimisation, PhD thesis, University of the West of England.

Mitchell, T. J. (2020), An exploration of evolutionary computation applied to frequency modulation audio synthesis parameter optimisation, PhD thesis, The University of the West of England.

Mitchell, T. J. and Creasey, D. P. (2007), Evolutionary sound matching: A test methodology and comparative study, *in* 'Sixth International Conference on Machine Learning and Applications (ICMLA 2007)', IEEE, pp. 229–234.

Moler, C. (1986), ‘Matrix computation on distributed memory multiprocessors’, *Hypercube Multiprocessors* **86**(181-195), 31.

Moore, F. R. (1979), Signal processing requirements for computer music, *in* ‘Real-Time Signal Processing II’, Vol. 180, International Society for Optics and Photonics, pp. 33–40.

Moore, G. E. (1998), ‘Cramming more components onto integrated circuits’, *Proceedings of the IEEE* **86**(1), 82–85.

Moore, G. E. et al. (1965), ‘Cramming more components onto integrated circuits’.

Morse, P. M. and Ingard, K. U. (1986), *Theoretical acoustics*, Princeton university press.

Morse, P. M., of America, A. S. and of Physics, A. I. (1948), *Vibration and sound*, Vol. 2, McGraw-Hill New York.

Muchnick, S. et al. (1997), *Advanced compiler design implementation*, Morgan kaufmann.

Natarajan, B. (2013), ‘clfft’, `https://clmathlibraries.github.io/clFFT`. Accessed: 2019-10-21.

Nelson, R. C. (2019), ‘Context-free grammars’.

NESS (2019), ‘faustmanual’, `https://www.ness.music.ed.ac.uk/project`. Accessed: 2019-08-11.

Newton, P. and Browne, J. C. (1992), The code 2.0 graphical parallel programming language, *in* ‘Proceedings of the 6th international conference on Supercomputing’, pp. 167–177.

Nickolls, J. and Dally, W. J. (2010), ‘The gpu computing era’, *IEEE micro* **30**(2), 56–69.

Nurseitov, N., Paulson, M., Reynolds, R. and Izurieta, C. (2009), ‘Comparison of json and xml data interchange formats: a case study.’, *Caine* **9**, 157–162.

O’Haver, T. (1997), ‘A pragmatic introduction to signal processing’, *University of Maryland at College Park* .

Olsen (2018), ‘Functional programming in 40 minutes’, `https://www.youtube.com/watch?v=0if71HOyVjY`.

Onofrei, M. G., Willemsen, S. and Serafin, S. (2021), 'Real-time implementation of a friction drum inspired instrument using finite difference schemes'.

Orfanidis, S. J. (1995), *Introduction to signal processing*, Prentice-Hall, Inc.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. and Purcell, T. J. (2007), A survey of general-purpose computation on graphics hardware, *in* 'Computer graphics forum', Vol. 26, Wiley Online Library, pp. 80–113.

Patel, J. K. and Read, C. B. (1996), *Handbook of the normal distribution*, Vol. 150, CRC Press.

Peetre, J. (2000), 'On fourier's discovery of fourier series and fourier integrals', *preprint* .

Pierce, B. C. (2002), *Types and programming languages*, MIT press.

Pohlmann, K. C. (2000), *Principles of digital audio*, Vol. 4, McGraw-Hill New York.

Pospichal, P. and Jaros, J. (2009), 'Gpu-based acceleration of the genetic algorithm', *GECCO competition* .

Raghunath, K. J. and Rambaud, M. M. (1999), 'Sine/cosine lookup table'. US Patent 5,937,438.

Raman, S. K., Pentkovski, V. and Keshava, J. (2000), 'Implementing streaming simd extensions on the pentium iii processor', *IEEE micro* **20**(4), 47–57.

Ramm, R. (2021), 'What is a good buffer size for recording?', https://www.orpheusaudioacademy.com/buffer-size/#:~:text=A%20good%20buffer%20size%20for%20recording%20is%20128%20samples%2C%20but,not%20run%20in%20real%20time. Accessed: 2022-12-07.

Rechenberg, I. (1965), 'Cybernetic solution path of an experimental problem', *Royal Aircraft Establishment Library Translation 1122* .

Reinders, J. (2007), 'Understanding task and data parallelism | zdnet', https://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129. Accessed: 2019-08-11.

Renardy, M. and Rogers, R. C. (2006), *An introduction to partial differential equations*, Vol. 13, Springer Science & Business Media.

Renney, H., Gaster, B. and Mitchell, T. J. (2022), 'Survival of the synthesis—gpu accelerating evolutionary sound matching', *Concurrency and Computation: Practice and Experience* p. e6824.

Renney, H., Gaster, B. R. and Mitchell, T. (2019), Opencl vs: Accelerated finite-difference digital synthesis, *in* 'Proceedings of the International Workshop on OpenCL', pp. 1–11.

Renney, H., Gaster, B. R. and Mitchell, T. J. (2020), 'There and back again: The practicality of gpu accelerated digital audio'.

Renney, H., Willemsen, S., Gaster, B. R. and Mitchell, T. J. (2022), 'Hypermodels - a framework for gpu accelerated physical modelling sound synthesis'.

Repp, B. H. and Su, Y.-H. (2013), 'Sensorimotor synchronization: a review of recent research (2006–2012)', *Psychonomic bulletin & review* **20**(3), 403–452.

Richter, C., Schöps, S. and Clemens, M. (2013), 'Gpu acceleration of finite difference schemes used in coupled electromagnetic/thermal field simulations', *IEEE transactions on magnetics* **49**(5), 1649–1652.

Risset, J.-C. (1965), 'Computer study of trumpet tones', *The Journal of the Acoustical Society of America* **38**(5), 912–912.

Rotem, N. and Ben Asher, Y. (2014), 'Block unification if-conversion for high performance architectures', *IEEE Computer Architecture Letters* **13**(1), 17–20.

Roth, M. and Yee-King, M. (2011), A comparison of parametric optimization techniques for musical instrument tone matching, *in* 'Audio Engineering Society Convention 130', Audio Engineering Society.

Russo, R., Serafin, S., Michon, R., Orlarey, Y. and Letz, S. (2021), Introducing finite difference schemes synthesis in faust: A cellular automata approach, *in* 'Proceedings of the 18th Sound and Music Computing Conference'.

Ryan, E. M., Tartakovsky, A. M. and Amon, C. (2010), 'A novel method for modeling neumann and robin boundary conditions in smoothed particle hydrodynamics', *Computer Physics Communications* **181**(12), 2008–2023.

Sakakibara, Y. (1992), 'Efficient learning of context-free grammars from positive structural examples', *Information and Computation* **97**(1), 23–60.

Sanders, J. and Kandrot, E. (2010), *CUDA by example: an introduction to general-purpose GPU programming*, Addison-Wesley Professional.

Sandgren, J. (2013), 'Transfer time reduction of data transfers between cpu and gpu'.

Sasajima, M., Yamaguchi, T. and Hara, A. (2010), Acoustic analysis using finite element method considering effects of damping caused by air viscosity in audio equipment, *in* 'Applied Mechanics and Materials', Vol. 36, Trans Tech Publ, pp. 282–286.

Scherer, R. C., Torkaman, S., Kucinschi, B. R. and Afjeh, A. A. (2010), 'Intraglottal pressures in a three-dimensional model with a non-rectangular glottal shape', *The Journal of the Acoustical Society of America* **128**(2), 828–838.

Schloss, J. (2019), 'cooley_tukey', https://www.algorithm-archive.org/contents/cooley_tukey/cooley_tukey.html. Accessed: 2019-10-10.

Schordan, M. and Quinlan, D. (2003), A source-to-source architecture for user-defined optimizations, *in* 'Joint Modular Languages Conference', Springer, pp. 214–223.

Sellers, G., Wright Jr, R. S. and Haemel, N. (2013), *OpenGL superBible: comprehensive tutorial and reference*, Addison-Wesley.

Serra, E. and Bonaldi, M. (2009), 'A finite element formulation for thermoelastic damping analysis', *International Journal for Numerical Methods in Engineering* **78**(6), 671–691.

Shampine, L. F. (2018), *Numerical solution of ordinary differential equations*, Routledge.

Shneiderman, B. (1997), Direct manipulation for comprehensible, predictable and controllable user interfaces, *in* 'Proceedings of the 2nd international conference on Intelligent user interfaces', pp. 33–39.

Shreiner, D., Group, B. T. K. O. A. W. et al. (2009), *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*, Pearson Education.

Sizikov, V. S. et al. (2011), *Well-posed, ill-posed, and intermediate problems with applications*, Vol. 49, Walter de Gruyter.

Skare, T. and Abel, J. (2019), Gpu-accelerated modal processors and digital waveguides, *in* 'Linux Audio Conference'.

Slotnick, D. L., Borck, W. C. and McReynolds, R. C. (1962), The solomon computer, *in* 'Proceedings of the December 4-6, 1962, fall joint computer conference', pp. 97–107.

Smith III, J. O. (2010), 'Audio signal processing in faust', *online tutorial: https://ccrma. stanford. edu/jos/aspf* .

Smith, J. (1997), 'Acoustic modeling using digital waveguides', *Musical Signal Processing* **7**, 221–264.

Smith, J. O. (2007), *Introduction to digital filters: with audio applications*, Vol. 2, Julius Smith.

Sosnick, M. and Hsu, W. (2010), Efficient finite difference-based sound synthesis using gpus, *in* 'Proceedings of the Sound and Music Computing Conference', pp. 42–44.

Sun, Y., Gong, X., Ziabari, A. K., Yu, L., Li, X., Mukherjee, S., McCardwell, C., Villegas, A. and Kaeli, D. (2016), Hetero-mark, a benchmark suite for cpu-gpu collaborative computing, *in* '2016 IEEE International Symposium on Workload Characterization (IISWC)', IEEE, pp. 1–10.

Sutter, H. (2005), 'The free lunch is over: A fundamental turn toward concurrency in software', *Dr. Dobb's journal* **30**(3), 202–210.

Sweetcare (2022), 'Which buffer size setting should i use in my daw?', `https://www.sweetwater.com/sweetcare/articles/which-buffer-size-setting-should-i-use-in-my-daw/#:~:text=The%20most%20common%20buffer%20size,of%20memory%20and%20processing%20power.` Accessed: 2022-12-07.

Terzo, O., Djemame, K., Scionti, A. and Pezuela, C. (2019), *Heterogeneous Computing Architectures: Challenges and Vision*, CRC Press.

Thibault, A. (2019), Wind Instrument Sound Synthesis through Physical Modeling, PhD thesis, ENS Paris-Ecole Normale Supérieure de Paris; Sorbonne Université; Inria ....

Thomas, J. (1995), 'Partial differential equations, finite difference methods'.

Thompson, D. L. (2000), 'Power consumption reduction in medical devices employing multiple digital signal processors'. US Patent 6,023,641.

Tompson, J. and Schlachter, K. (2012), 'An introduction to the opencl programming model', *Person Education* **49**, 31.

Trevett, N. (2013), 'Opencl introduction', *Khronos Group* .

Tsoy, Y. R. (2003), The influence of population size and search time limit on genetic algorithm, *in* '7th Korea-Russia International Symposium on Science and Technology, Proceedings KORUS 2003.(IEEE Cat. No. 03EX737)', Vol. 3, IEEE, pp. 181–187.

Ullrich, S. and de Moura, L. (2019), Counting immutable beans: Reference counting optimized for purely functional programming, *in* 'Proceedings of the 31st Symposium on Implementation and Application of Functional Languages', pp. 1–12.

Visser, E. (2007), Webdsl: A case study in domain-specific language engineering, *in* 'International summer school on generative and transformational techniques in software engineering', Springer, pp. 291–373.

Walinsky, C. and Banerjee, D. (1990), A functional programming language compiler for massively parallel computers, *in* 'Proceedings of the 1990 ACM conference on LISP and functional programming', ACM, pp. 131–138.

Wang, J. C. (1984), 'Young's modulus of porous materials', *Journal of materials science* **19**(3), 801–808.

Webb, C. J. and Bilbao, S. (2015), On the limits of real-time physical modelling synthesis with a modular environment, *in* 'Proceedings of the International Conference on Digital Audio Effects', p. 65.

Weber, T. (2014), Micropolygon Rendering on the GPU, PhD thesis.

Wei, Z., Jang, B. and Jia, Y. (2014), 'A fast and interactive heat conduction simulator on gpus', *Journal of Computational and Applied Mathematics* **270**, 496–505.

Weik, M. (2012), *Communications standard dictionary*, Springer Science & Business Media.

Wen-mei, W. H. (2015), *Heterogeneous System Architecture: A new compute platform infrastructure*, Morgan Kaufmann.

Willemsen, S. (2021), The Emulated Ensemble: Real-Time Simulation of Musical Instruments using Finite-Difference Time-Domain Methods, PhD thesis, Aalborg University Copenhagen.

Willemsen, S., Andersson, N., Serafin, S. and Bilbao, S. (2019*a*), 'Real-time control of large-scale modular physical models using the sensel morph'.

Willemsen, S., Andersson, N., Serafin, S. and Bilbao, S. (2019*b*), Realtime control of large-scale modular physical models using the sensel morph, *in* 'Proc. of the 16th Sound and Music Computing Conference', pp. 275–280.

Willemsen, S., Bilbao, S., Ducceschi, M. and Serafin, S. (2021), 'A physical model of the trombone using dynamic grids for finite-difference schemes'.

Willemsen, S., Bilbao, S. and Serafin, S. (2019), Real-time implementation of an elasto-plastic friction model applied to stiff strings using finite difference schemes, *in* '22nd International Conference on Digital Audio Effects'.

Willemsen, S., Serafin, S., Bilbao, S. and Ducceschi, M. (2020), Real-time implementation of a physical model of the tromba marina, *in* '17th Sound and Music Computing Conference', pp. 161–168.

Willemsen, S., Serafin, S. and Jensen, J. R. (2017), Virtual analog simulation and extensions of plate reverberation, *in* '14th Sound and Music Computing Conference'.

Wong, B. (2011), 'Color blindness', *nature methods* **8**(6), 441.

Wu, W.-H. and Chen, C.-Y. (2001), 'Simple lumped-parameter models of foundation using mass-spring-dashpot oscillators', *Journal of the Chinese Institute of Engineers* **24**(6), 681–697.

Yang, C., Kurth, T. and Williams, S. (2020), 'Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system', *Concurrency and Computation: Practice and Experience* **32**(20), e5547.

Yee-King, M. and Roth, M. (2011), A comparison of parametric optimisation techniques for musical instrument tone matching, *in* 'Audio Engineering Society Convention 130', Goldsmiths, University of London.

Yu, W., Mittra, R., Yang, X. and Liu, Y. (2009), Performance analysis of parallel fdtd algorithm on different hardware platforms, *in* '2009 IEEE Antennas and Propagation Society International Symposium', IEEE, pp. 1–4.

Zakai, A. (2011), Emscripten: an llvm-to-javascript compiler, *in* 'Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion', pp. 301–312.

zappi (2019), ‘Drumhead synthesizer’, `https://www.youtube.com/watch?v=_qLwJC4PYR0`. Accessed: 2019-08-11.

Zappi, V. (2017), *The Hyper Drumhead: Making Music with a Massive Real-time Physical Model*, Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

Zappi, V., Allen, A. and Fels, S. (2017), Shader-based physical modelling for the design of massive digital musical instruments, *in* ‘Proceedings of the International Conference on New Interfaces for Musical Expression’, p. 145.

Zubair, M., Mughal, M. J. and Naqvi, Q. (2011), ‘An exact solution of the cylindrical wave equation for electromagnetic field in fractional dimensional space’, *Progress In Electromagnetics Research* **114**, 443–455.

Zwillinger, D. (1998), *Handbook of differential equations*, Vol. 1, Gulf Professional Publishing.

Glossary

benchmarking Benchmarking is the comprehensive evaluation of software and hardware. Involves measuring the run-time performance and resources used in a series of benchmarking micro & macro tests. See Section 3.1. Pages. 35

CPU The Central Processing Unit is a processor typically used as the primary proccessing unit by computer systems. See Section 2.3.1.1. Pages. 33–35, 41

DSL Domain-specific languages are languages specialized to operate within a particular domain. DSLs like Faust contrast General-purpose languages like C++. See section 2.4.5. Pages. 48

DSP Digital Signal Processing is the application of digital computer systems for singal processing. Where signals are a continuous sequence of values representing some quantifiable phenomena or entity. See Section 2.1.1. Pages. 14, 15, 54

ES Evolution strategies (ES) are a type of evolutionary algorithm that is used for optimisation problems, where a suitable solution needs to be found within a search space of numerous possible solutions. Pages. 74

GPGPU General-purpose GPU computing is the idea of using GPUs for general computation of other problems, outside of the original graphics domain. See Section 2.3. Pages. 9

GPU A Graphics Processing Unit is a special device originally designed for massively parallel processing of computer graphics for display devices. Modern GPUs also fully support the use of the GPU for general-purpose computing. See Section 2.3.3. Pages. 8, 9, 34, 35, 41

Haskell Haskell is a functional programming language based on lambda calculus. Haskell has the following properties: polymorphically statically typed, lazy, purely functional. Pages. 43

HyperModels HyperModels is a novel framework for facilitating the development of GPU accelerated physical model synthesisers presented in this thesis. See Section 7. Pages. 1, 4, 11, 12, 44, 52, 110, 116, 120–123, 125, 127–131, 133, 134, 136, 142, 143, 154, 157–159, 161, 162, 164–168

OpenCL The Open Computing Language is an open-source, cross-platform, heterogeneous programming framework. Pages. 9, 34

SIMD Single-Instruction Multiple-Data (SIMD) processing paradigm applies same sets of instructions across multiple data elements in parallel. Pages. 97

SIMT Single-Instruction Multiple-Threading, extends the SIMD paradigm by enabling different sets of instructions to be processed simultaneously across threads. Pages. 98

There and Back Again: The Practicality of GPU Accelerated Digital Audio

Harri Renney
Com Sci Research Centre
University of West of England
Bristol, UK
harri.renney@uwe.ac.uk

Benedict R. Gaster
Com Sci Research Centre
University of West of England
Bristol, UK
benedict.gaster@uwe.ac.uk

Thomas J. Mitchell
Creative Technology Lab
University of West of England
Bristol, UK
tom.mitchell@uwe.ac.uk

ABSTRACT

General-Purpose GPU computing is becoming an increasingly viable option for acceleration, including in the audio domain. Although it can improve performance, the intrinsic nature of a device like the GPU involves data transfers and execution commands which requires time to complete. Therefore, there is an understandable caution concerning the overhead involved with using the GPU for audio computation. This paper aims to clarify the limitations by presenting a performance benchmarking suite. The benchmarks utilize OpenCL and CUDA across various tests to highlight the considerations and limitations of processing audio in the GPU environment. The benchmarking suite has been used to gather a collection of results across various hardware. Salient results have been reviewed in order to highlight the benefits and limitations of the GPU for digital audio. The results in this work show that the minimal GPU overhead fits into the real-time audio requirements provided the buffer size is selected carefully. The baseline overhead is shown to be roughly $0.1ms$, depending on the GPU. This means buffer sizes 8 and above are completed within the allocated time frame. Results from more demanding tests, involving physical modelling synthesis, demonstrated a balance was needed between meeting the sample rate and keeping within limits for latency and jitter. Buffer sizes from 1 to 16 failed to sustain the sample rate whilst buffer sizes 512 to 32768 exceeded either latency or jitter limits. Buffer sizes in between these ranges, such as 256, satisfied the sample rate, latency and jitter requirements chosen for this paper.

Author Keywords

NIME, DMI, GPGPU, HPC

CCS Concepts

• **Computing methodologies → Graphics processors;**

1. INTRODUCTION

General-purpose GPU (GPGPU) computing provides the capacity for massively parallel processing using a widely available hardware accelerator: the graphics processing unit (GPU). There are many digital audio processes that are suitable for the GPGPU environment and can result in a substantial performance increase, relative to a CPU bound program. Examples of academic work exploring the use of GPUs in digital audio with notable results can be seen in [17] and [1]. However, the communication overhead imposed when using a hardware accelerator, like the GPU, can outweigh the benefits. This is especially relevant for applications with real-time requirements, such as digital audio processing. There seems to be an understandable caution within the digital audio developer community surrounding GPGPU, and consequently GPGPU optimisation is sparsely used. But what are the practical limitations of audio processing on the GPU? This paper aims to investigate the performance overhead of GPGPU within the audio domain, by measuring the communication overhead between the CPU and GPU in both unidirectional and bidirectional cases. In particular, the contributions are:

- An open-source benchmarking suite for evaluating GPU computation within a digital audio domain;
- Results from the benchmarking suite across various hardware systems; and
- A summary of the salient findings.

1.1 Digital Audio

Digital audio is the representation of acoustic sound in a discretized and quantized form in order for it to be computable [2]. An originally continuous audio signal must be broken up into a finite set of samples representing the signal (quantization), where each sample is represented with finite precision (discretization). When signals are represented in this form, computer systems can process existing or synthesise new signals.

The most convenient, and abundant way to process digital audio is to program tools and software that runs on the central processing unit (CPU) in a language such as C++. The CPU has limitations as a powerful, but coarse-grained parallel processor, as shown when compared with other processors by Mistry et al. [9]. Highly regarded experts predict future computational growth will come from utilizing parallel architectures and heterogeneous computing [16]. As a result, processor design is undergoing significant changes. As the state of the art hardware develops, software must map appropriately. There is a variety of fundamentally different processor types that can be used in digital audio processing. These include digital signal processors (DSP), field-programmable gate arrays (FPGA) and graphics processing units (GPU). In this work, we explore the practicalities of using the GPU as a device for offloading suitable tasks in digital audio.

1.2 General-Purpose GPU Computing

GPGPU computing is the use of the GPU, originally intended for rendering graphics, for general computation [8]. For a long time, GPU architectures combined with the available software APIs, required general compute problems to

NIME'20, July 21-25, 2020, Royal Birmingham Conservatoire, Birmingham City University, Birmingham, UK.

Requirement	Recommended	Limit
Sample Rate	96000	44100
Latency	10ms	20ms
Jitter	±1ms	±3ms

Table 1: Real-time audio requirements.

be mapped into the graphics domain. This non-trivial mapping was often not practical and was mostly only pursued by academics. Over time, GPGPU software standards and APIs were proposed and have been developed with great success. Owens et al. [12] cover the development of GPGPU from its origins where Mark Harris first coined the term GPGPU, to the mature development environments available today that are under continuous development in both industry and academia.

1.3 Real-time Processing

Real-time processing is the requirement for a program to meet a set of performance requirements consistently for a particular application. Requirements vary between applications, where variable amounts of data need to be processed within a fixed and inflexible time frame. In the case of real-time audio, a consistent number of audio samples needs to be produced every second. The real-time requirement in audio is very strict, as even a few missed samples or delays results in instantly noticeable 'glitches'. Latency is a term used in various fields to describe the delay between the initiation of an event to its conclusion. Within the context of this work, we focus on the time taken for a buffer of audio samples to be dispatched and returned from the CPU to the GPU. Although this avoids necessary stages in digital audio, like the operating system and sound devices, restricting this measurement isolates the GPU overhead specifically. If the latency is too high in a real-time, interactive application, it will often detriment performance. The other impacting factor for real-time audio is known as jitter. In this work, jitter will refer to the variation in latency between consecutive buffers dispatched to the GPU.

The standard requirements surrounding real-time audio has long been debated. In this paper, the real-time requirements discussed in [7] and [6] are used. These are shown in Table 1.

2. TECHNIQUES

In this section, the general techniques used in the design of GPGPU applications are briefly covered. These are important concepts or optimizations which help to maximize the benefits of using the GPU.

2.1 Buffering

Buffering is an important technique used to reduce the communication overhead considerably between the CPU and GPU. It works by requesting the GPU to execute and generate a variable sized buffer of audio samples each time, rather than a single sample. Figure 1 visualizes buffer transfers to and from the GPU for computation.

The ideal buffer size for GPU dispatch is an extremely important factor in the performance within real-time applications. This paper aims to explore the limits for the range of buffer sizes that work within the constraints of real-time audio.

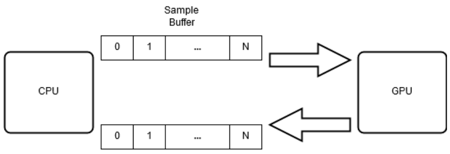


Figure 1: Buffers of N samples are dispatched to GPU and returned to CPU.

2.2 Unified Memory

Discrete GPUs are independent devices typically included in a system as an extra peripheral. Therefore, discrete GPUs have their own local memory and communicate with the CPU over system buses, shown in Figure 2. System buses take time to transfer data and introduce unavoidable latency, irrespective of the data size.

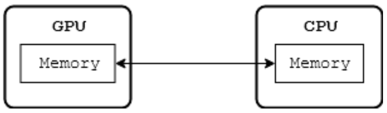


Figure 2: CPU to GPU memory accesses across system bus into respective device's memory.

Unified memory is a single addressable memory space accessible by separate processors, shown in Figure 3. This means the overhead of transferring over the system bus is avoided. Integrated GPUs take advantage of this technology and share a memory space with the CPU. This allows fast memory transfers between CPU and integrated GPU by default, which is an important perspective to consider for this paper.

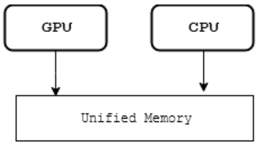


Figure 3: CPU and GPU memory accesses using shared, unified memory.

2.3 Pinned Memory

Pinned memory, or page-locked memory is a special type of memory that is located within the CPU host memory and GPU memory at the same time. Changes in the host memory can take effect in the GPU memory which can be accessed by the GPU for computation. In essence, pinning memory removes any extra step in copying data between CPU and GPU. Although this memory is typically faster to use, it is a limited resources. ([10] - 3.2.4) Therefore, it is advised to use for small amounts of data that need to be frequently transferred between CPU and GPU.

3. IMPLEMENTATIONS

This paper explores the different methods of utilizing the GPU within application software. Performance is heavily dependant on the software API used. OpenCL and CUDA are two of the most widely used methods for GPGPU. All performance tests in this paper will be implemented using both methods for comparison.

3.1 OpenCL

The Open Computing Language (OpenCL) [4] is an open-standard, cross-platform, heterogeneous programming framework. It provides a single abstract programming model that developers adhere to. Hardware vendors that support OpenCL translate the abstract model to match the particular architecture of their devices. OpenCL currently has support ranging from FPGA, DSP and, of interest to this study, GPUs. OpenCL version 1.2 was used across the benchmarking results collected here, as the lowest denominator supported by NVIDIA, AMD and Intel.

3.2 CUDA

CUDA [14] is NVIDIA's propriety, parallel computing platform for supporting their own GPUs for general compute. CUDA is widely used, including in research and academic studies. In research applications, it can be acceptable to constrain to specific hardware, as is the case with CUDA. However, in industry and commercial environments, this becomes more problematic, as applications implemented using CUDA would not be compatible with machines that have AMD GPUs.

4. BENCHMARK METHODOLOGY

In this section, the benchmarking methodology and tests are described. The system specifications are outlined briefly. The general performance benchmarks are listed, which expose results for GPGPU tests in general. After these, the real-time digital audio tests are defined. These tests aim to identify the limits on the buffer size for example cases, that are representative of typical real-time audio processing scenarios. Harris in [5] explains how to accurately benchmark CUDA applications, OpenCL specific profiling is covered in [15]. The methodology followed here uses CPU timers for measuring overall times and vendor specific profiling tools for measuring isolated parts of the process. The benchmarking suite is open source and available at `https://github.com/Harri-Renney/ThereAndBackAgain-NIME`

4.1 System Specifications

The specifications for the system on which the benchmarking has been performed for this study is shown in Table 2. Various systems, with hardware from different vendors have been chosen in order to observe the performance in general. The systems include discrete GPUs from AMD and NVIDIA, along with an integrated Intel GPU. Integrated GPUs are closely coupled to the CPU and usually have very fast data transfers between them, using faster memory buses and unified memory space. Discrete GPUs typically communicate over slower memory buses across the motherboard and therefore have a larger initial overhead. However, the trade off is that discrete GPUs are usually much more powerful than integrated GPUs.

4.2 Test Format

The general format of the benchmark tests follows the pseudo code below.

```
void test() {
    prepareTest(hostVariables, deviceVariables);

    if(isWarmup) {
        runTest();
    }
    for(int i = 0; i != numRepeats; ++i) {
        startTime = timestamp();
        runTest();
        endTime = timestamp();
    }
    checkTestResults(testResults);
    cleanup(hostVariables, deviceVariables);
}
```

To begin each test, all preparations and initializing of host and device variables are made. Further, kernel code is prepared if necessary. Both CUDA and OpenCL take considerably longer running kernels and data transfers for the first time. This is as preparations and optimizations are made to increase performance of subsequent execution. For this reason, a warm-up variable has been added to the benchmarks controlling whether a warm-up run executes before profiling starts. The test runs begin by timestamping either side of the test. When the tests are complete, the results are checked to ensure the processing done by the GPU is correct. Using the GPU requires memory allocations which the programmer must manually manage. So to finish the test, all associated memory is deallocated.

4.3 Microbenchmarks

This section lists the microbenchmarking tests covered in this paper. The suite includes further tests that are not described here; we intend to explore these in future work.

- *null kernel* - A minimal test to measure the threshold overhead to execute an empty program on the GPU.
- *cpu to gpu to cpu* - A bidirectional test measuring the round-trip transfer time between CPU and GPU.
- *complex buffer processing* - Applies a triangular smoothing operation ([11] - P.g 34. Smoothing) to the input signal and returns 'smoothed' buffers to CPU. Involves bidirectional transfers and multiple memory accesses in the kernel.
- *simple buffer synthesis* - Generation of a sinusoidal signal at a given frequency, generating sine values that fill the buffer length in parallel. This operation involves only unidirectional memory transfers from the GPU to the CPU, returning synthesised sample buffers.
- *complex buffer synthesis* - The complex buffer synthesis is an application that has a challenging amount of computation, bidirectional CPU-GPU transfers and involves memory management. An application which meets these requirements is a finite-difference time-domain physical model synthesizer [18]. For the full details of the design and implementation, see [13].

Each of the tests are implemented in OpenCL and CUDA. Furthermore, each test is implemented using standard memory buffers and pinned memory. All tests have their total times and specific details measured over 10,000 repetitions. From these repeated results, the average, minimum and maximum buffer times are calculated, along with the maximum and average jitter. The results of the warm-up runs have been recorded and are available in the results database.

4.4 Real-time Digital Audio Tests

The audio buffer size is an extremely important factor for achieving real-time performance with GPU acceleration. This section outlines the real-time tests that aim to identify the limits for the buffer size in the unidirectional and bidirectional cases.

To measure real-time performance, the following limits (Described in Section 1.3) must be satisfied:

1. The maximum acceptable latency for each buffer will be 20ms, though the recommended 10ms is preferred.
2. The target sample rate of 44.1KHz should be satisfied within the other limits, though the recommended 96KHz is preferred.
3. The deviation, or jitter, between each buffer generated should not be greater than $\pm 3ms$, though the recommended range $\pm 1ms$ is preferred.

An enumeration of buffer sizes will be applied in each test to find an approximate range of values the buffer size can comfortably operate in. These tests will be conducted for

Specification	Mid-range Laptop	High-end AMD	High-end NVIDIA GeForce	High-End NVIDIA Titan
CPU	Intel Core i7-8550U	Intel Core it-9800X	Intel Core it-9800X	Intel Core it-9800X
Integrated GPU	Intel UHD Graphics 620	None	None	None
Discrete GPU	AMD Radeon 530	Radeon Pro WX 7100	GeForce RTX 2080 Ti	Titan RTX
CPU RAM	8GB	32GB	32GB	32GB

Table 2: Hardware specification used for benchmarking

unidirectional and bidirectional cases. The tables 3 and 4 in the results Section 5 highlights values in green if in the recommended limit, orange if in the maximum limits and red if outside of the limits.

4.4.1 Total Time

Here is proposed an equation for total execution time in a GPGPU environment to formalise the overhead and limitations:

$$t_{total}(x) = t_{tran}(x) + c(x) + g(x) \tag{1}$$

Where:

- $t\_total(x)$ = The total execution time of x samples for a GPGPU application.
- $t\_tran(x)$ = The transfer time for x samples between CPU & GPU.
- $c(x)$ = The function of processing executed on the CPU.
- $g(x)$ = The function of processing executed on the GPU.
- x = The number of samples in the buffer/vector to be processed.

4.4.2 Baseline Limits

The baseline limits will be the time for the minimum transfer and null kernel execution for different buffer sizes. From here, a limit involving variable computation can be derived. Taking Equation 1 and assuming $c()$ or $g()$ are negligible, the baseline overhead is defined as:

$$t_{total}(x) = t_{trans}(x) \tag{2}$$

4.4.3 Kernel Computation Limits

Once the baseline limit has been measured, the kernel computation time can be calculated. By subtracting the baseline $t_{trans}(x)$ from the full Equation 1, the total computation time $t_{comp}(x)$ remains, see Equation 3. If the CPU compute time $c(x)$ is not considered, just the GPU compute time $g(x)$ remains. By considering this Equation 3 and the baseline overhead Equation 2, an understanding of GPGPU becomes more clear. The baseline overhead serves as an initial cost to be considered. From here, the computation cost involved can be increased within the limits of the application.

$$t_{comp}(x) = c(x) + g(x) = t_{total}(x) - t_{trans}(x) \tag{3}$$

5. RESULTS

As is to be expected, due to all the permutations of configurations, a lot of results have been collected. In this section, we analyse particular highlights of the results, in areas considered most relevant within the audio domain. A collection of all results for those interested can be found at: `https://muses-dmi.github.io/benchmarking/benchmarking_database_there_and_back_again`.

The results considered here have been collected following a conservative approach. This means after every API call to the GPU, an explicit synchronization is made between the CPU and GPU. Further, pinned memory in OpenCL is mapped and unmapped to ensure it is defined even though on many systems this is not necessary. This is important to keep in mind and trivial modifications to the tests would improve performance further. By taking this approach, more confidence can be given to the results knowing that they can be improved.

	GeForce2080_cl			GeForce2080_cuda		
Buffer Length	Total Time	Average Latency	Max Jitter	Total Time	Average Latency	Max Jitter
1	6133.319	0.139	0.741	5676.300	0.128	1.032
2	3053.499	0.138	0.796	2838.170	0.128	0.725
4	1518.143	0.137	0.336	1412.804	0.128	0.669
8	751.078	0.136	0.166	708.566	0.128	0.632
16	378.847	0.137	0.156	378.930	0.137	0.743
32	190.267	0.1375	0.199	183.560	0.133	0.622
64	96.077	0.139	0.170	95.619	0.138	0.646
128	48.746	0.141	0.183	51.526	0.149	0.612
256	24.731	0.142	0.217	27.137	0.156	0.337
512	12.663	0.145	0.192	29.270	0.336	0.474
1024	6.348	0.144	0.164	16.507	0.375	0.490
2048	3.110	0.141	0.152	7.866	0.357	0.340
4096	1.606	0.146	0.166	3.816	0.346	0.496
8192	0.907	0.151	0.158	1.375	0.229	0.251
16384	0.498	0.166	0.160	0.706	0.235	0.264
32768	0.368	0.184	0.186	0.508	0.254	0.252

Table 3: Baseline bidirectional real-time test.

5.1 Minimum GPU Overhead

The minimum GPU overhead involved for the different buffer sizes is a key factor for many applications. If the overhead alone exceeds the requirements, then the GPU will not be appropriate for the task. This means the minimum overhead is a good foundational position to start. The results for executing the null kernel test were $0.002051ms$ on the Radeon 530, $0.000455ms$ on the UHD, $0.009392ms$ on Geforce 2080, $0.011468ms$ on Titan. These are impressively small times, but only execute empty kernels, avoiding critical stages transferring data and processing or synthesis. With the bare minimum results established, the bidirectional baseline test which involves round-trip memory transfers and execution of the null kernel is shown in Table 3. The tests operates at a sample rate of 44.1KHz [1] with measurements taken to calculate the total time to process 44100 samples and the latency and jitter per buffer. These results show that even for a round-trip data transfer with no processing, certain smaller buffer ranges are not practical. Buffer sizes 1, 2 and 4 all have total times above a second for 44100 samples. Therefore, for applications that require single or very smaller buffer sizes, discrete GPUs will not perform sufficiently and the CPU would be the better option. Using Equation 2, the transfer time can be used to demonstrate the minimum GPU overhead at each buffer size. For example, a buffer of 128 is $t_{total}(128) = t_{trans}(128) = 0.141295ms$

5.2 Standard vs Pinned

Figure 4 plots the round-trip data transfer from CPU to GPU and back to the CPU for various buffer lengths in OpenCL, with no kernel executed on the GPU. The solid coloured lines indicate the standard buffer allocation and transfer approach, while the dashed coloured lines represent the pinned buffer memory approach. For all the discrete GPUs, it seems that the pinned memory approach performs better in this test. The smallest difference seen for the GeForce2080 is still ±0.04 and largest for the Radeon530 is

[1] Results for higher sample rates of 48KHz and 96KHz can be found in the results database.

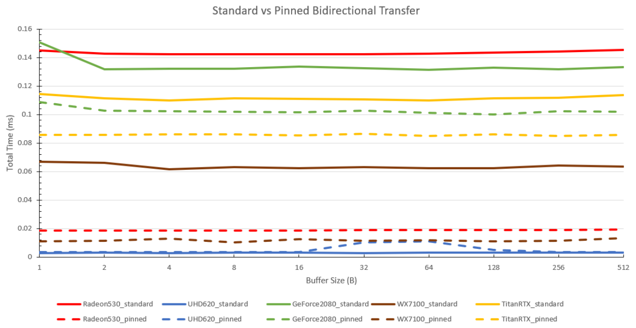


Figure 4: Standard vs pinned bidirectional memory transfers.

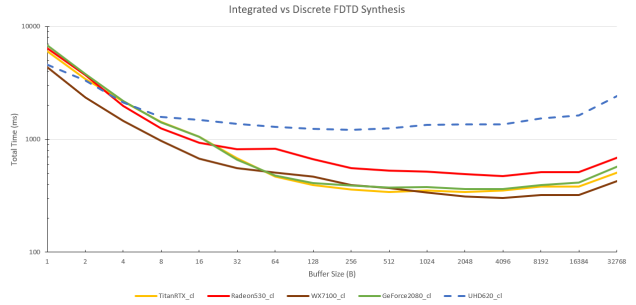


Figure 5: Integrated vs Discrete in bidirectional synthesis.

±0.12. These are significant differences, though it is important to consider no processing is involved in this test and therefore the performance implications during computation as a result of memory choice is avoided. For the integrated Intel GPU, using the standard or pinned approach does not impact performance. This is because the integrated GPU shares its unified memory space with the CPU anyway. OpenCL and CUDA possibly default either approach to the same unified memory approach instead.

	GeForce2080			Radeon7100		
Buffer Length	Total Time	Average Latency	Max Jitter	Total Time	Average Latency	Max Jitter
1	6802.339	0.154	0.958	4372.550	0.099	0.469
2	3790.166	0.171	0.285	2339.507	0.106	0.476
4	2186.325	0.198	0.319	1459.335	0.132	0.387
8	1416.375	0.256	0.321	966.981	0.175	0.546
16	1049.813	0.380	0.395	674.703	0.244	0.617
32	659.516	0.478	0.757	553.152	0.401	0.858
64	478.657	0.693	0.899	508.541	0.737	1.114
128	410.979	1.191	1.178	465.922	1.350	1.544
256	389.878	2.253	2.410	392.696	2.269	2.572
512	373.748	4.295	6.064	370.717	4.261	4.526
1024	377.182	8.572	12.736	338.996	7.704	7.995
2048	363.365	16.516	16.854	309.835	14.083	14.946
4096	361.906	32.900	33.704	302.421	27.492	28.727
8192	393.858	65.643	66.226	322.224	53.704	54.844
16384	414.806	138.268	141.349	320.432	106.810	107.738
32768	570.743	285.371	286.467	427.697	213.848	215.347

Table 4: Physical model synthesizer bidirectional real-time test.

5.3 Integrated vs Discrete

One of the biggest influences on the GPU overhead involved is the type of hardware used. Here, the performance of integrated and discrete GPUs are examined, highlighting where each type of device performs better.

In Figure 5, the bidirectional physical model synthesis is plotted. All the discrete graphics cards have been displayed as solid coloured lines, the integrated graphics card tested is a dashed coloured line. The total time to compute the 44.1KHz of samples has been shown on a logarithmic scale to emphasize the subtle differences. For small buffer lengths, the overhead experienced by most of the discrete GPUs heavily outweighs the benefits and the integrated GPU performs better. However, once larger buffer lengths are used and the transfer overhead reduced considerably, the discrete GPUs take the lead by a large measure. This is expected as physical model synthesis is computationally expensive, and the discrete GPUs have higher computational performance than the integrated GPU. Note that the discrete GPUs settle beneath the 1000ms threshold around buffer length 16. Whilst the integrated GPU is not powerful enough at any of the buffer lengths to successfully compute in real-time.

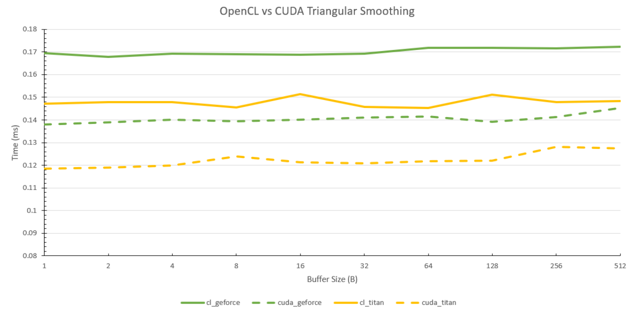


Figure 6: OpenCL vs CUDA for triangular smoothing on various buffer lengths measured in ms.

5.4 OpenCL vs CUDA

The benchmarking paper [3] compares OpenCL and CUDA in general. The paper concludes that for 'trivial' tests they have similar results and in more complicated 'non-trivial' tasks, CUDA appeared to perform better. This section discusses the latest results in 2020 in the audio domain.

Figure 6 plots the results for the complex buffer processing test for different buffer lengths. This test applies a triangular smoothing operation across the signal in the buffers. OpenCL implementations are in solid coloured lines while CUDA is shown in dashed coloured lines. It can be seen that on the NVIDIA GPUs, the CUDA implementations performed better for this kind of task. OpenCL is a defined standard implemented by supporting vendors. Considering NVIDIA develop CUDA themselves, it is possible that they have put more effort into the development of CUDA in comparison to OpenCL. The difference is small, being around $\pm 0.03ms$, though this can make a significant difference under certain circumstances.

5.5 Real-time Performance

The real time performance tests highlight the bidirectional memory transfers for a physical model synthesizer on the GPU. This test involves bidirectional memory transfers, complex and heavy computation and multiple memory accesses from within the kernel. Demonstrating if the GPUs can process a synthesizer like this within the real-time limits will set a high boundary for the GPU environment to prove itself. Table 4 shows the overall time to compute 44.1KHz of samples, the average latency of each buffer and the maximum jitter observed in the test for the NVIDIA GeForce 2080 and the Radeon 7100 in OpenCL. Both implementations seem to have a peak performance around the buffer lengths 2048 & 4096. However, at these sizes, the latency exceeds the recommended $10ms$. The best total performance within the 10ms latency is 512 & 1024. Again however, these buffers have a jitter above $\pm 1ms$. To avoid this, a smaller buffer length of 32 or 64 could be used, which still results in a comfortable half second total time for the whole process to complete.

This test involves the minimal transfer overhead, and the GPU processing. Referring to Equation 1, with buffer size of 128, $t_{total}(128) = t_{tran}(128) + c(128) + g(128)$. By considering the CPU function $c()$ negligible and taking the previously calculated baseline, the equation values consists of $1.191245 = 0.141295 + 0 + g(128)$ and therefore approximately $g(128) = 1.04995ms$. This demonstrates that for a modest buffer size, the minimal overhead is small in comparison to the complex processing that can take place on the GPU. This leaves a lot of room for processing on the GPU to take place, which supports the viability of the GPU in a real-time environment.

6. CONCLUSION

This paper has presented a microbenchmarking suite aimed at profiling GPU performance within digital audio. The benchmarking suite has been used to gather a collection of results across various hardware systems, using both OpenCL and CUDA. From the results gathered, selected sections were highlighted to explore the limitations involved when working within the GPU environment. Buffer sizes dispatched to the GPU for processing is one of the key variables impacting the performance. The general trend observed in the results showed that smaller buffer sizes from 1 to 16 could not meet the sample rate requirement, while larger buffer sizes as high as 32768 down to 512 exceeded limitations for latency and jitter. When comparing different GPU devices, the results showed that integrated GPUs have a significantly smaller transfer overhead between CPU and GPU, this is expected as they share unified memory. The discrete GPUs had a larger initial overhead to transfer data, but performed faster for more complex processing. This reinforces the idea that the integrated GPU is better suited for lighter tasks with less overhead, while the discrete GPUs include a higher initial transfer overhead, but are significantly more powerful. Therefore, assigning them more computationally expensive tasks is recommended. The microbenchmarks test results were also used to compare OpenCL and CUDA on NVIDIA GPUs. On both of the GPUs tested, CUDA appeared to consistently perform better, at least $3ms$ for the triangular smoothing test. It can be speculated that OpenCL support by NIVIDIA is not as well developed as it could be, given that CUDA is their proprietary GPGPU API. The performance benefit from using pinned memory was highlighted, showing a clear advantage for its use when compared with the standard approach. However, performance on pinned memory is dependant on how it is used and has a limited memory size.

7. ACKNOWLEDGMENTS

We would like to acknowledge the consistent support and insight from colleagues in the Computer Science Research Centre and Creative Technology Lab. Special thanks to Sam Hunt, Nathan Renney and Corey Ford. This work was supported by a grant from the HSA Foundation and the UWE 50-50 PhD fund.

8. REFERENCES

[1] J. A. Belloch, B. Bank, L. Savioja, A. Gonzalez, and V. Välimäki. Multi-channel iir filtering of audio signals using a gpu. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6692–6696. IEEE, 2014.

[2] D. Creasey. *Audio Processes: Musical Analysis, Modification, Synthesis, and Control.* Routledge, 2016.

[3] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[4] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous computing with openCL: revised openCL 1.* Newnes, 2012.

[5] M. Harris. How to implement performance metrics in cuda c/c++. `https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/l`, 2019. Accessed: 2012-10-07.

[6] R. H. Jack, A. Mehrabi, T. Stockman, and A. McPherson. Action-sound latency and the perceived quality of digital musical instruments: Comparing professional percussionists and amateur musicians. *Music Perception: An Interdisciplinary Journal*, 36(1):109–128, 2018.

[7] D. Lavry. Sampling theory for digital audio. *Lavry Engineering, Inc. Available online: http://www. lavryengineering. com/documents/Sampling_Theory. pdf (checked 24.5. 2010)*, 2004.

[8] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. Gpgpu: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006.

[9] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli. Valar: a benchmark suite to study the dynamic behavior of heterogeneous systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 54–65. ACM, 2013.

[10] C. Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[11] T. O'Haver. A pragmatic introduction to signal processing. *University of Maryland at College Park*, 1997.

[12] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[13] H. Renney, B. R. Gaster, and T. Mitchell. Opencl vs: Accelerated finite-difference digital synthesis. 2019.

[14] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming, portable documents.* Addison-Wesley Professional, 2010.

[15] M. Scarpino. Opencl in action: how to accelerate graphics and computations. 2011.

[16] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[17] P.-Y. Tsai, T.-M. Wang, and A. Su. Gpu-based spectral model synthesis for real-time sound rendering. In *Proceedings of the 13th International Conference on Digital Audio Effects, Graz*, pages 1–5, 2010.

[18] V. Zappi, A. Allen, and S. Fels. Shader-based physical modelling for the design of massive digital musical instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, page 145, 2017.

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Survival of the Synthesis - GPU Accelerating Evolutionary Sound Matching

Harri Renney* | Benedict Gaster | Thomas J. Mitchell

Department of Computer Science and Creative Technologies, University of the West of England, Bristol, UK

Correspondence
Harri Renney, UWE Bristol - Frenchay Campus, Coldharbour Ln, Bristol BS16 1QY. Email: harri.renney@uwe.ac.uk

Abstract

Manually configuring synthesiser parameters to reproduce a particular sound is a complex and challenging task. Researchers have previously used different optimisation algorithms, including evolutionary algorithms to find optimal sound matching solutions. However, a major drawback to these algorithms is that they typically require large amounts of computational resources, making them slow to execute. This paper proposes an optimised design for matching sounds generated by frequency modulation (FM) audio synthesis using the graphics processing unit (GPU). A benchmarking suite is presented for profiling the performance of three implementations: serial CPU, data-parallel CPU, and data-parallel GPU. Results have been collected and discussed from a high-end NVIDIA desktop and a mid-range AMD Laptop. Using the default configuration for simple FM, the GPU accelerated design had a speedup of 128X over the naive serial implementation and 8.88X over the parallel CPU version on a desktop with an Intel i7 9800X CPU and NVIDIA RTX GeForce 2080Ti GPU. Furthermore, the relative speedup over the naive serial implementation continues to increase beyond simple FM to more advanced structures. Further observations include comparisons between integrated and discrete GPUs, toggling optimisations, and scaling evolutionary strategy population size.

KEYWORDS:
GPU, Evolutionary Computing, Synthesis, Benchmark, Parallel

1 | INTRODUCTION

Modern technology has had a profound effect on the structure, form and performance of music. Powerful and inexpensive general-purpose systems have made musical apparatus universally available to amateur and professional composers alike. The audio synthesiser is a core component in the development of music, enabling composers to recreate acoustic instrument sounds or explore entirely new sounds electronically. Numerous synthesis techniques have been discovered and enabled the creation of a considerable range of timbres. Synthesisers expose controllable parameters, which shape the sound character of the particular synthesis architecture. Consequently, there is often a complex mapping between the control space of synthesis parameters and the timbre space of the sound's perceived character. Therefore, effective control and navigation of a synthesiser's sound space requires expert knowledge of the underlying technique, often drawing from deep theoretical and/or experiential knowledge. Steps towards achieving an automated process for mapping sound qualities to sound synthesis parameters could make synthesisers a more transparent compositional tool. Achieving this will require techniques that can efficiently search the synthesis parameter space to identify parameter configurations to match specific timbral characteristics.

Abbreviations: ES, evolutionary strategy; GA, genetic algorithm; GPGPU, general-purpose GPU; FM, Frequency Modulation;

In the field of Evolutionary Computation, there are powerful optimisation techniques designed to search complex design spaces and find optimal solutions. Evolutionary algorithms have previously been applied to synthesis parameter matching with promising results[1]. However, a major disadvantage of optimisation algorithms, including evolutionary ones, is that they typically require substantial computation time to converge on optimal solutions. Processing units with data-parallel architectures open up opportunities for increasing computational throughput for appropriate cases of the algorithm. The Graphics Processing Unit (GPU) has matured from its origins in the graphics domain into a general-purpose hardware accelerator known as a general-purpose GPU (GPGPU)[2]. GPGPU computing aims to exploit the full potential of massively parallel processing architectures for general processes[3] and evolutionary sound matching are a suitable class of problems for fine-grained parallel processing, with the potential to map efficiently to the GPU environment. Therefore, using the GPU may provide an opportunity to improve the performance of synthesis parameter matching.

This work investigates the potential performance benefits of GPU processors for synthesiser parameter matching using evolutionary computing. A design is proposed that uses the evolution strategy[4] for searching the parameter space of an FM synthesiser. The performance between CPU and GPU implementations is evaluated using a comprehensive benchmarking suite. The application of evolutionary computation for FM synthesis parameter matching is a well-explored area and has proven to be an effective and accurate technique for both simple and more advanced FM synthesis structures[5 6 7]. However, a major disadvantage is that they require considerable computational time, and researchers have highlighted a desire for ways to optimise this approach[8]. Therefore, this work does not advance the state-of-the-art in evolutionary computation for audio synthesis matching, rather, it proposes algorithms that accelerate existing methods by providing a data-parallel GPU design that overcomes challenges faced by the distinct architecture. The contributions of this work can be summarised as:

- A design for a GPU accelerated sound matching framework.
- The implementation of the design in the OpenCL GPGPU environment.
- A benchmarking suite for collecting performance profiles across hardware systems for serial CPU, data-parallel CPU and data-parallel GPU implementations.
- Benchmark results from different systems demonstrate the performance acceleration of the proposed design.

2 | PREVIOUS WORK

There has been extensive prior research surrounding the application of Evolutionary Computation for matching sounds using audio synthesisers. This is typically realised by optimising the parameters of a particular synthesis type using Genetic Algorithms (GA)[9] or Evolution Strategies (ES)[10]. FM has become a common application domain for sound matching, and the GA is by far the most prevalent approach adopted in the literature[9 11 12]. For example, in[13], a GA is used for matching sounds using FM synthesis, which uses an advanced timbral extraction technique for assessing the fitness of potential solutions. The authors demonstrate how the advanced technique increases the efficiency of the evolutionary sound matching application.

More recently, Smith[14] demonstrated the accurate matches for an electronic keyboard and piano synthesiser using a modified GA; this approach was considered promising for music producers. However, Smith stressed that the time to find solutions was an essential component when evaluating the practicality of the method. Therefore, optimising the speed of the matching process would be a beneficial next step in future work. Yee-King et al. made a comparative study between several automatic synthesiser parameter matching algorithms[15], particular focus given to neural networks and genetic algorithms. Neural networks proved to find accurate solutions in "near real-time". However, this approach first required a training phase that took a day to complete. In contrast, the genetic algorithm required no pre-training but required 2 hours to find a solution. The advantage of the genetic algorithm is that no training is required when adapting the algorithm for a new synthesiser. However, it still requires significant computation and would therefore benefit from optimisation.

The concept of using the graphics hardware for general-purpose processing was introduced in the 1990s with works like Lengyel et al., who used graphics devices for robot motion planning[16]. GPUs have advanced significantly and are now accessible for general-purpose programming outside of graphics. In 2006, NVIDIA[17] introduced the Tesla GPU, introducing the beginning of a new unified architecture. Contemporaneously, AMD made similar developments with the TeraScale GPU. NVIDIA and AMD have continued to develop comprehensive GPGPU support in all subsequent GPU architectures[18]. GPGPU has been successfully applied for processing a range of numerical and scientific computation techniques like molecular dynamics[19] and audio synthesis[20]. A well-known and successful example of this is the folding@home project[21], which reported a speedup of 20 - 30X when accelerated on the GPU. These examples motivate the exploration and establishment of GPU designs for appropriate methods and applications. Recently, Turian et al.[22] have leveraged the GPU to generate a billion sample dataset for synthesisers, including FM synth timbre and subtractive synth pitch. With the use of the GPU, this dataset is 100x larger than any other recorded datasets in the literature.

The success and nonlinear mapping of FM synthesis has generated interest in designing parameter and sound matching methods. Consequently, this paper's proposed application is designed for targeting FM synthesis in particular. The results in Section 6 starts with benchmarking the GPU accelerated design for the simple FM synthesis algorithm originally proposed by Chowning[23]. This limits the complexity of the synthesis stage and allows the benchmarking results to also emphasise the performance of the audio analysis and evolutionary computing stages. After a complete discussion of the simple FM synthesis results, two more advanced synthesisers are presented, and the evaluation of the GPU accelerated performance beyond simple FM synthesis is explored to determine the scalability of the design's capability to support other FM structures. The design proposed in this work is limited to analysing static sounds and does not support dynamic sounds that change with time. Although, steps towards supporting dynamic sounds by analysing audio in blocks is partially supported at the moment and discussed in the results.

There are other well-studied methods for synthesiser sound matching, such as the hill-climber algorithm. Although contextually applicable, the hill-climber algorithm has been considered insufficient for successful exploitation of FM synthesis parameter space by Horner in[24], whilst the evolutionary algorithm proved superior. Neural networks are another powerful method with superior real-time performance when matching sounds. However, they require extensive pre-training periods that can make them unable to handle dynamic problems. The appeal of the evolutionary approach is that it requires no pre-training and can be easily applied to match sounds using different synthesis techniques. However, the trade-off is that evolutionary algorithms typically take considerable time to then find the solutions. Improving the performance will enable larger population sizes to be used in practical time scales, improving the accuracy and the usability of the approach. The population size is a parameter of the evolutionary algorithm, enabling greater exploration of the search space at the cost of further computation. Therefore, optimising the scaling of the population will improve the evolutionary algorithm's exploration of the problem space more effectively[25]. A natural consequence of increasing the population size is that the computation increases. If the performance can be improved, the range of applications that synthesiser parameter matching can be used for will increase. Although many of the papers referenced here use the genetic algorithm, we adopt a different, albeit nearly identical, evolutionary algorithm called the evolution strategy. Both algorithms model the process of evolution to some degree, but we have chosen to work within the framework of an evolution strategy as its canonical form represents real numbers directly as floating-point numbers and so lends itself neatly to the FM matching domain. However, with the appropriate adjustment of terminology, this work equally applies to genetic algorithms. The literature on evolutionary algorithms for sound matching is well explored and has shown to be an effective method over the alternatives. Using ideally large populations in evolutionary algorithms requires significant computation and researchers suggest that finding ways to optimise this in the context of sound matching will be a beneficial advancement.

2.1 | FM Synthesis

One of the first commercially available and successful digital audio synthesis methods was FM synthesis[23]. Invented by John Chowning, FM synthesis was discovered and initially developed in the 1970s[26]. It is regarded as a highly efficient method for generating complex and rich audio timbres with simple graphs of interconnected sinusoidal oscillators. The original equation proposed by Chowning, with some symbols modified for this paper, is given as:

$$y(t) = A\sin(ct + I\sin(mt)) \tag{1}$$

Where four parameters are exposed: peak amplitude *A*, carrier frequency *c* (rad/s), modulation frequency *m* (rad/s) and modulation index *I*. Therefore, *y* generates an instantaneous FM output for a given time *t*. The modulation frequency *m* sets the frequency of the modulating oscillator, the output of which is multiplied by the modulation index *I* to control the intensity of the frequency modulation applied to the carrier oscillator. This is then added to the input for the carrier oscillator's frequency *c*. Finally, the variable *A* is used to control the peak amplitude output. When the modulation index $\mathrm{I} = 0$, the modulation oscillator has no effect. When $\mathrm{I} > 0$, the modulation oscillator begins to affect the carrier oscillator, and symmetrically spaced intervals of frequencies occur above and below the carrier frequency. The number of side frequencies relates to the modulation index; therefore, as *I* increases, energy is taken from the carrier frequency and distributed further across the sideband frequencies. Bessel functions determine the amplitudes of the carrier and side frequencies and the interested reader is referred to Chowning's original work[23] for further details . FM synthesis has a nonlinear mapping between parameters and the resulting sound[27], this means that there is a non-trivial correlation between a synthesiser's parameter space and the resulting timbre space, i.e., minor changes to the input parameters can generate vastly different changes in the resulting sound. This makes it a difficult synthesiser to control and parameter match.

Simple FM synthesis is used as a fundamental building block in more complex and musically interesting synthesisers. The technique was adopted by Yamaha in the DX and TX synthesisers that were commercially produced throughout the 1980's[28] and is still a prevalent technique used in many plugins and synthesisers that are currently available[29]. In this paper, two further synthesisers are considered for GPU optimisation that build upon Chowning's simple FM synthesis technique. The first has been established as a "real-world" by Das et al. and is used to evaluate evolutionary algorithm performance[8]. In this paper, this will be referred to as nested modulator FM synthesis[30]:

$$y(t) = Asin(ct + I_1 sin(m_1 t + I_2 sin(m_2 t))) \tag{2}$$

Where y is the output of the nested modulator FM, *t* is the input variable time, *A* is the peak amplitude, *c* is the carrier frequency, *m* is a vector of modulator frequencies and *I* is a vector of modulation indices. This equation adds a second nested modulation oscillator that modulates the original modulation oscillator of the simple FM structure seen in Equation 1. This arrangement requires a total of 6 parameters to control the output sound. The third FM synthesis method considered in the scope of this paper uses the idea of combining separate FM synthesis components in parallel. For the context of this work, the parallel FM[1] will be used:

$$y(t) = \sum_{i=1}^{3} A_i \sin(c_i t + I_i \sin(m_i t)) \tag{3}$$

Where *y* is the output of the parallel FM, *t* is the input time, *A*, *c*, *m* and *I* are vectors of the peak amplitude, carrier frequencies (rad/s), modulation frequencies (rad/s) and modulation indices for each FM structure. This equation is the summation of three separate simple FM structures, each having four parameters and therefore requires a total of 12 parameters to control the output. This paper begins by considering the GPU optimisations concerning simple FM synthesis and then scales up to assess the nested modulator and parallel FM synthesis methods.

3 | DESIGN

This section covers the abstract overview of the FM synthesis parameter matcher process. Figure 1 are used to aid the reader in understanding the entire flow of the program, considering there are many stages involved. Each of the stages are explained with sufficient detail in the following subsections.

3.1 | Evolution Strategies

Evolution strategies[4] are a class of evolutionary algorithm[31] that is used to find optimal solutions within a search space. Evolution strategies follow ideas inspired by Darwinian evolution and natural selection to create progressively fitter solutions to a problem by iteratively applying mutation and recombination operations on a set or population. Evolution strategies can be used for any optimisation problem, including sound matching. In this paper, the algorithm is used to search the parameter space of an FM synthesiser to find the parameters that closely replicate a given target sound[10].

A population of solutions comprises a set of individuals where each individual contains a complete set of synthesis parameters that can be used to generate a candidate sound or solution to the sound matching problem. For the simple FM synthesiser, this corresponds to the four parameters described in Equation 1, along with an additional set of four step-size parameters used by a self-adaptive mutation operator.

Recombination blends the genetic material (or parameters) from two or more parent individuals' to generate new offspring. Figure 2, shows how the uniform discrete recombination[31] operator works, which is one of the standard ES recombination operators used in this work. The value at each position is taken from a random parent in the current population and combined to create a new individual, used in the next generation offspring population.

Figure 3, shows how the mutation operator typically works. The random values shown in red are generated for each element using a pseudorandom number generator and Gaussian Distribution[32], scaled in proportion to the step size. These values are then added to the individual parameters to introduce novelty to the population. Gaussian distribution is used to increase the likelihood of smaller mutations occurring more frequently than larger values. This results in smaller steps in values but occasionally larger steps which can be useful to explore the search space and to escape local optima. The self-adaptive mutation operator uses the step size parameter to enable distant values (exploration) as well as precise local values (exploitation) to be generated at different stages of evolution.

3.2 | Fitness

The evolutionary strategy's fitness function (or objective function) is context dependant and defined by the optimisation problem. For synthesiser parameter matching, this function requires two stages. First, a clip of audio is generated by the chosen FM synthesiser using the parameters of

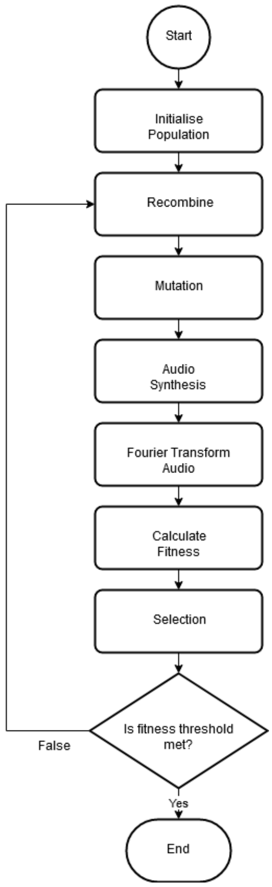


FIGURE 1 Flow diagram for the overall evolutionary parameter matching design flow.

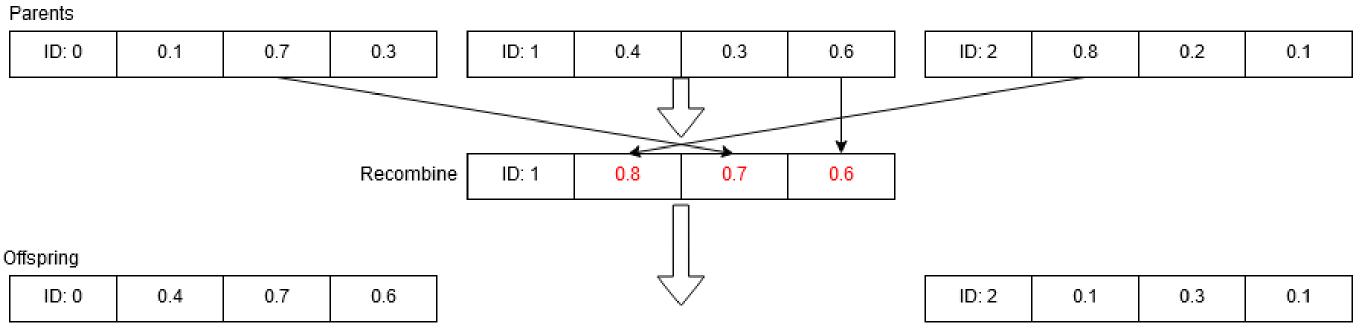


FIGURE 2 Recombine operator working on three example individuals

each population individual. Second, the generated audio is compared to the target audio, and the similarity between them is used to determine the "fitness" of each individual.

Each individual in the evolutionary strategy population has four parameters when used for simple FM synthesis parameter matching. The values of these 4 parameters vary between individuals and are mapped to the 4 parameters of the simple FM synthesis algorithm in Equation 1. A frame of audio is then generated from the FM synthesiser and stored contiguously in memory, ready for further processing to determine similarity with the target audio.

The similarity between the generated and target audio is determined by first mapping the audio signal to the frequency domain using a Fourier transform. The fast Fourier Transform (FFT)[33] is the de facto algorithm for calculating the Fourier transform of a signal efficiently. However, the FFT algorithm requires the input data to be cyclic, spanning from one end and back to the beginning. Therefore, before the FFT algorithm can process an audio signal, it must first undergo a form of pre-processing; this is known as FFT windowing. To reduce the occurrence of artefacts in

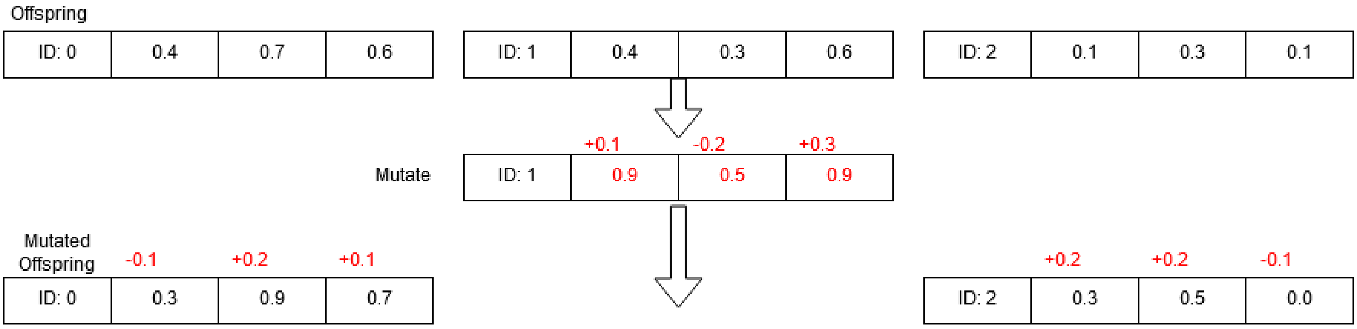


FIGURE 3 Mutation operator working on three example individuals.

the spectrum occurring as a result of frame boundary discontinuities, a windowing function is used. In this paper, a Hann window[34] is applied to each audio frame as follows:

$$\omega(n) = 0.5 \times \left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right) \qquad (4)$$

Where ω is the processed Hann value, n is the value of the current sample and N is the total number of samples in the window. Once the time-domain audio signal is prepared using Hann windowing, it can be processed by the FFT algorithm[35] to produce a series of complex numbers representing the same signal in the frequency domain. FFT is a highly optimised and supported algorithm that is available in software libraries such as FFTW[33], which handles the intricacies of the implementation. With each respective individual's audio signal mapped to the frequency domain, the similarity of the signal to the target audio can be calculated. A primary method for comparing the error/difference between two frequency spectra can be achieved using relative spectral error. Studies performed by Beauchamp et al.[36] have shown that the relative spectral error delivers the best correspondence to average discrimination data extracted from human listeners when compared with alternative spectral error metrics. The equation for relative spectral error is defined as:

$$rse = \sqrt{\frac{\sum\limits_{b=0}^{N_{bin}} (T_b - S_b)}{\sum_{b=0}^{N_{bin}} T_b^2}} \qquad (5)$$

Where rse is the calculated relative spectral error, T and S are vectors of the target and synthesised audio frequency spectra respectively, and N_{bin} is the number of frequency bins that control the resolution of the analysed spectra. The relative spectral error (RSE) between two audio signal frequency domains T and S can be calculated. The number of frequency bins N_{bin} is the resolution of the frequencies represented in the audio frequency spectrum, this must be the same for both T and S. Using the RSE, the fitness for each individual as a candidate for matching with the target signal can be determined. As the RSE is the error between two frequency spectra, the fitness is the inverse of the RSE. Therefore, an RSE $= 0.0$ is a perfect match, whilst increasing rse identifies differences between the signals. Evolution strategies typically involve a stopping criteria, such as when a sufficiently "fit" individual is found, the evolutionary strategy iterations stop and the individual used as the solution. In the context of this work, the stopping criteria is a set number of iterations/generations to avoid an undefined number of iterations and so the computation time can be fairly measured.

The selection stage involves taking a set number of individuals from the offspring and using them in the next generation's parent population. The approach used in this paper is to sort the individuals by fitness and select the top individuals for the next parent generation. A parallel merge sort is used and has been shown to map efficiently into the GPU architecture[37].

4 | GPGPU DESIGN

The GPGPU literature contains a range of different approaches and associated terminology. In this work, the OpenCL standard language will be used[38 39 40]. The GPU architecture is comprised of streaming multiprocessors that load and execute program instructions in a parallel Single-Instruction Multiple-Data (SIMD) format. The control flow of GPU programs comprises of workitems; these are streams of execution running on

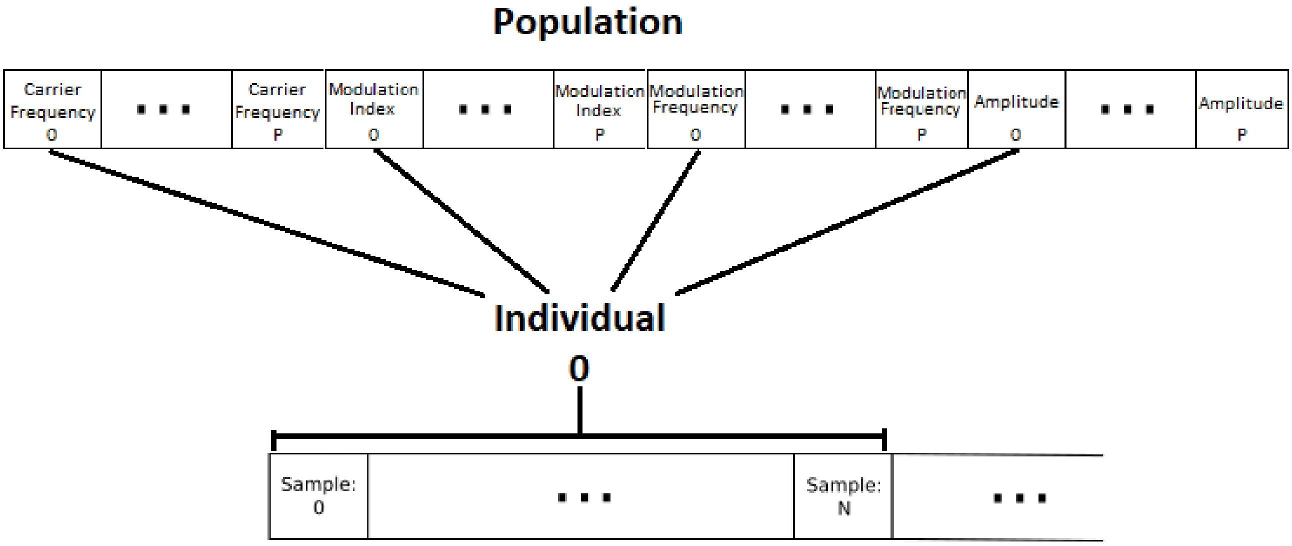


FIGURE 4 The data structure format for the GPU design.

the multiprocessors. The multiprocessors require access to regions of memory arranged in a memory hierarchy similar to a CPU, where memory types vary in size and access speed[41]. The largest region accessible by all workitems is known as global memory. It is typically the slowest region of memory to access. Local memory is a faster region only accessible in a small group of workitems, this group is called a workgroup. Finally, private memory is the smallest but fastest memory, accessible only by the workitem itself. In this section, the techniques and design choices relevant to the GPU implementation are considered.

4.1 | Data Structure

All GPU processed data starts with the ES population. Figure 4, provides a visual aid to help describe the data structure that is used in this GPU design. The first buffer indicated by "Population" contains all the individuals' parameters. Although naturally it seems coherent to place each individual's parameters contiguously in memory, this structure is sub-optimal within the GPU. Therefore, when designing data structures for the GPU, it is advantageous to store all of the first parameters contiguously, then the second parameters, and so forth. This design is commonly referred to as a structure of arrays instead of an array of structures. Various research demonstrates that this is the most efficient GPU data layout for comprehensive processing[42 43]. The parameters of an individual are fetched by indexing each array of parameters using the individual's ID. During the synthesis stage, each individual has an audio block of a set length of samples generated from the parameters. Each audio block is held contiguously in memory, against the generated audio of the next individual's parameters. With this format, the size of the buffers are: $\text{population} = \text{P}, \text{audioSamplesBlocks} = \text{P} * \text{N}$ where P is the population size and N is the audio block length.

The population buffer containing each individual's values is stored in one large GPU buffer as the GPU is optimised for fewer, larger memory allocations with offsets to access them than multiple smaller ones. Furthermore, the population buffer is twice the size of the population in order to store a copy of the sorted population when evaluating population fitness. Using a rotation index, an offset to the beginning of the sorted population can be used. This greatly improves performance and memory usage as it avoid copying memory between three separate population buffers.

In the context of the GPU, memory coalescing is achieved when threads on the same streaming multiprocessor access memory simultaneously. To achieve this, consecutive threads should be programmed to access consecutive memory addresses. The GPGPU programming model exposed by OpenCL involves reconsidering the way loops iterate when processing data. The application proposed in this paper aims to optimize memory accesses where possible. This is typically achieved by using the workitem index *idxWorkitem* and loop index *i* in such a way as to access adjacent memory at once between workitems, instead of accessing memory in completely different locations in memory. An example of correctly programming the GPU to achieve memory coalescing can be found in the following FFT Hann windowing kernel:

```
void applyWindowPopulation(float* audioDate, uint32_t audioLength)
{
    int idxWorkitem = get_global_id();
    float mu = ( FFT_ONE_OVER_SIZE - 1) * 2.0f * M_PI;

    int stride_factor = audioLength;
    for(int i = 0; i < POPULATION_COUNT; i++)
    {
        int coalesced_index = stride_factor * i + global_index;
        float fftWindowSample = 1.0 - cos(index  * mu);
        audio_waves[coalesced_index] = fft_window_sample * audio_waves[coalesced_index];
    }
}
```

Here, Equation 4 is implemented on the GPU with an optimized memory access pattern. The index is multiplied with a 'stride' factor equal to the audio wave size. This forces the indices of each workitem to access memory adjacent to one another when processing the windowing. All the indices in the respective workitems then stride to the next audio wave to process when *i* increments. This means that each workitem can share the memory fetched by its neighbouring workitems without requesting a memory fetch somewhere else in memory itself. A more comprehensive description of memory coalescing on GPUs with visual aids can be found in[44].

4.2 | Processing Format

The population size determines the number of workitems the GPU allocates for processing in parallel. These workitems are dispatched to execute each individual's recombination, mutation, audio synthesis, fitness, and selection. Each individual is entirely independent from all other individuals and therefore can execute efficiently without stalling to synchronize data. The FFT processing is handled separately using the clFFT library[45], a well-established and optimized open-source implementation of FFT on the GPU.

During the audio synthesis stage, the chosen FM synthesis arrangement will typically make calls to the trigonometric functions like sin(). These are computationally expensive compared to other basic math operations. Optimized implementations might use Taylor series expansion or the CORDIC algorithm[46]. Instead of calculating these values, a lookup table of previously calculated sine values can be used[47]. As covered in Section 3.2, each individual is tasked with generating an audio block of samples using simple FM synthesis. A pre-calculated wavetable of values approximating sin is uploaded to the GPU and indexed instead of using the computationally expensive sin operator. A finite number of values can be stored in the lookup table, resulting in quantization. Therefore, sufficient resolution must be used in the lookup table. The wavetable optimized synthesis stage (without interpolation for simplicity) is demonstrated in the GPU kernel code snippet below:

```
int idxWorkitem = get_global_id();

float cur_sample = 0.0;
float pos_1 = 0.0;
float pos_2 = 0.0;

const float wavetableIncrementOne = (WAVETABLE_SIZE / 44100.0) * modulationFrequency;
for(int i = 0; i < WAVE_FORM_SIZE; i++)
{
    cur_sample = wavetable[pos_1] * modulationIndex + carrierFrequency;
    out_audio_waves[idxWorkitem * WAVE_FORM_SIZE + i] = wavetable[pos_2] * amplitude;

    pos_1 += wavetableIncrementOne;
    pos_2 += (WAVETABLE_SIZE / 44100.0) * cur_sample;

    if (pos_1 >= WAVETABLE_SIZE)
        pos_1 -= WAVETABLE_SIZE;
    if (pos_1 < 0.0)
        pos_1 += WAVETABLE_SIZE;
    if (pos_2 >= WAVETABLE_SIZE)
        pos_2 -= WAVETABLE_SIZE;
    if (pos_2 < 0.0)
        pos_2 += WAVETABLE_SIZE;
}
```

Here, the section of the code that generates the simple FM synthesis waveform is shown. This implements the equivalent of Equation 1 on the GPU using a wavetable optimisation. In this code, *m* is contained in variable "modulationFrequency", *I* in "modulationIndex", *c* in "carrierFrequency" and *A* in "amplitude". Instead of making calls to $\sin(\dots)$, the wavetable is accessed using pos_1 and pos_2 that are incremented in such a way as to access approximate pre-computed values of $\sin()$. Variable *idxWorkitem* multiplied with WAVE_FORM_SIZE is used to stride through the memory written to in a GPU optimized way.

Batching processes to the GPU is a fundamental technique involved in GPU programming. It is often used to avoid consuming GPU memory resources if an individual task is too big. This can quickly become the case here, as the population size is a controllable parameter, which can exceed GPU memory limits. The amount of GPU memory resources depends on the system's hardware. Taking the systems in Table 1 as examples, the GeForce 2080 has 8GB of memory whilst the Radeon 530 has only 2GB. The intel UHD GPU shares its 8GB of RAM with the CPU. In order to avoid misusing the GPU memory, the GPU design proposed breaks up data into manageable, equally sized blocks, loads them onto the GPU and processes them one at a time. The pseudocode for the batching is given below:

```
void parameterMatchAudio(float* aTargetAudio, uint32_t aTargetAudioLength)
{
    blockSize = objective.audioLength;
    blocks = aTargetAudioLength / blockSize;

    for (uint32_t i = 0; i < blocks; i++)
    {
        setTargetAudio(&aTargetAudio[i*blockSize], blockSize);
        initPopulationCL();
        executeAllGenerations();
    }
}
```

An additional advantage to using the batching technique is that the system is easily extended in the future to involve analysing dynamically changing sounds. This means if the characteristics of a sound changes over time, analysing each block can identify the parameters necessary to match the changing sound in each block. If this advancement was explored, there would be a set of parameters for each audio block analysed.

5 | BENCHMARKING

The benchmarking suite involves the autonomous execution and performance profiling of a collection of planned tests. Configurations of parameters are exhausted and the results collected and written to files for analysis. The benchmarking suite is deployed on different devices to collect profiles on a range of hardware systems. The GPU implementation will synchronize between each OpenCL call to confirm the action is finished to accurately measure the execution time. The benchmarking suite is open-source and publically available online [a]. The implementations here use OpenCL v1.2 in order to support the widest range of hardware possible. Three different implementations were developed from previously defined designs.

CPU Serial - In order to establish the minimal baseline performance, a serial single-core implementation targeting the CPU. Modern CPUs include multiple cores and vector processors which are not used here. The purpose is to demonstrate the impact of restricting a program to a purely serial format.

CPU OpenCL - A parallel implementation targeting the CPU using OpenCL. Following the OpenCL programming model utilizes all parallel processing components of the target hardware. In the case of the CPU, multiple cores and vector processors are utilized. This implementation will be compared to the serial CPU version, but more importantly, the massively parallel GPU version.

GPU OpenCL - Implementation using OpenCL to target parallel processing on the GPU.

[a] https://github.com/Harri-Renney/Survival_of_the_Synthesis-GPU_Accelerated_Frequency_Modulation_Parameter_Matcher

Specification	Mid-range Laptop	High-end NVIDIA GeForce
CPU	Intel Core i7-8550U	Intel Core it-9800X (16 SM)
Integrated GPU	Intel UHD Graphics 620	None
Discrete GPU	AMD Radeon 530	GeForce RTX 2080 Ti (32 SM)
CPU RAM	8GB	32GB

TABLE 1 Hardware specification used for benchmarking. Streaming Multiprocessor (SM)

Parameter	Value	Notation
Number Generations	1000	G
Number Parameters	4	D
Parent Population Size	1024	P
Offspring Population Size	7168	O
Target Audio Length	2048	T
Audio Block Size	2048	N
GPU Workgroup Size	32	W

TABLE 2 Default benchmarking parameters

5.1 | Hardware Systems

The benchmarking suite was run on a collection of different systems containing GPU devices from different hardware vendors, including integrated and discrete GPUs. GPUs are often classed as being one of two types, discrete or integrated GPUs. Discrete GPUs are separate from the CPU and connected across system buses known as PCI. However, communicating over the PCI buses involves data transfer overhead. In contrast, integrated GPUs are tightly coupled to the CPU, even sharing memory spaces. This means that the latency overhead is avoided and communication between CPU and GPU is faster than the discrete GPU[48]. However, integrated GPUs have constraints imposed by limited physical space and therefore typically have fewer and slower multistreaming processors. The hardware systems considered in this paper are detailed in Table 1. In addition, results have been collected for another system titled "High-End NVIDIA Titan". This system's results have not been included in the paper but are similar to the "High-End NVIDIA Geforce" system. The results for this system can be found in the full database of results found at the link provided in Section 6.

5.2 | Benchmarking Targets

The benchmarking suite is designed to measure the overall execution for default parameters and used to measure execution time when scaling controllable parameters. The controllable parameters affect the performance and accuracy of the algorithm and therefore play a crucial role. Unless stated, the default parameters will have the values shown in Table 2. The default parameters have been chosen as they provide solutions with acceptable fitness and give the benchmarking sufficient processing to profile and consider. Instead of using a fitness threshold as the stopping criteria, the benchmarking suite will execute a fixed number of generations. This ensures parity between results by guaranteeing that all implementations processed an equal amount of computation, mitigating any stochastic variations. The focus of this benchmarking suite is to compare the performance differences between implementations for the basic ES and FM synthesiser sound matching application. It is not to evaluate the accuracy of the solutions generated by the ES, the literature covered in Section 2 has already established ES as an effective method for accurately finding solutions. However, the reader can be reassured that the default parameters' fitness converge to an error of 0.0 and, therefore, an exact match is found using the default parameters. The benchmarking suite covers performance profiling the following aspects listed:

Implementation	Total Execution Time (s)
CPU Serial	409.98
CPU OpenCL	28.33
GPU OpenCL	3.19

TABLE 3 Total execution time of the application for the CPU and GPU implementations.
Parameters = Default **System** = High-end NVIDIA GeForce

Overall Execution - The overall execution time for the program to complete is the primary concern. The overall execution time includes all stages in Section 3, including GPU specific initialization of: The starting population, wavetable (Section 4.2), random number lookup table.

Program Stage Execution - In order to highlight the computational implications at each stage of the processing, the stages in Section 3 are independently timed. This is useful to expose potential bottlenecks and the most intensive stages.

Audio Analysis Block Size - The size of each block of audio analysed at a time by the application. The block size controls the amount of data transfers between CPU and GPU. Therefore, the impact of scaling the audio block size will be evaluated in these results.

Population scaling - It has been shown that increasing the population size in evolutionary algorithms can produce fitter solutions in a more complicated problem space. Measuring how the performance and accuracy are affected when changing population size is an important aspect.

optimisations On/Off - In order to demonstrate any performance benefits from using the GPU optimisation in Section 4, implementations employing the optimisations will be profiled against implementations that do not use them.

Discrete vs Integrated - The benchmarks are executed on an integrated and discrete GPU in the same system. This maintains a common environment excluding the target GPU, provides insight into the performance of integrated and discrete GPUs for fair comparison.

6 | RESULTS

In this section, the results collected across the benchmarking suite are presented, along with a discussion of the results. This paper considers the salient results collected for the two systems outlined in Table 1. The full collection of results including another High-End NIVIDA Titan setup have been made available online [b]. Particular focus on the results are given to the High-End NVIDIA GeForce desktop that better reflects the expected audience of synthesis parameter matching. The laptop is used to highlight integrated vs discrete GPUs and the difference between a high-end desktop and mid-range laptop. Readers can use the open-source benchmarking suite provided in Section 5 to collect results for additional systems.

6.1 | CPU vs GPU

Comparing the execution speed between the CPU and GPU designs is of primary interest. The results shown in Table 3 include the three implementation's total time with the default parameters for the High-End NVIDIA system. The CPU serial version takes considerably longer at 409.98s, whilst the OpenCL CPU and GPU are 28.33s and 3.19s. The CPU has parallel vector processors available and it can be seen that a clear improvement of 14X speedup is achieved when using them by the CPU OpenCL version. GPUs advance this data-parallel processing further, achieving a speedup of 128X over the serial CPU version. This demonstrates that not only are ES and FM synthesis suitable for data-parallel processing, but that they are also suitable in combination using the design proposed in this paper. Comparing the CPU OpenCL version to the GPU OpenCL version, it can be seen that the GPU has a speedup of 8.88X over the CPU. This is expected as the GPU architecture is designed to maximize the data-parallel processing whilst the CPU is not.

6.2 | Implementation Stages Compared

The execution time for each stage in the application on the High-end NVIDIA system has been recorded and displayed in Figure 5. The execution time is marked in ms on a logarithmic scale using the default parameters. When comparing each implementation, the stages follow a similar pattern of the CPU serial taking the most time, followed by CPU OpenCL and finally GPU OpenCL. There is a particularly high improvement on the GPU for the "synthesis", "Window" and "FFT" stages. All these stages are included in the design and highlight the importance of this novel approach

[b] https://muses-dmi.github.io/benchmarking/benchmarking_database_survival_of_the_synthesis

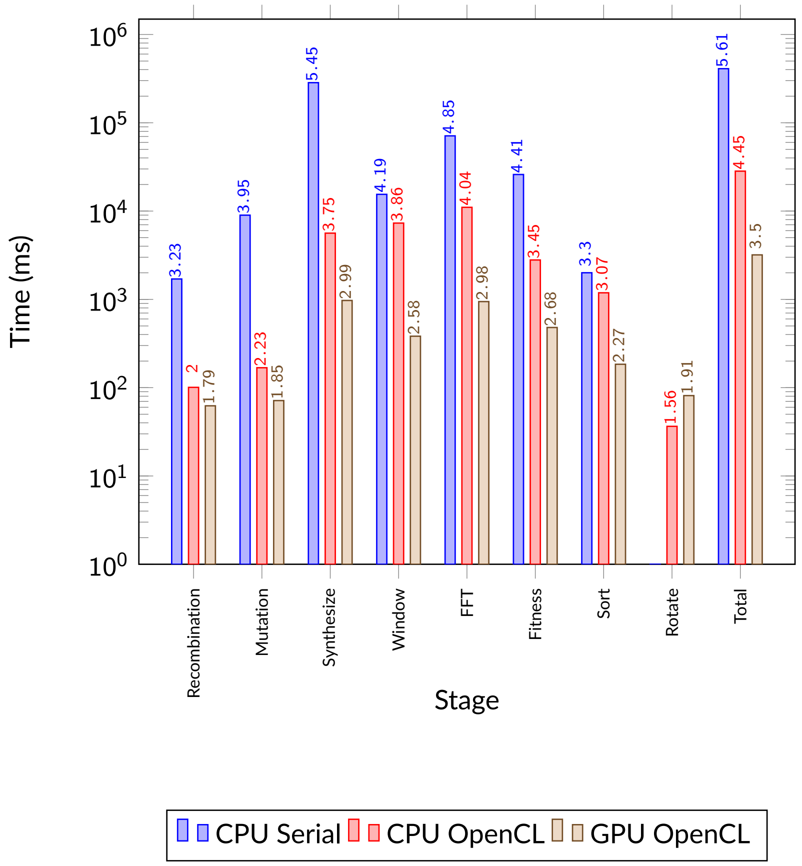


FIGURE 5 Execution time of each stage across the CPU and GPU implementations. **Parameters** = Default **System** = High-end NVIDIA GeForce

considering the domain-specific stages and the evolutionary algorithm. The data-parallel version improves over the CPU versions in all stages, except for the "Rotate" stage, where the CPU serial had a comparatively negligible 0.3544ms. This is a negligible amount as it only involves updating a variable in CPU memory. In contrast to the OpenCL versions that require an OpenCL function with added overhead. For example, the GPU version requires updating the rotation index in GPU memory, this involves a data transfer over the PCI interface to update the variable in GPU memory. This is a stage the CPU will naturally surpass the OpenCL versions. However, it is a small difference to the vast improvements observed in the rest of the stages.

6.3 | Hardware Systems Compared

The High-End NVIDIA desktop is expected to execute faster for all population sizes as it utilizes the more powerful NVIDIA GeForce RTX 2080 Ti, whilst the mid-range Laptop contains a less capable AMD Radeon 530. It can be seen in Figure 6 that the NVIDIA 2080 GPU shows little increase in computation time when the population is scaled up to 32768. Whilst the AMD 530 GPU has a roughly directly proportional increase in execution time with population size, the NVIDIA 2080 architecture contains more multistreaming-processors and therefore can process more individuals in parallel. This demonstrates, as is the case with most processes in computing, that the speed and throughput of processing units is still an important factor for data-parallel processors.

6.4 | Kernel Execution Time Ratio

Figure 7 shows the relative execution times for every stage of the GPU OpenCL implementation running with default parameters. This highlights where the majority of the processing takes place. The stages processing the ES population: "Recombine", "Mutate" and "Sort" together only account for less than 13% of the time. These stages execute quickly as they only process the population $(P + O) * D$, which is $(1024 + 7168) * 4 = 32,768$

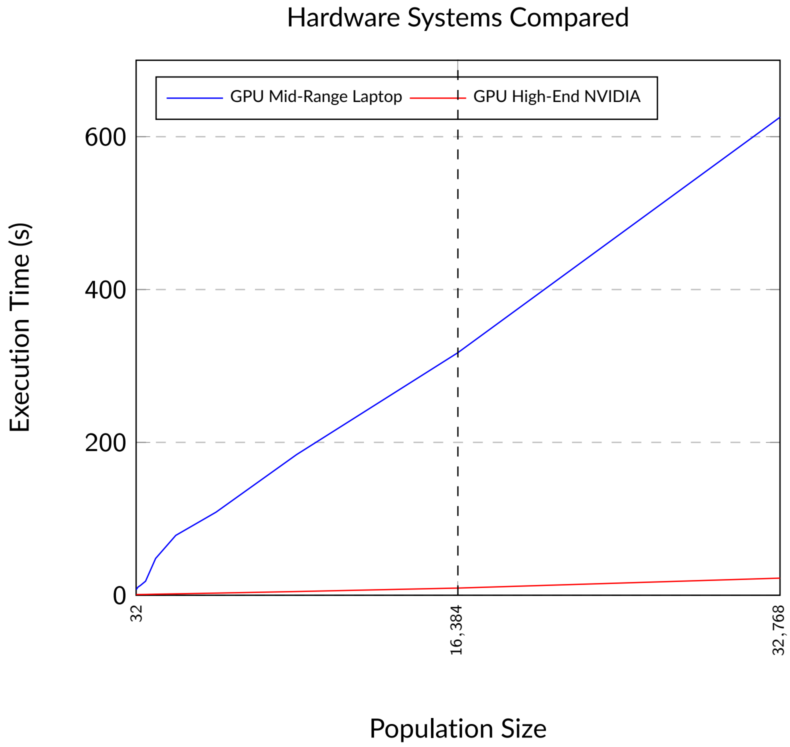


FIGURE 6 Execution time when scaling the population size for the High-End and Mid-range systems.
Parameters: G = 1000, P+O = Scaled **System** = High-end NVIDIA GeForce & Mid-range Laptop

floating point values for the default parameters. The stages processing audio for the population: "Synthesis", "Window", "CLFFT" and "Fitness" take considerably longer accumulating 87.43% of the time. The audio stages require processing $(P+O)*N$ which is $(1024+7168)*2048 = 16,777,216$ floating point values. The majority of the execution time is taken up by the synthesis and CLFFT stages, taking around 30% of the time each. This highlights the importance of the design this paper describes as it not only efficiently processes the evolutionary algorithm stages, but incorporates the domain-specific processes that take up the majority of the execution time.

6.5 | Integrated vs Discrete

Figure 8 shows the results comparing the integrated Intel and AMD discrete GPU in the mid-range laptop. It can be seen that initially, when the population size is small, the integrated GPU performs better than the discrete GPU. This is expected as using the PCI-e system bus to communicate between the discrete GPU and the CPU takes considerably longer than sharing a memory space. The communication overhead surpasses any benefits of using the discrete GPU for smaller population sizes, whilst the integrated GPU avoids the communication overhead using unified memory. However, at around population size 2048, the performance of the discrete GPU exceeds the integrated GPU. The more powerful discrete GPU scales more efficiently for larger population sizes and the benefit exceeds the communication overhead involved.

6.6 | Optimized vs non-optimized

The results when comparing the use of the FM synthesis lookup table optimisation described in Section 4.2 are compared against the use of the original trigonometric functions in Figure 9. This optimisation has a huge effect on the performance of the synthesis stage, for simple FM synthesis, this involves replacing two calls to the sin() function per sample with direct accesses into a wavetable. Back in Figure 7 it can be seen that the synthesis stage was one of the most consuming stages, therefore, this optimisation offers a 4X speedup in the synthesis stage for the default parameters. This significant improvement results in an overall 2X speedup for an optimisation affecting just the synthesis stage. This demonstrates the potential improvements context specific optimisations can make for intensive processing involved in the fitness stage.

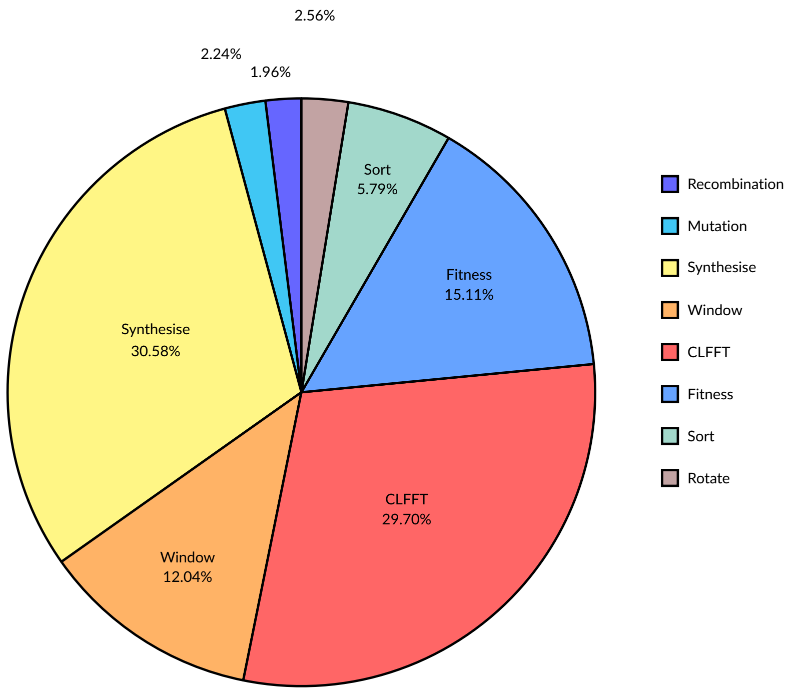


FIGURE 7 Ratio of execution time across stages in the GPU OpenCL implementation.
Parameters = Default **System** = High-end NVIDIA GeForce

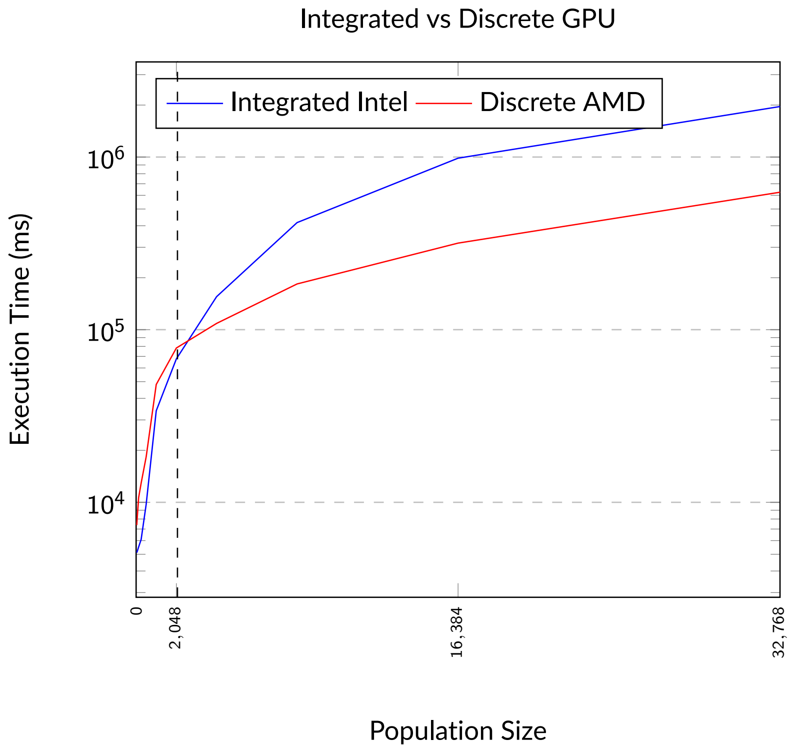


FIGURE 8 Execution time when scaling the population size for the GPU OpenCL implementation targeting the integrated and discrete GPU.
Parameters: Default, G = 1000, P+O = Scaled **System** = Mid-range Laptop

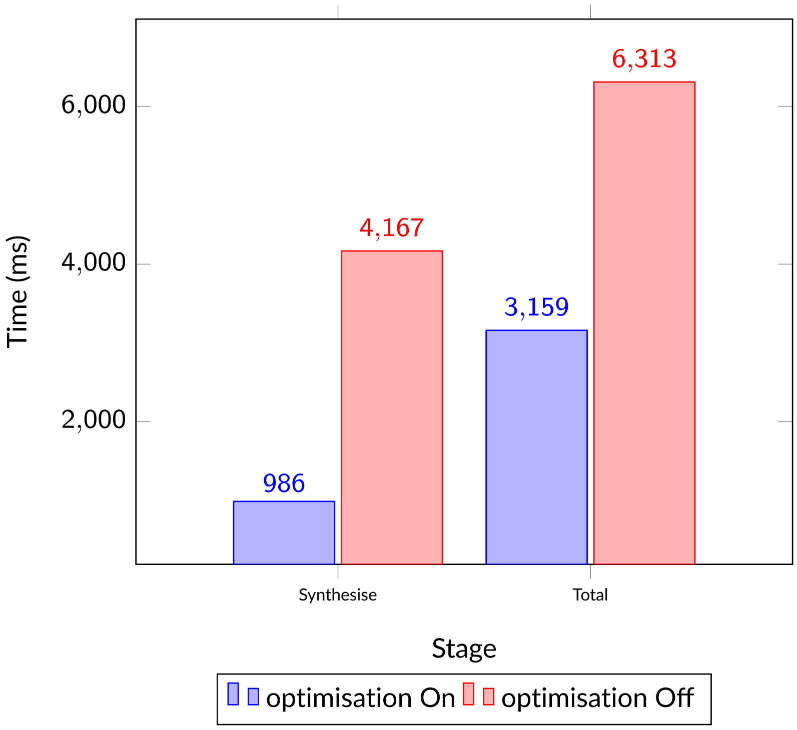


FIGURE 9 The total and synthesis stage execution time when the FM wavetable lookup optimisation is on and off. **Parameters** = Default **System** = High-end NVIDIA GeForce

6.7 | Population Scaling

Figure 10, provides a comparison between the implementations on the High-End NVIDIA system when scaling the population size. The execution time is plotted on a logarithmic scale in seconds. The CPU Serial version initially starts at a higher execution time at 1.87s compared to the GPU OpenCL version taking 0.89s. It is clear that as the population size scales, the execution time for the CPU versions is significantly higher. The GPU version's execution time increases gradually as its ability to process far more individuals in parallel is realised. At population size 16,384, the GPU version runs at approximately 10s, while the CPU OpenCL version records 55s, roughly 5X slower and the CPU Serial at 870s, 87X slower. The GPU continues to show it provides a speedup even at higher population sizes, beyond the 8192 size used in the default configuration. The GPU performance achieved brings the application considerably closer to a practical, real-time tool. Complex synthesisers that require considerably large population sizes to find accurate solutions will benefit further by using GPUs over CPUs.

6.8 | Audio Analysis Block Size Scaling

Figure 11 shows the total time of the GPU OpenCL application when scaling audio block size on the NVIDIA 2080 discrete GPU. The audio blocks size determines the number of audio samples processed and analysed on the GPU at a time. Decreasing the audio block size splits the target audio into further separate blocks for analysis. There are two reasons a smaller block size might be considered. First, the GPU memory would not be able to accommodate the larger blocks of audio. Second, this approach is advantageous if the audio being analysed is dynamic. However, decreasing the audio block size increases the number of dispatches to the GPU, which increases the communication overhead over the PCI bus. The overhead adds up considerably, resulting in severely reduced performance when the audio block size is below 128. Beyond block size of 128, the performance reaches a significantly improved state. Continuing to increase the audio block size to the target audio size has a less significant performance growth. These results show that although a smaller block size can be used, a size below 128 begins to have a significant impact on performance. Therefore, dynamic audio samples that involve frequent changes in timbre will be challenging to process with the current design and will need to be improved to support it.

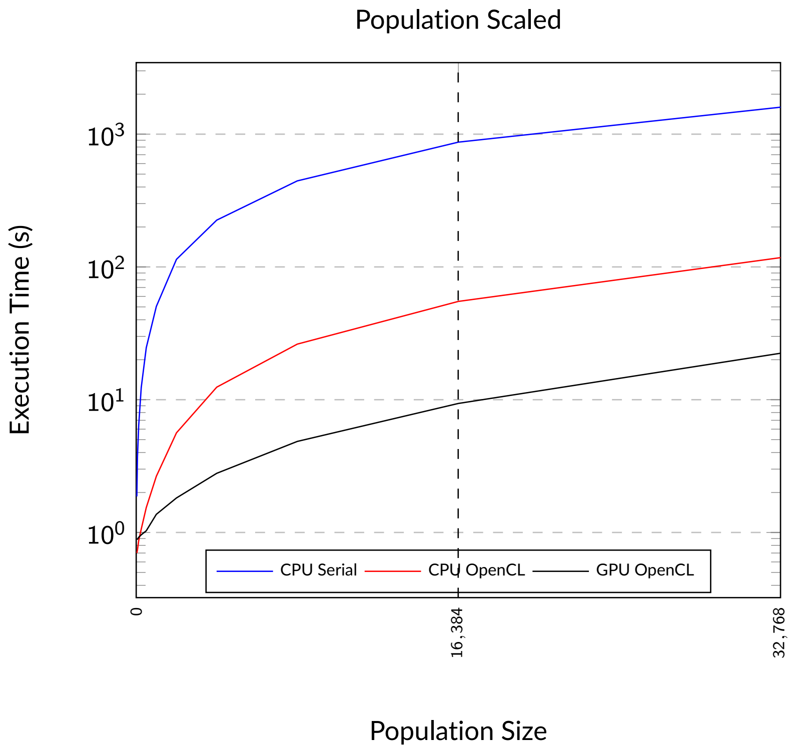


FIGURE 10 Execution time when scaling the population size for all implementations. **Parameters**: G = 1000, P+O = Scaled **System** = High-end NVIDIA GeForce

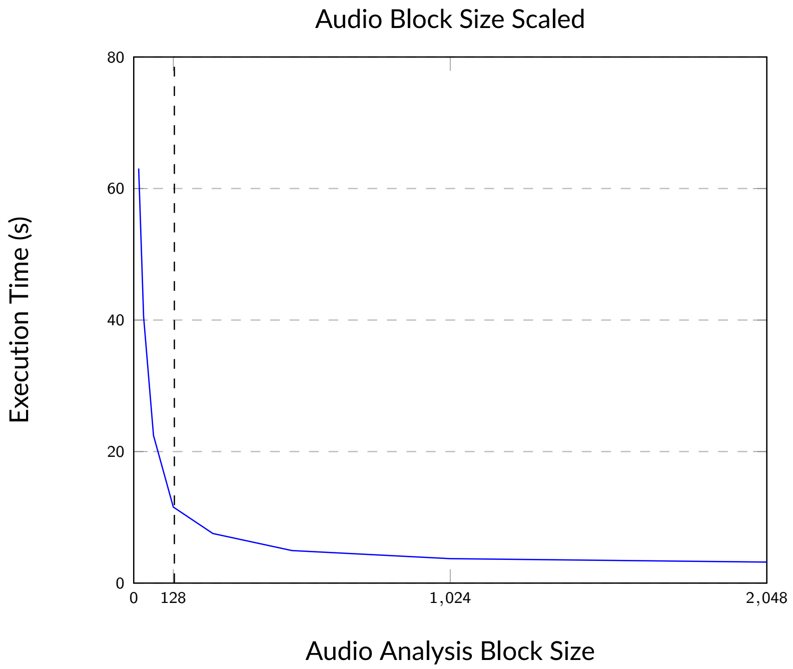


FIGURE 11 Execution time when scaling audio block size for the GPU OpenCL implementation. **Parameters**: Default, N = Scaled **System** = High-end NVIDIA GeForce

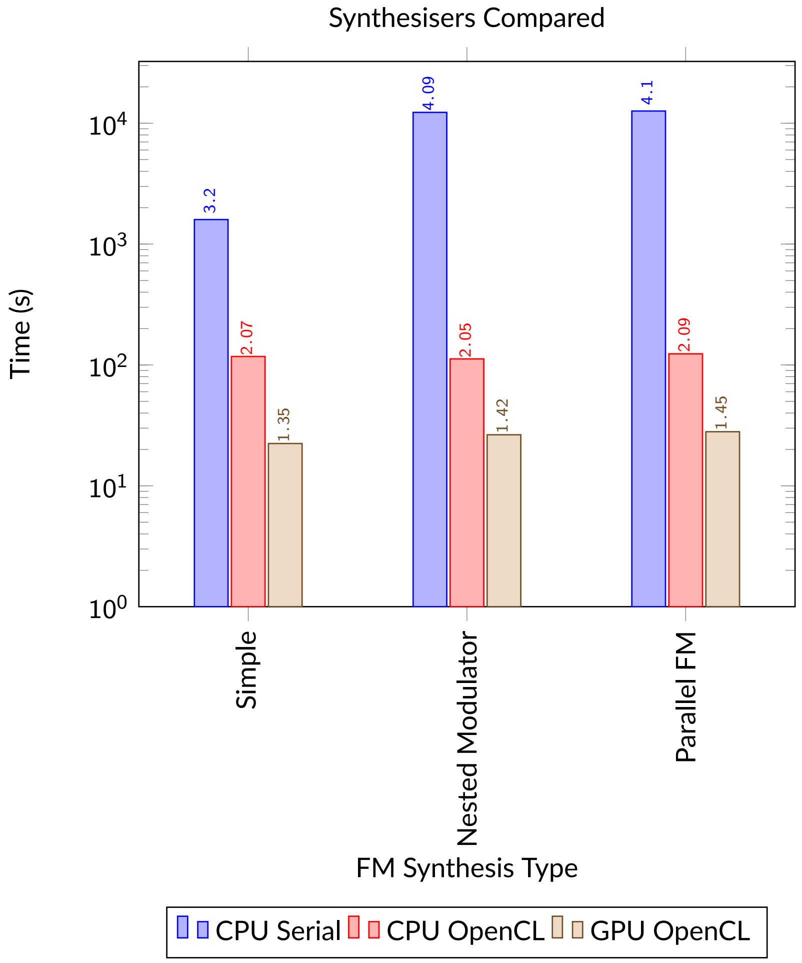


FIGURE 12 Execution time of three synthesis types for the CPU and GPU implementations.
Parameters: Default, P+O=32768 **System** = High-end NVIDIA GeForce

6.9 | Advanced FM Synthesisers

To demonstrate the scalability of the GPU optimised framework, two further FM synthesisers have been used in place of the simple FM. The first is nested modulator FM and the second is parallel FM synthesis. Both of these multi-operator FM synthesisers have been described in Section 2.1. Figure 12 presents the results of the simple, nested modulator and parallel FM synthesisers on all three implementations. Again, a large population size of 32768 has been used and the time in seconds is again plotted on a logarithmic scale. It can be seen for the CPU Serial implementation, both the nested modulator and parallel synthesisers take considerably longer with an additional 10699s and 11024s respectively. By contrast, the GPU OpenCL implementation only requires an additional $\approx$ 4s for the nested modulator and $\approx$ 6s for parallel FM. The results suggest that the GPU accelerated framework for handling the evolutionary computation for parameter matching supports more advanced forms of FM synthesis. Although this cannot be extrapolated to all possible arrangements of FM synthesis, for these two examples, it continues to improve performance over a naive serial implementation.

7 | CONCLUSION

This paper has presented the design for a GPU optimised algorithm for parameter matching for FM synthesis. The designs have been implemented as serial CPU, parallel CPU and parallel GPU versions in a benchmarking suite. The benchmarking suite is open-source and can be used to evaluate the performance of the implementations on different hardware systems. Benchmarking results were collected on various systems and the salient results were discussed. For the default parameter configuration on a high-end desktop, the GPU had a speedup of 128X over the serial CPU version and 8.88X over the parallel CPU version. This highlights the significant improvement potential when using parallel processing in general, but also the massively parallel architecture of the GPU in comparison to the CPU. The population size of the ES has a significant impact on the execution time of all implementations. However, the impact of the GPU version was significantly lower than the other implementations, suggesting that the GPU design proposed can be used to process larger population sizes more rapidly, increasing the usability of parameter matching as a suitable tool for music creators. The results show that the performance benefits apply to simple FM synthesis and extend to support more advanced arrangements such as nested modulator and parallel FM synthesisers. Another controllable GPU parameter is the data block size, this was scaled and was shown to harm the GPU processing time when block sizes below 128 are used. This is a weakness of the GPU design caused by the data transfer overhead between CPU and GPU. An optimised design needs to be adapted to better support this use-case.

ACKNOWLEDGMENTS

We would like to acknowledge the consistent support and insight from colleagues in the Computer Science Research Centre and the Creative Technologies Lab. This work was supported by a grant from the HSA Foundation and the UWE 50-50 PhD fund. Extended thanks to Jonathan Andrew and Professor Simon McIntosh-Smith at the University of Bristol for the guidance their research provided. Special thanks to Dr David Creasey for his discussions on FM synthesis.

References

1. Mitchell T. Automated evolutionary synthesis matching. *Soft Computing* 2012; 16(12): 2057–2070.

2. Owens JD, Luebke D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware. In: . 26. Wiley Online Library. ; 2007: 80–113.

3. Luebke D, Harris M, Govindaraju N, et al. GPGPU: general-purpose computation on graphics hardware. In: ACM. ; 2006: 208.

4. Rechenberg I. Cybernetic solution path of an experimental problem. *Royal Aircraft Establishment Library Translation 1122* 1965.

5. Mitchell TJ, Creasey DP. Evolutionary sound matching: A test methodology and comparative study. In: IEEE. ; 2007: 229–234.

6. Horner A. Nested modulator and feedback FM matching of instrument tones. *Speech and Audio Processing, IEEE Transactions on* 1998; 6: 398 - 409. doi: 10.1109/89.701371

7. Yee-King M, Roth M. A comparison of parametric optimisation techniques for musical instrument tone matching. In: Goldsmiths, University of London. ; 2011.

8. Das S, Suganthan PN. Problem definitions and evaluation criteria for CEC 2011 competition on testing evolutionary algorithms on real world optimization problems. *Jadavpur University, Nanyang Technological University, Kolkata* 2010: 341–359.

9. Horner A, Beauchamp J, Haken L. Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis. *Computer Music Journal* 1993; 17(4): 17–29.

10. Mitchell TJ. *An exploration of evolutionary computation applied to frequency modulation audio synthesis parameter optimisation*. PhD thesis. University of the West of England, 2010.

11. Lee KY, Yang FF. Optimal reactive power planning using evolutionary algorithms: a comparative study for evolutionary programming, evolutionary strategy, genetic algorithm, and linear programming. *IEEE Transactions on Power Systems* 1998; 13(1): 101-108. doi: 10.1109/59.651620

12. Macret M, Pasquier P, Smyth T. Automatic calibration of modified fm synthesis to harmonic sounds using genetic algorithms. 2012.

13. Lai Y, Jeng SK, Liu DT, Liu YC. Automated optimization of parameters for FM sound synthesis with genetic algorithms. In: Citeseer. ; 2006: 205.

14. Smith BD. Play it Again: Evolved Audio Effects and Synthesizer Programming. In: Springer. ; 2017: 275–288.

15. Yee-King MJ, Fedden L, d'Inverno M. Automatic programming of VST sound synthesizers using deep networks and other techniques. *IEEE Transactions on Emerging Topics in Computational Intelligence* 2018; 2(2): 150–159.

16. Lengyel J, Reichert M, Donald BR, Greenberg DP. Real-time robot motion planning using rasterizing computer graphics hardware. *ACM SIGGRAPH Computer Graphics* 1990; 24(4): 327–335.

17. Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro* 2008; 28(2): 39–55.

18. McClanahan C. History and evolution of gpu architecture. *A Survey Paper* 2010; 9.

19. Rapaport DC. GPU molecular dynamics: Algorithms and performance.. *arXiv: Computational Physics* 2020.

20. Renney H, Gaster BR, Mitchell T. OpenCL vs: Accelerated finite-difference digital synthesis. In: ; 2019: 1–11.

21. Desell T, Waters A, Magdon-Ismail M, et al. Accelerating the MilkyWay@Home Volunteer Computing Project with GPUs. In: Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J., eds. *Parallel Processing and Applied Mathematics*Springer Berlin Heidelberg; 2010; Berlin, Heidelberg: 276–288.

22. Turian J, Shier J, Tzanetakis G, McNally K, Henry M. One Billion Audio Sounds from GPU-enabled Modular Synthesis. *arXiv preprint arXiv:2104.12922* 2021.

23. Chowning J, Bristow D. FM theory and applications. *By Musicians for Musicians. Tokyo: Yamaha Music Foundation* 1986.

24. Horner A. A comparison of wavetable and FM parameter spaces. *Computer Music Journal* 1997; 21(4): 55–85.

25. Arabas J, Michalewicz Z, Mulawka J. GAVaPS-a genetic algorithm with varying population size. In: IEEE. ; 1994: 73–78.

26. Chowning JM. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society* 1973; 21(7): 526–534.

27. Roth M, Yee-King M. A comparison of parametric optimization techniques for musical instrument tone matching. In: Audio Engineering Society. ; 2011.

28. Fukuda Y. *Yamaha DX7 digital synthesizer*. Music Sales Corp . 1985.

29. Albano J. An Overview Of Logic Pro X's Powerful Synths. https://macprovideo.com/article/audio-software/an-overview-of-logic-pro-x-s-powerful-synths; 2016. Accessed: 2020-05-03.

30. Horner A. Nested modulator and feedback FM matching of instrument tones. *IEEE Transactions on Speech and Audio Processing* 1998; 6(4): 398–409.

31. Beyer HG, Schwefel HP. Evolution strategies–A comprehensive introduction. *Natural computing* 2002; 1(1): 3–52.

32. Patel JK, Read CB. *Handbook of the normal distribution*. 150. CRC Press . 1996.

33. Frigo M, Johnson SG. FFTW: An adaptive software architecture for the FFT. In: . 3. IEEE. ; 1998: 1381–1384.

34. Harris FJ. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE* 1978; 66(1): 51–83.

35. Schloss J. cooley_tukey. 2019. Accessed: 2019-10-10.

36. Beauchamp JW, Horner A. Error metrics for predicting discrimination of original and spectrally altered musical instrument sounds. *The Journal of the Acoustical Society of America* 2003; 114(4): 2325–2325.

37. Davidson A, Tarjan D, Garland M, Owens JD. *Efficient parallel merge sort for fixed and variable length keys*. IEEE . 2012.

38. Group KOW, others . The OpenCL Spec V2. 0. 2013.

39. Gaster B, Howes L, Kaeli DR, Mistry P, Schaa D. *Heterogeneous computing with openCL: revised openCL 1.* Newnes . 2012.

40. Corporation N. NVIDIA Fermi Compute Architecture Whitepaper. 2009.

41. Hestness J, Keckler SW, Wood DA. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In: IEEE. ; 2014: 150–160.

42. Mei G, Tian H. Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *SpringerPlus* 2016; 5(1): 1–18.

43. Micikevicius P. GPU performance analysis and optimization. In: . 3. ; 2012.

44. Kim DH. Evaluation of the performance of GPU global memory coalescing. *Evaluation* 2017; 4(4).

45. Natarajan B. clfft. https://clmathlibraries.github.io/clFFT; 2013. Accessed: 2019-10-21.

46. Bertrand M. The CORDIC method for faster sin and cos calculations. *C Users Journal* 1992; 10(11).

47. Raghunath KJ, Rambaud MM. Sine/cosine lookup table. 1999. US Patent 5,937,438.

48. Renney H, Gaster BR, Mitchell TJ. There and Back Again: The Practicality of GPU Accelerated Digital Audio. 2020.

How to cite this article: Renney H,, Gaster BR,, Mitchell T. (2016), Survival of the Synthesis - GPU Accelerating Evolutionary Sound Matching, *Concurrency Computat Pract Exper*, *2021;33:N/A*.

OpenCL vs: Accelerated Finite-Difference Digital Synthesis

Harri Renney
harri2.renney@live.uwe.ac.uk
Computer Science Research Centre
University of West of England
Bristol, UK

Benedict R. Gaster
benedict.gaster@uwe.ac.uk
Computer Science Research Centre
University of West of England
Bristol, UK

Tom Mitchell
tom.mitichell@uwe.ac.uk
Creative Technologies Lab
University of West of England
Bristol, UK

ABSTRACT

Digital audio synthesis has become an important component of modern music production with techniques that can produce realistic simulations of real instruments. Physical modelling sound synthesis is a category of audio synthesis that uses mathematical models to emulate the physical phenomena of acoustic musical instruments including drum membranes, air columns and strings. The synthesis of physical phenomena can be expressed as discrete variants of Newton's laws of motion, using, for example, the Finite-Difference Time-Domain method or FDTD.

FDTD is notoriously computationally expensive and the real time demands of sound synthesis in a live setting has led implementers to consider offloading to GPUs. In this paper we present multiple OpenCL implementations of FDTD for real time simulation of a drum membrane. Additionally, we compare against an AVX optimized CPU implementation and an OpenGL version that utilizes a careful mapping to the GPU texture cache. We find using a discrete, laptop class, AMD GPU that for all but the smallest mesh sizes, the OpenCL implementation out performs the others. Although, to our surprise we found that optimizing for workgroup local memory provided only a small performance benefit.

CCS CONCEPTS

- **Computing methodologies** → **Parallel programming languages**; - **Software and its engineering** → *Compilers*.

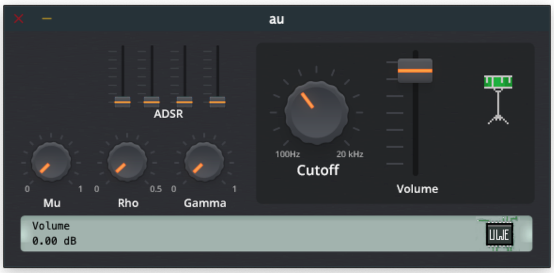
KEYWORDS

Finite-Difference, SIMD, OpenCL, OpenGL

ACM Reference Format:
Harri Renney, Benedict R. Gaster, and Tom Mitchell. 2019. OpenCL vs: Accelerated Finite-Difference Digital Synthesis. In *Proceedings of IWOCL '19: The International Workshop on OpenCL (IWOCL '19)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

IWOCL '19, May 13-15, 2019, Boston, USA

ACM ISBN 978-1-4503-9999-9/18/06...$15.00
https://doi.org/10.1145/1122445.1122456



Volume
0.00 dB

Figure 1: Audio Unit (AU) plugin

1 INTRODUCTION

The physical synthesis of sound is the general process of using mathematical models to simulate a physical source of sound. The technique of simulating sound using mathematical models, although not the earliest example, was first made popular by Karplus and Strong, using a method of physical modelling synthesis that circulates a short waveform through a filtered delay line to simulate the sound of a hammered or plucked string [12]. The algorithm was later extended by David and Smith [13] and remains in use today, due in part to its low computational footprint.

While the Karplus-Strong algorithm remains popular in the domain of real time synthesis, it does not accurately represent the way in which vibrations propagate through a medium, e.g. a drum membrane, and alternative models have been proposed that address these shortcomings[1]. For example, finite difference approximation is a common method to simulate the movement of waves through physical mediums, presented very early on in the area of acoustics and synthesis, e.g. Hiller and Ruiz studied using finite difference methods for sound synthesis in the early 1970s [10].

To simulate vibrations moving through a material, we can utilize Newton's laws of motion, describing the movement using Ordinary Differential Equations (ODEs). For these to be implemented as a discreet algorithm, ODEs are expressed as Finite Discreet Time-Domain (FDTD) equations. FDTDs are used as the basis of numerous physical modelling efforts that seek to digitally synthesize the sounds of, for example, drum membranes [16], wind instruments [3] and strings [8].

[1] It is worth noting that in the end, most sound synthesis is performed in the context of music composition and as such the need to sound like a "real" drum or some other form of instrument is subjective.

Of course, it is well known that the real-time simulation of FDTD is hard, it requires a large amount of floating point computation. Consider, for example, the case of simulating a drum membrane with a mesh of 32x32 at an audio sample rate of 48kHz. Each point in the domain requires 40 floating point instructions per sample and thus approximately 2 Giga FLOPS of compute for real time synthesis. Of course, in a real audio application, any particular sound engine must compete for resources with, synthesis, effects, mixing, and so on. To this end, acceleration of FDTDs on GPUs has been proposed as a method to offload the simulation of drum membranes and other physical models of musical phenomena. For example, Sosnick and Hsu describe a straightforward implementation using NVIDIA's Cuda [16], while Zappi et al use OpenGL [20]. In this paper we again step into the breach to use GPUs to accelerate FDTD, for real time audio synthesis, initially utilizing OpenCL[2] [9], but we then go further presenting a comparison of implementations ranging from a serial CPU implementation, an SOA AVX variant, a recreation of Zappi et al's OpenGL implementation, and two OpenCL versions, one using only global memory, and one using workgroup local memory. We find that on AMD hardware, in all but the smallest grids, OpenCL outperforms the other implementations, and the use of workgroup local memory provides little to zero benefit.

We have implemented two variants of our drum simulation. The first implementation uses our OpenCL and CPU implementations and is provided as a Apple Audio Unit (AU) [1] plugin, that can be loaded into a Digital Audio Workstation (DAW) (e.g. Ableton Live, Logic Pro, and so fourth). A screen shot of the plugin is given in Figure 1 and includes controls that change properties of the drum membrane, such as the speed at which sound travels through the material. While not directly relevant to how the FDTD was optimized, the focus of this paper, it played an important role within the context of the work as a whole, enabling practicing musicians to utilize the drum within their standard work-flow and to provide feedback on the sound quality—it is of little use to provide real time synthesis for a drum that sounds terrible!

The second implementation is focused on providing a simple test framework in which the OpenCL, OpenGL, and CPU variants can be compared. The simulations are controlled from (JSON) configuration files and are fully automated. For the most part the framework for each implementation is the same, however, for the OpenGL version we also support a simple visualization of the drum membrane.

The OpenGL version's ability to generate visual feedback is shown in Figure 2. This figure simply takes the pressure points of the current implementation and uses it to calculate a colour gradient, using an additional `drawcall`. The red dot is the positioning of the microphone and is the point that is sampled for the audio output. The yellow square is the excitation point, i.e. the point where an excitation function is fed to the membrane—in general, it would not be placed in the centre of the mesh, but for simplicity it is located there for demonstration.

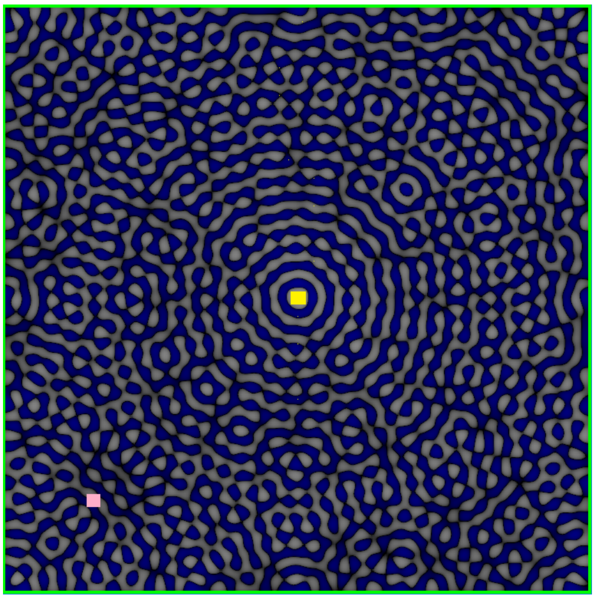


Figure 2: OpenGL Visual rendering of 2D membrane

We conclude this introduction by outlining the remainder of the paper:

1. Section 2 provides a short overview of related and existing work;
2. Section 3 introduces the FDTD equations, describing implementation as pseudo code, and then details the CPU, OpenCL, and OpenGL implementations;
3. The benchmark results for the different implementations are presented in Section 4; and
4. Finally, Section 5 concludes and provides pointers to future work.

2 RELATED WORK

To our knowledge there are no OpenCL implementations of FDTD that have been specifically developed for digital audio synthesis. As we would expect, there have been a selection of implementations of 3D FDTDs. For example, in the context of electromagnetic wave interaction, Cannon and Honary present an implementation in OpenCL [7]. They demonstrate good speedup, including multiple GPUs, however, unlike our work they fail to consider alternatives, such as OpenGL or AVX and focus on HPC style platforms and highend GPUs (Nvidia Tesla M2075), utilizing a large number of SIMD units and DDR4 memory. It is worth noting that due to the problem they are tackling, the size of meshes they consider are much larger than those we present, which has the potential to provide an easier context for acceleration, when comparing to single threaded CPU performance.

Although FPGAs are a good way of processing FDTD grids [18], the focus of this work has been towards commodity

[2] Throughout this paper we use OpenCL as shorthand for OpenCL 1.2 and do not consider OpenCL 2.x.

hardware, in particular, live digital musician will tend towards using a medium to highend laptop, with an external audio device to handle input and output to the analog domain. To date easy to access FPGAs is not accessible on a commodity scale within this domain.

Returning the focus to 2D FDTDs for digital audio synthesis for GPUs a number of researchers have studied this area, for example [4, 6, 16, 20]. In particular, Sosnick and Hsu [16] implement a simple 2D membrane simulation using a FDTD written in CUDA. However, there results are limited, testing only on small meshes and single buffer size. They also to not consider other optimized implementations, such as AVX CPU or other GPU programming models such as OpenGL or OpenCL.

Zappi et al [20] implement an FDTD simulation of a drum head model very similar to ours. They use a novel approach to packing the previous, current, and next step simulation data within a single RGBA texture. The OpenGL implementation used in the benchmarks presented in Section 4 uses a variant of their design. While it performs well it fails, by a small amount, to out perform our OpenCL implementation. This leaves us to hypothesize that while single texture encoding is interesting, it might in fact lead to a less cache friendly algorithm.

3 IMPLEMENTATION

In this section we describe the FDTD algorithm, firstly as equations, then as pseudo code. This is then followed by details of each implementation. The example used to demonstrate the implementation is the OpenCL global memory kernel, and, although the other implementations differ in details regarding the particular target language, they are for the most part similar. In the case were they differ, e.g. using a packed texture for the OpenGL implementation a discussion is included to outline important aspects. The full source code for each implementation used within the Section 4, outlining the benchmark results, can be found here [15].

3.1 The FDTD Algorithm

In this subsection, the numerical algorithm used for modelling the drum membrane is described. It follows the discretization of the standard 2D wave propagation equation [17].

$$p^{n+1} = \frac{2p^n + \left(\mu - 1\right) p^{n-1} + \alpha \left(p_l + p_r + p_u + p_d - 4p^n\right)}{\mu + 1} \tag{1}$$

$$P_{L,R,U,D} = \begin{cases} p^n \gamma & \text{if } n \text{ boundary} \\ p^n_{l,r,u,d} & \text{if } n \text{ free} \end{cases} \tag{2}$$

where:

- p^{n+1} is the pressure point to be calculated for the next time step, $n + 1$.
- p^n is the pressure point of current time step, n.
- p^{n-1} is the pressure point of the previous time step, $n - 1$.
- μ is the damping/absorption coefficient of the modelled material. ($0.0 < \mu < 1.0$), for all values of μ.
- α is the propagation factor. ($\alpha \leq 0.5$), for all values of α.
- $p_{l,r,u,d}$ are the pressure points for the neighbouring values of the centre point currently under consideration.
- γ is the centre points boundary gain. This is the degree at which the pressure is reflected back into the grid. ($0.0 < \gamma < 1.0$), for all values of γ.

```
for i = 0 to bufferSize
  for row = 1 to gridHeight
    for column = 1 to gridWidth
      centrePoint = getPoint(row,column)
      if(centrePoint == boundary)
        neighbours = calculateBoundary(centrePoint)
      else
        neighbours = getNeighbours(centrePoint)
    compute(centrePoint, neighbours)
    end for
  end for
  rotateGrids()
end for
```

Figure 3: Pseudocode for FDTD membrane simulation.

This equation is used to simulate wave propagation across the 2D surface when applied to all grid points. Every time step, the equation is used to calculate the next pressure value of the currently considered grid point from the current, previous and neighbouring pressure values.

The neighbouring values are determined by the centre points boundary value (See equation 2). If the centre is not a boundary point, then the actual neighbouring pressure points are taken for $P_{L,R,U,D}$. If it is a boundary point, then the neighbour pressure values are not used and instead the centre pressure point multiplied by the boundary gain γ is used in place of the neighbour values in the equation.

α, the propagation factor determines the speed at which sound passes through the medium. It is formed from the speed of sound, the sample rate and the size of each grid point. Therefore when modelling some material, the size of the grid and the sample rate affect the speed at which waves are simulated to pass through the material if the propagation factor is not adjusted.

3.2 Pseudo Code

The pseudocode for implementing the previous equations is given in Figure 3 and outlines the sequential method of calculating the FDTD grid. This code is a straightforward way to compute a 2D FDTD grid, of any size. It works by visiting each point in the grid and calculating the next pressure value using the compute function. This basically applies equation 3.1 to generate the next time step pressure value. The grid of $n + 1$ is updated with the new value.

Once the whole grid has advanced one time step by setting the centre point to the new computed point, the grids are advanced too, by rotating the grids. Aligning them correctly ready for the next time step.

3.3 OpenCL Kernel

Figure 4 is the source code for the OpenCL global memory variant of our FDTD implementation. For ease of presentation the local memory variant is not presented in this paper, but the interested reader can find the code for the all the implementations on our Gitlab repository [15].

Note how the rotation index is used to define the FDTD pointers to each grid. By incrementing the index, the address to which the pointers are set to shifts along to the next grid, simulating the advancement of a time-step. It is important to note that the rotation index is not incremented within the kernel, but in the host application between kernel calls. As a consequence, an explicit synchronization barrier is necessary between each OpenCL "ndrange" dispatches, thus, kernel calls cannot be batched. An alternative would be to utilize a kernel to increment the rotation index in between FDTD kernels; this would allow batching and avoid host side synchronization and copying of the rotation index. To date we have not felt this necessary, but it is likely that as we optimize further this will become necessary.

3.4 Implementations

This section provides an overview of each of the different implementations. The implementations, which are outlined in the following subsections, all share an FDTD grid class that stores the locations of the excitation and listener points, along with the pressure and boundary grids as a flattened 2D Structure of Arrays (SOA) data structure. Although the underlying kernels for each of the different implementations vary in design, with the goal of utilizing the different optimizations opportunities provided by each programming model.

3.4.1 CPU Serial. The serial implementation works by visiting each grid point and calculating the new pressure value using the equation 1, then moving onto the next cell sequentially. The whole grid must be calculated before one sample can be obtained. A few optimizations were used to avoid needless computation like cache alignment, avoiding copies and redundant calculations.

Cache alignment is achieved by using flattened 2D arrays of the grids. This ensures the next item in the array is usually held in the same cache line, even at the ends of the grid rows. For each time step, the pressure grids need to increment along in time. Take a look at Figure 5. After the newly calculated grid of N+1 pressure values is complete, a pointer which determines the current pressure grid addresses the N+1 grid. All the pointers shift forward one to correctly address the new set of pressure grids. This is done rather than copying all the data between the grids which would be highly inefficient.

Redundant calculations are removed including only calculating $\mu - 1$ once for calculating the next pressure point. μ can change, therefore it should still be calculated once per buffer fill, but it is not necessary to calculate every cell visited.

3.4.2 CPU AVX. The AVX optimized CPU implementation follows after most of what the serial version does. However, it vectorizes the grids for processing. Using AVX SIMD intrinsics, the grid can be vectorized into vectors of four floats. These vectors are processed and stored, using Structure of Arrays, in parallel. Intel's compiler intrinsics [2] were used to do an explicit vectorization of FDTD computation, rather than using compiler pragmas or a C++ SIMD library, for example.

Although the grid calculations are applied universally, the checks for boundary, excitation and listener points cause each element in the vectors to be checked, often using shuffles. This can likely be improved, with further considerations for packing certain data bits, but we leave this to further work.

Additionally, in future work we intend to extend the SIMD vectorization to support AVX-256/512. This would effectually double the vector size and therefore in ideal circumstances, would result in doubling of performance.

3.5 OpenCL global memory kernel

The OpenCL versions start by initializing the FDTD grid and the OpenCL configuration on the CPU. Then, when a buffer of samples is to be computed, an excitation buffer is loaded onto the GPU. Every iteration, the kernel is called which calculates the next time step and produces an output sample. The output samples remain on the GPU in a sample buffer. Only once the buffer is full is it read back by the host application on the CPU to minimise transfer overheads associated with passing data back and forth.

As with the serial implementation described earlier, the method for rotating pressure grids with pointers is also used here. At each time step, the GPU uses an index value, incremented by the CPU, to determine which grids the pointers address. In the OpenCL global version, the grids are held in the GPUs global memory and no local memory caching is performed.

3.5.1 OpenCL workgroup local kernel. An OpenCL version almost identical to the previous one was developed, but using the workgroup local memory to store the current pressure grid. In the kernel before any calculations are made the workitems load the current pressure value into a local grid accessible by all workitems in the same workgroup. This means when the neighbouring values of the centre point are needed, they can be fetched from the local grid which has shorter access times than the global grid. There are cases on the edge of workgroups where a work item will need to access a neighbour outside the workgroup, see figure 6. Therefore, conditional checks are made and if the neighbour is outside the workgroup, it will need to be fetched from global memory. This technique has been used in optimized convolution kernels, see for example [14]. These conditionals are suspected to impact any performance gained from using the local memory.

```
__kernel
void ftdtCompute(__global float* gridOne, __global float* gridTwo, __global float* gridThree,
  __global float* boundaryGain, int samplesIndex, __global float* samples, __global float* excitation,
  int listenerPosition, int excitationPosition, float propagationFactor,
  float dampingFactor, int rotationIndex) {
    // get index for current and neighbouring nodes
    int ixy = (get_global_id(1)) * get_global_size(0) + get_global_id(0);
    int ixMy = (get_global_id(1)-1) * get_global_size(0) + get_global_id(0);
    int ixPy = (get_global_id(1)+1) * get_global_size(0) + get_global_id(0);
    int ixyM = (get_global_id(1)) * get_global_size(0) + get_global_id(0)-1;
    int ixyP = (get_global_id(1)) * get_global_size(0) + get_global_id(0)+1;

    // determine each buffer in relation to time from a rotation index//
    __global float* nMOne;   __global float* n;  __global float* nPOne;
    if(rotationIndex == 0) {
      nMOne = gridOne;
      n = gridTwo;
      nPOne = gridThree;
    } else if(rotationIndex == 1) {
      nMOne = gridTwo;
      n = gridThree;
      nPOne = gridOne;
    } else if(rotationIndex == 2) {
      nMOne = gridThree;
      n = gridOne;
      nPOne = gridTwo;
    }
    // initialize pressure values//
    float centrePressureNMO = nMOne[ixy];
    float centrePressureN = n[ixy];
    float leftPressure;  float rightPressure;  float upPressure; float downPressure;

    if(boundaryGain[ixy] > 0.0) {
      leftPressure = n[ixy] * boundaryGain[ixy];
      rightPressure = n[ixy] * boundaryGain[ixy];
      upPressure = n[ixy] * boundaryGain[ixy];
      downPressure = n[ixy] * boundaryGain[ixy];
    } else {
      leftPressure = n[ixMy];
      rightPressure = n[ixPy];
      upPressure = n[ixyM];
      downPressure = n[ixyP];
    }
    // calculate next pressure value
    float newPressure = 2 * centrePressureN;
    newPressure += (dampingFactor - 1.0) * centrePressureNMO;
    newPressure += propagationFactor * (leftPressure + rightPressure +
                                      upPressure +  downPressure - (4 * centrePressureN));
    newPressure *= 1.0 / (dampingFactor + 1.0);

    // if the cell is the listener position, sets the next sound sample in buffer to value contained here
    if(ixy == listenerPosition) {
       samples[samplesIndex] = n[ixy];
    }
    if(ixy == excitationPosition) { // if the position is an excitation...
      // input excitation value into point. Then increment to next excitation in next iteration.
      newPressure += excitation[samplesIndex];
    }
    // update grid plus one
    nPOne[ixy] = newPressure;
}
```

Figure 4: OpenCL FDTD Kernel

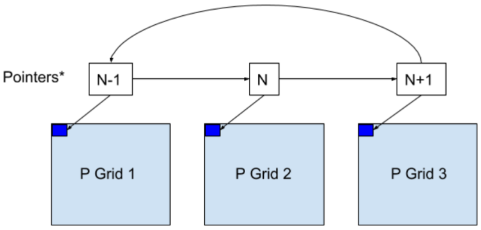


Figure 5: Pressure grid rotation achieved using pointers

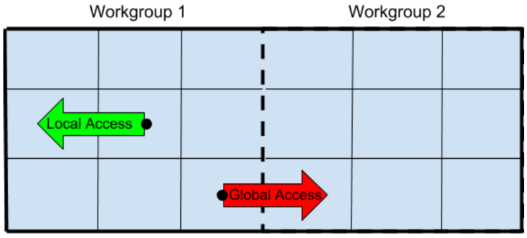


Figure 6: Neighbour access within workgroup and across workgroups.

3.6 OpenGL

At initialization, the same FDTD grid object as the earlier methods is created, but then, translated into a graphical context in order to be understood by OpenGL. This translation involves forming a texture buffer, which is loaded in and out of the GPU and processed according to the defined GLSL shaders. This formats the grid data into RGBA channels in the graphics fragment shader, where the FDTD calculations takes place.

An advantage of using OpenGL is that a render shader program can be used for visualizing the FDTD grid in its current state. Of course, OpenCL/OpenGL interop could be used, but this has a performance cost, due to different driver stacks. An alternative approach that we are planning to investigate is to use Khronos' Vulkan [19], with a combination of compute and graphic shaders.

4 RESULTS

In this section we outline the results obtained when executing the serial CPU, AVX CPU, OpenCL, and OpenGL implementations of the 2D FDTD simulation of a simple drum membrane. The results compare a variety of different mesh sizes, audio buffer sizes, and, for OpenCL, different workgroup optimizations, including workgroup size and workgroup local memory utilization. All speedup measurements are calculated in relation to the results of the naive, serial CPU implementation. Although it might not seem fair to compare massively parallel processing to a serial one, this was done to show the instant benefit by considering using a GPU for processing these kinds of problems. Clearly with a little more work, the CPU can still be improved. This can be seen to an extent with the CPU AVX version.

System Specifications	
CPU	Intel Core i7-8550U 4 Cores 1.99GHZ
GPU	AMD Radeon 530 with 2GB GDDR5
CPU RAM	8GB 2400MHz DDR4

Table 1: Laptop specification used for benchmarking

	OpenCL Global	OpenCL Local
Workgroup Size	time (ms)	time (ms)
4x4	136.072666	149.841333
8x8	46.170966	49.19716
16x16	46.5292	45.363366

Table 2: Mean buffer compute time calculated over 100 filled buffers. Buffer size = 512, Grid Size = $256x256$

The benchmarks are run on a single test machine, which is a mid-range PC laptop, with a discrete AMD GPU and Intel i7. The complete specification of the test setup is given in Table 1.

Table 2 compares the different workgroup sizes configurable in OpenCL. The workgroups represent the number of work items that can be processed concurrently. In this case, work items are equivalent to computation of each pressure point in the grid.

Table 2 is using a larger grid size of $256x256$. There is little difference between the results of the smaller grid dimensions (e.g. $8x8$ and $16x16$) as the $16x16$ workgroup size does not make use of the increased capacity of concurrent work items.

The maximum workgroup size used for benchmarking the system's GPU is 256 ($16x16 = 256$), corresponding to an equal axis grid arrangement of $16x16$. Our initial tests concluded that smaller arrangements were less efficient so the largest workgroup dimensions were used for all the following tests $16x16$.

Figure 7 plots the compute time in milliseconds for each implementation with a buffer size of 512 samples.

Here a collection of different sample buffer sizes were tested. Taking Figure 11 specifically, it can be seen that although the AVX implementation starts with a good speedup. It cannot handle the increasing grid size and plateaus at 1-2 times speedup. The GPU versions are slower for processing smaller grids. This is likely due to the transfer overhead between CPU-GPU communication. Although slower for small grids, the GPU versions outperform the CPU versions as the grid scales. The OpenCL versions continue to speedup as it scales. Interestingly, the OpenGL version does not reliably increase. To date we have not established why this is the case, but hypothesize that it might well be down to the constraints and perhaps redundant steps involved in the graphics pipeline. More likely the smart single texture encoding proposed by Zappi et al [20] might actually be causing the slow down, due

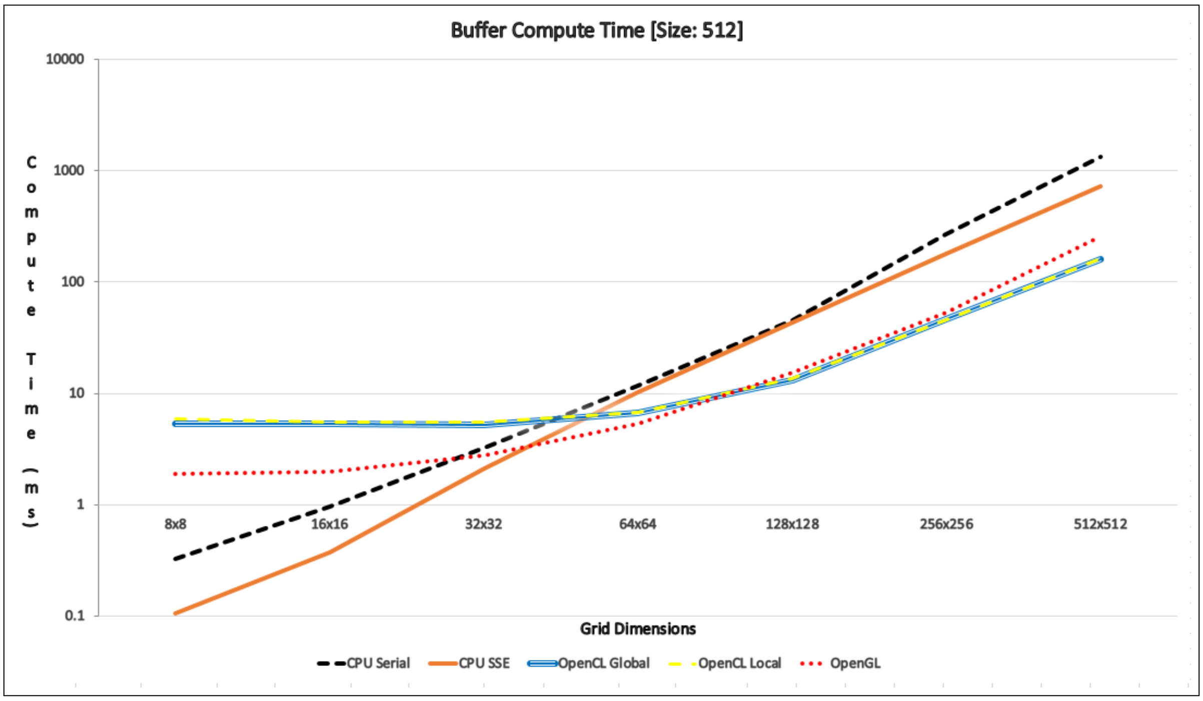


Figure 7: Average compute time measured for buffer size 512

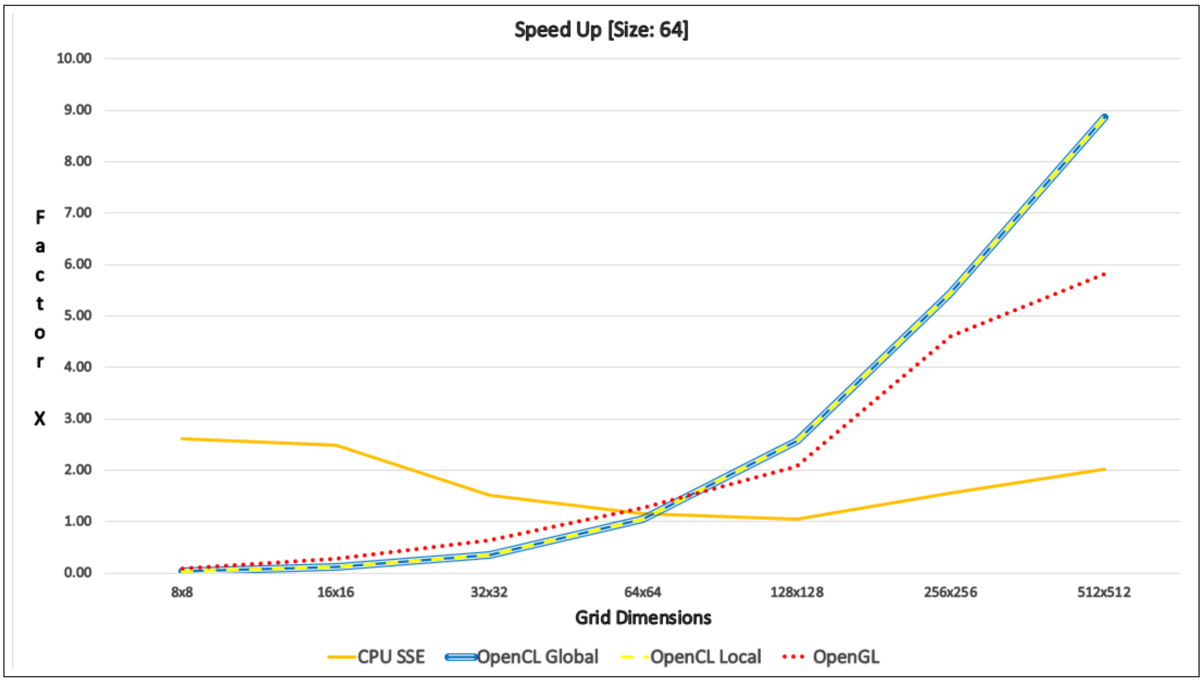


Figure 8: Speedup measured for buffer size 64

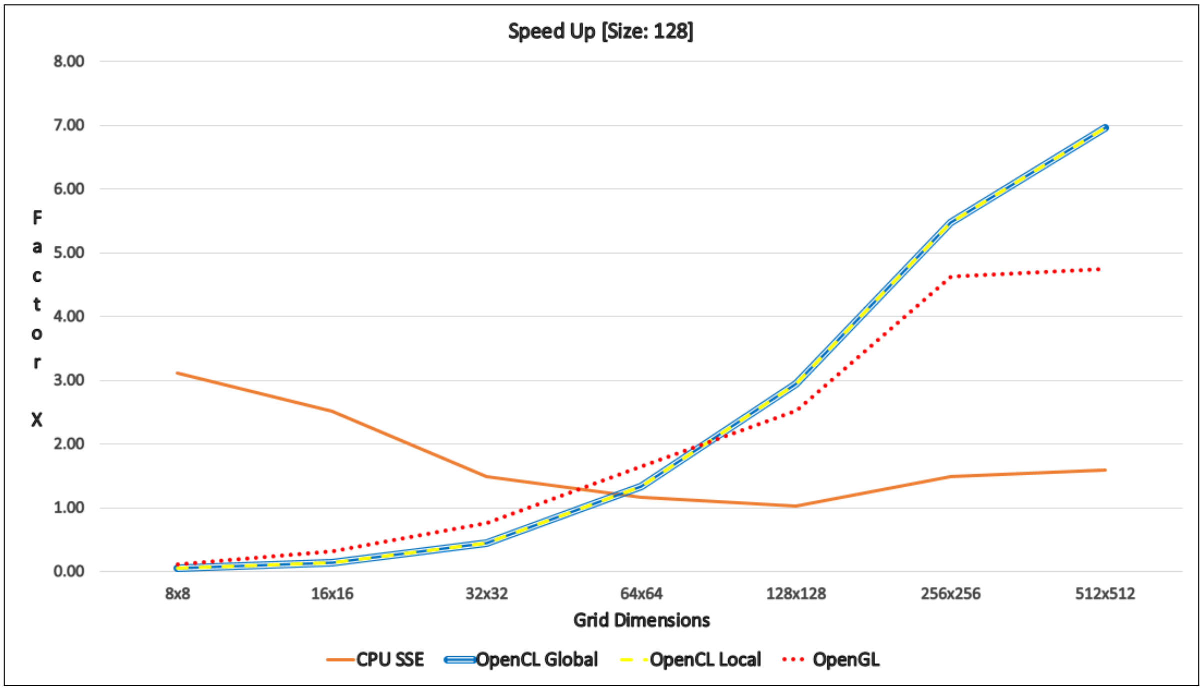


Figure 9: Speedup measured for buffer size 128

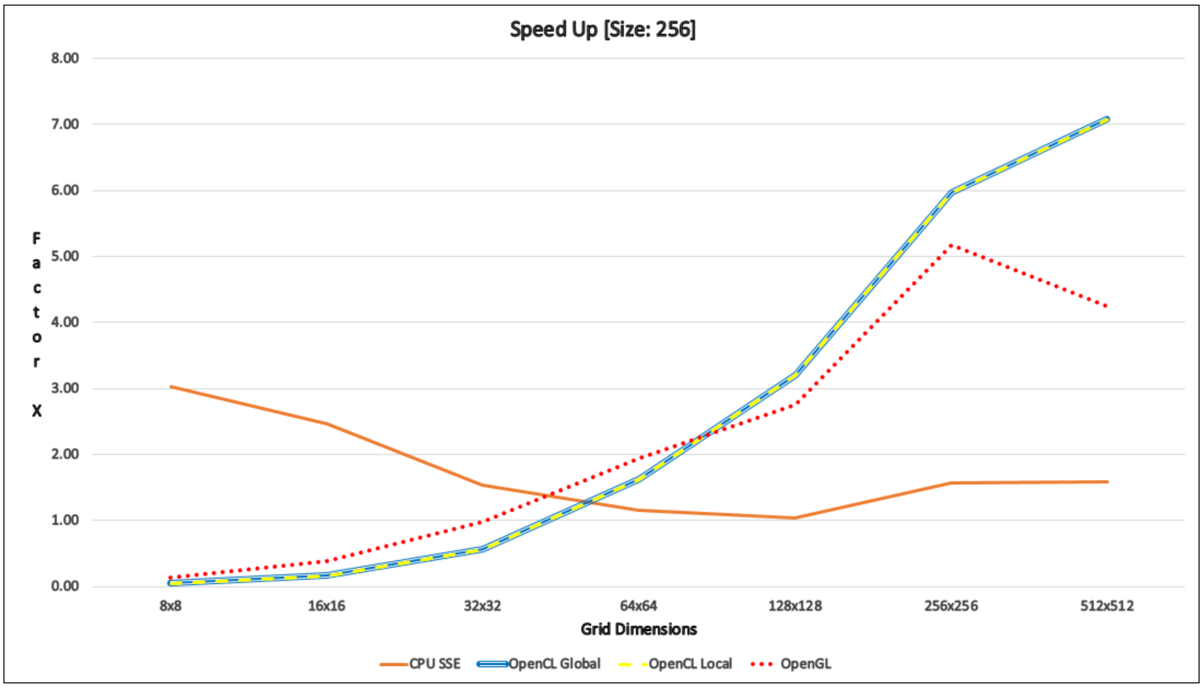


Figure 10: Speedup measured for buffer size 256

	CPU Serial		**CPU SSE**		**OpenCL Global**		**OpenCL Local**		**OpenGL**	
Dimensions	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
8x8	0.3247	1.0	0.105525	3.08	5.30303	0.06	5.96503	0.06	1.899273	0.17
16x16	0.954201	1.0	0.377217	2.53	5.29219	0.18	5.48204	0.18	1.97544	0.48
32x32	3.27136	1.0	2.12098	1.54	5.230725	0.63	5.5009	0.63	2.75727	1.19
64x64	11.7197	1.0	10.1995	1.15	6.62889	1.77	6.7361	1.77	5.337273	2.20
128x128	45.8261	1.0	43.01075	1.07	13.21315	3.47	13.672	3.47	15.256833	3.00
256x256	271.803	1.0	178.2885	1.52	46.5482	5.84	45.4415	5.84	53.18915	5.11
512x512	1327.78	1.0	726.5175	1.83	162.295	8.18	160.995	8.18	257.278	5.16

Table 3: Mean buffer compute time calculated over 100 filled buffers. Buffer size = 512

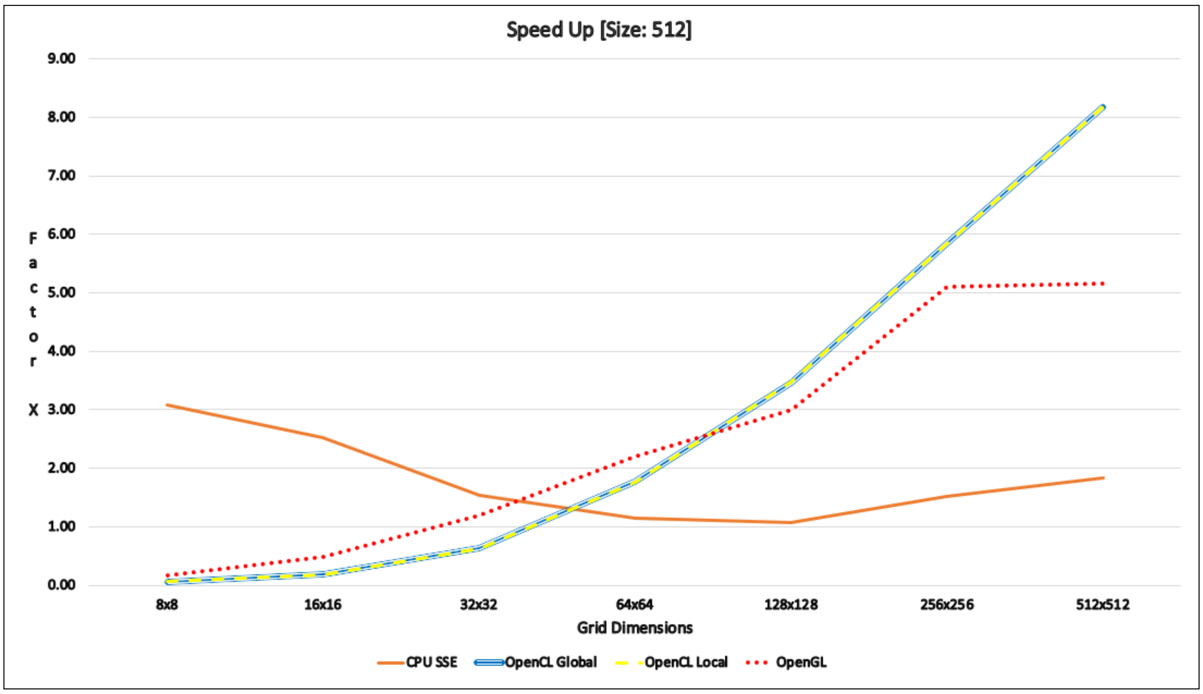


Figure 11: Speedup measured for buffer size 512

to read/write conflicts to the same buffer for different time steps. It may well be that, as in the OpenCL implementation, separating these into their own texture, enabling only the next time step texture to be write-only, would provide better memory access performance. However, this is speculation and we leave this analysis to future work.

It can be seen across the benchmark results that for smaller grid sizes, e.g. $8x8$, $16x16$ and $32x32$, the CPU versions perform well in comparison to the GPU versions. This is likely caused by the overhead involved with data transfer to and from the GPU and, in general, keeping the host and GPU in sync. This outweighs the benefit from the acceleration provided by the compute kernel itself. To confirm this would require further micro-benchmarking. However, from dimensions $64x64$ on wards, the GPU versions perform increasingly better than the CPU versions as the benefits from processing on the GPU exceed the data transfer overhead. By $512x512$, the OpenCL version scales to around 8 times faster than the original serial implementation. The results suggest this would continue to increase further with larger grids.

The results in the OpenCL global and local caching version were not significantly different. Taking a look at Table 3, at the higher dimensions the local caching version is a couple of milliseconds faster. This would not be a noticeable performance increase. The caching method used in the implementation was basic and could be improved with more advanced techniques, which may produce faster computation.

5 CONCLUSION AND FUTURE WORK

We have described implementations for a physical model simulation of a drum membrane in C++, AVX, OpenGL, and OpenGL, providing an analysis and performance of the

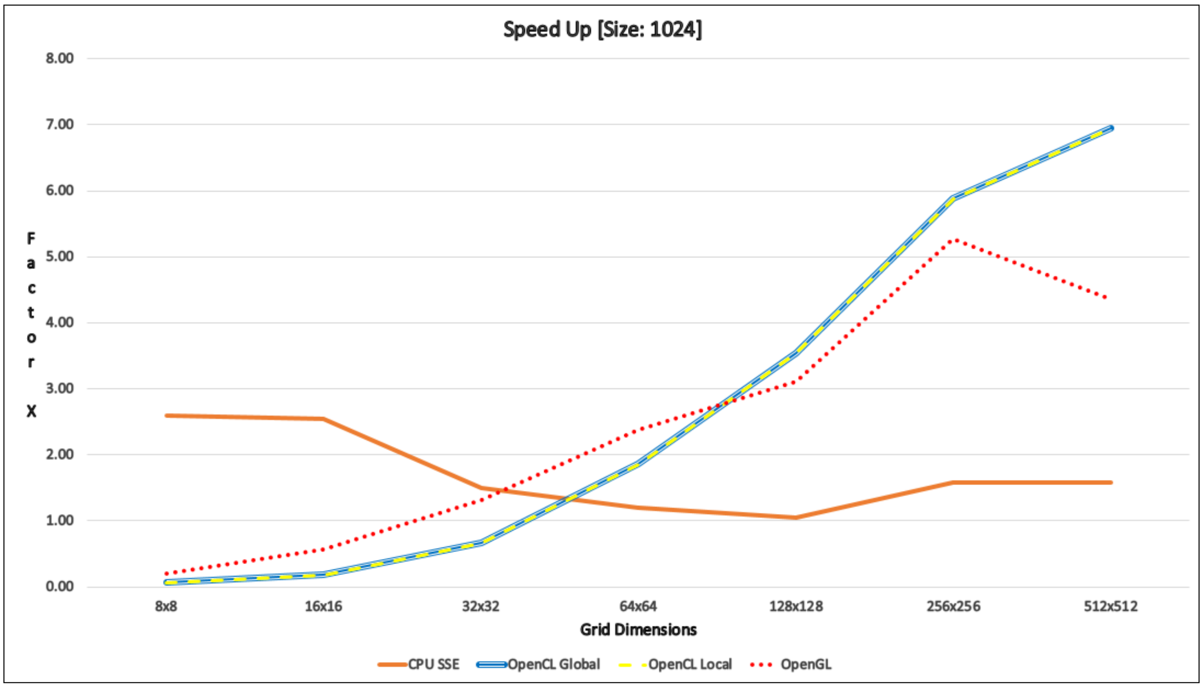


Figure 12: Speedup measured for buffer size 1024

different algorithms. The results found that as mesh size increased, the performance of OpenCL, on a particular Intel (CPU)/AMD (GPU) test system, surpassed the others, at times by a factor of eight over the serial CPU and two compared to OpenGL.

At the time of writing, the results are limited to a single platform and we plan to extend to others, including GPUs from other manufactures and also different CPUs. It would be interesting to look at the performance in the context of mobile devices, e.g. iOS and Android, as these platforms are becoming increasingly popular with musicians.

Of particular interest for future work, is targeting modern graphics APIs, such as Apple's Metal 2 and Khronos' Vulkan [5, 19]. Both of these APIs provide advanced compute capabilities that in some cases surpass that of OpenCL 1.2, e.g. Vulkan 1.1's subgroups extension, while retaining close integration with their graphics components.

Finally, our current CPU implementation is limited to only using 128-bit SIMD, while recent versions of AVX support 512-bit SIMD [11] and scatter/gather memory operations, which are likely to provide significant benefits on laptop class Intel i9 processors, as found in the latest Apple MacBook Pro models, for example. Further, OpenCL can be used to target CPUs and measurements can be taken to show how well it utilizes the CPUs parallel capabilities.

ACKNOWLEDGMENTS

We would like to thanks our colleagues in the Computer Science Research Centre and Creative Technology Lab for useful discussions and feedback on this work. In particular, Nathan Renney and David Creasey provided useful insights into physical modeling for digital synthesis. This work was supported in part by a grant from the HSA Foundation and the UWE 50-50 PhD fund.

REFERENCES

[1] [n. d.]. Apple's AudioUnit. https://developer.apple.com/documentation/audiounit?language=objc. Accessed: 2019-01-24.
[2] [n. d.]. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide. Accessed: 2019-01-23.
[3] Andrew Allen and Nikunj Raghuvanshi. 2015. Aerophones in Flatland: Interactive Wave Simulation of Wind Instruments. *ACM Trans. Graph.* 34, 4, Article 134 (July 2015), 11 pages. https://doi.org/10.1145/2767001
[4] Andrew Allen and Nikunj Raghuvanshi. 2015. Aerophones in flatland: Interactive wave simulation of wind instruments. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 134.
[5] Inc. Apple. [n. d.]. Metal. https://developer.apple.com/metal/. Accessed: 2019-01-25.
[6] Stefan Bilbao and Maarten van Walstijn. 2005. A Finite difference plate Model.. In *ICMC*.
[7] Patrick D. Cannon and Farideh Honary. 2015. A GPU-Accelerated Finite-Difference Time-Domain Scheme for Electromagnetic Wave Interaction With Plasma. *IEEE Transactions on Antennas* (2015).
[8] Cumhur Erkut and Matti Karjalainen. 2002. Virtual strings based on a 1-D FDTD waveguide model: Stability, losses, and traveling waves. In *Audio Engineering Society Conference: 22nd International Conference: Virtual, Synthetic, and Entertainment Audio*. Audio Engineering Society.
[9] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. 2012. *Heterogeneous computing with openCL: revised openCL 1*. Newnes.
[10] L. Hiller and P. Ruiz. 1971. Synthesizing Musical Sounds by Solving the Wave Equation for Vibrating Objects. *Journal of the Audio Engineering Society* (1971).

[11] Intel. [n. d.]. Intel Advanced Vector Extensions 512 (Intel AVX 512). https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/avx-512-overview.html. Accessed: 2019-01-25.

[12] Kevin Karplus and Alex Strong. 1983. Digital Synthesis of Plucked-String and Drum Timbres. *Computer Music Journal* 7, 1 (1983), 43–55.

[13] Kevin Karplus and Alex Strong. 1983. Extensions of the Karplus-Strong Plucked String Algorithm. *Computer Music Journal* 7, 2 (1983), 56–69.

[14] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 54, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014977

[15] Harri Renney. [n. d.]. CPU, AUX, OpenCL, and OpenGL FDTD implementation source code. https://gitlab.uwe.ac.uk/Physical-Modeling/iwocl-fdtd-benchmarking. Accessed: 2019-01-25.

[16] Marc Sosnick and William Hsu. 2010. Efficient finite difference-based sound synthesis using GPUs. In *Proceedings of the Sound and Music Computing Conference*. 42–44.

[17] Maarten Van Walstijn and Konrad Kowalczyk. 2008. On the numerical solution of the 2D wave equation with compact FDTD schemes. *Proc. Digital Audio Effects (DAFx), Espoo, Finland* (2008), 205–212.

[18] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2016. FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL. In *15th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, June 26-29, 2016*. 1–6. https://doi.org/10.1109/ICIS.2016.7550742

[19] Vulkan working group. [n. d.]. Vulkan 1.1. https://www.khronos.org/vulkan/. Accessed: 2019-01-24.

[20] Victor Zappi, Andrew Allen, and Sidney Fels. 2017. Shader-based Physical Modelling for the Design of Massive Digital Musical Instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 145.

HyperModels - A Framework for GPU Accelerated Physical Modelling Sound Synthesis

Harri Renney
Com Sci Research Centre
University of West of England
Bristol, UK
harri.renney@uwe.ac.uk

Silvin Willemsen
Multisensory Experience Lab
Aalborg University Copenhagen
Copenhagen, Denmark
sil@create.aau.dk

Benedict R. Gaster
Com Sci Research Centre
University of West of England
Bristol, UK
benedict.gaster@uwe.ac.uk

Thomas J. Mitchell
Creative Technology Lab
University of West of England
Bristol, UK
tom.mitchell@uwe.ac.uk

Jan 22, 2022

Abstract

Physical modelling sound synthesis methods generate vast and intricate sound spaces that are navigated using meaningful parameters. Numerical based physical modelling synthesis methods provide authentic representations of the physics they model. Unfortunately, the application of these physical models are often limited because of their considerable computational requirements. In previous studies, the CPU has been shown to reliably support two-dimensional physical models in real-time with resolutions up to 64x64. However, the near-ubiquitous parallel processing units known as GPUs have previously been used to process considerably larger resolutions, as high as 512x512 in real-time. GPU programming requires a low-level understanding of the architecture, which often imposes a barrier for entry for inexperienced practitioners. Therefore, this paper proposes HyperModels, a framework for automating the mapping of finite-difference based physical modelling synthesis into an optimised parallel form suitable for the GPU. An overview of HyperModels is given, with a breakdown of all components involved. An implementation of the design is then used to evaluate the objective performance of the framework by comparing the automated solution to manually developed equivalents. For the majority of the extensive performance profiling tests, the auto-generated programs were observed to perform only 6% slower but in the worst-case scenario it was 50% slower. The initial results suggests that, in most circumstances, the automation provided by the framework avoids the low-level expertise required to manually optimise the GPU, with only a small reduction in performance. However, there is still scope to improve the auto-generated optimisations. When comparing the performance of CPU to GPU equivalents, the parallel CPU version supports resolutions of up to 128x128 whilst the GPU continues to support higher resolutions up to 512x512. To conclude the paper, two instruments are developed using HyperModels based on established physical model designs.

Author Keywords

NIME, GPU, High-Performance, Physical Modelling

CCS Concepts

Computing methodologies → Computer graphics → Graphics systems and interfaces → Graphics processors; Applied computing → Arts and humanities → Sound and music computing; Computing methodologies → Modeling and simulation;

Introduction

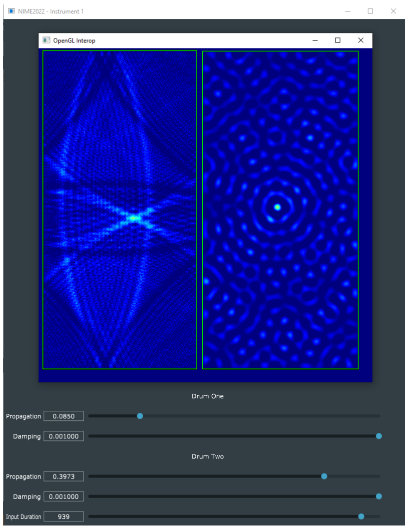
In the 1950s, digital audio synthesis was beginning to gain traction with components such as digital oscillators, filters and stored lookup tables [9], these were used to generate sound, and later, to build synthesis techniques including AM and FM [11]. These simple techniques are often highly computationally efficient [35] in order for them to be used in real-time applications at times when the available computation was very limited by today's standards [7, p. 3]. Furthermore, these methods are considered 'abstract' as they do not directly associate with a physical interpretation. Physical modelling methods contrast with abstract synthesis as they are built on direct interpretation of physical phenomena. Although a broad field of physical modelling methods exists, the direct numerical physical models are the most authentic forms that directly simulate the vibrations through a discretised mathematical representation [2]. Direct numerical physical models simulate an environment by approximating vibration values through N-dimensional space and time. Increasing the resolution of the simulated space has several advantages, including: improved accuracy, more stable simulations and the space to create more sophisticated instruments. However, increasing the resolution proportionally increases the computation required to run simulation. Considering the strict real-time requirements of audio synthesis [23] [22], the usefulness of these methods has been heavily restricted. But with the modern advancements in computer systems, physical modelling synthesis is seeing a possible resurgence [39]. For example, many academics involved in the Next Generation Sound synthesis (NESS) collaboration project believe physical models will play an important part in the future developments of sound synthesis. In 2015 and 2016, the NESS project published dozens of papers and demonstrations related to physical modelling [26], including thorough experimentation and discussions of utilising GPU acceleration for physical modelling sound synthesis [4, 6, 20]. The collective output from the NESS project highlights the benefit of increasing data throughput using the GPU. For example, in [19], GPU acceleration was used to approach offline processing of room acoustics and, for a particular arrangement, was shown to improve performance by X46 over the equivalent serial CPU version. However, they also address the issues of processing various physical modelling techniques in parallel - particularly the incompatibility of specific mathematical methods for parallelisation [3]. For instance, iterative methods like the Newton Raphson [5] for solving implicit schemes inherently involve serial stages that significantly limits the benefits of GPU parallel processing.

Although recent designs and implementations of physical models have been presented within the context of the CPU, for example, [28, 37, 42, 43, 44, 45], these are often restricted to one-dimensional or low resolution two-dimensional models. To support higher resolution , parallel hardware can be used to accelerate the simulations. The GPU is a nearly-ubiquitous massively parallel processing unit that can now be accessibly programmed for general computation because of the modern general-purpose GPU (GPGPU) architecture [8]. GPGPU has been successfully used for processing a range of numerical and scientific computation techniques like molecular dynamics [29]. A well-known and successful example of this is the folding@home project [15], which reported a speedup of 20 - 30X when accelerated on the GPU. The finite difference based numerical methods used for physical modelling synthesis are often described as "embarrassingly parallel" [21], meaning they can be efficiently mapped to the parallel GPU architecture. Previous investigations in [32] demonstrated that for a particular physical model on a modern desktop, the CPU could support two-dimensional models up to 64x64 in real-time. The equivalent approach on the GPU was shown to support resolutions as high as 512x512 in real-time.

In this paper, we present HyperModels, a framework for describing high-resolution physical models that utilise GPU hardware acceleration for real-time synthesis. Instruments are described using high-level descriptions of the physics equations and an instrument's shape, e.g. the strings or membrane, that are automatically translated into optimised low-level code that utilises the real time capabilities of modern GPUs. Our approach enables the instrument designer to focus on the sound design aspect of a new instrument, without necessarily requiring the advanced low-level architecture and programming knowledge often required to access parallel GPU programming. An example covered in Section Instrument 1: Hyper Drumhead is a physically modelled drum instrument. Figure 1a shows the screenshot of the GPU optimised application that is formed using the HyperModels framework when provided with the physics equations and vector graphic description in Figure 1b.

The rest of this paper provides details of the HyperModels framework as a design and implementation, including an analysis of how it performs compared to equivalent hand-written GPU simulations. To demonstrate the system in practice, two existing designs based on physical modelling instruments are implemented using HyperModels. The first is the *Hyper Drumhead*, previously demonstrated by Zappi et al. in hand optimised GPU code [32, 47]. The second is a variant of Willemsen et al. hammered dulcimer [42], that originally operated with a plate model of 17x6 points running on the CPU, this has been ported to the GPU using HyperModels to support resolutions up to 256x256. The remaining sections of the paper are structured as follows:

- Section Numerical Physical Modelling provides an introduction to numerical physical modelling;
- Section GPU Acceleration gives a brief description of the *GPU Architecture*;



Drum One

Propagation 0.0850

Damping 0.001000

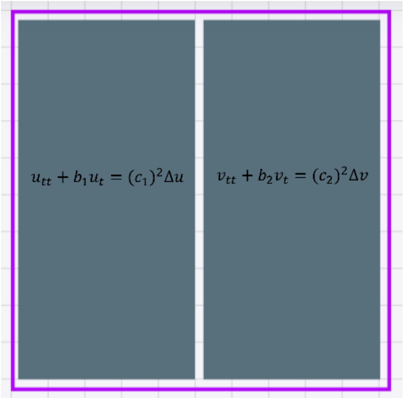
Drum Two

Propagation 0.3973

Damping 0.001000

Input Duration 939

(a) Screenshot of *Hyper Drumhead* application.


$u_{tt} + b_1 u_t = (c_1)^2 \Delta u$
$v_{tt} + b_2 v_t = (c_2)^2 \Delta v$

(b) Vector graphics and physical model description for u and v in system a.

Figure 1: Instrument 1 *Hyper Drumhead*

- Section HyperModels Framework contains the main contribution of this paper;
- Section Performance Evaluation evaluates the performance of the proposed system;
- Section Case Studies presents two instruments using HyperModels;
- Section Conclusion closes with final remarks and pointers to future work.

Numerical Physical Modelling

One of the most foundational numerical methods used for approximating derivatives is the finite-difference method. Although it has some drawbacks compared to other methods, such as the difficulties handling curved boundaries [30], it is an effective method for digital audio synthesis and is inherently suited to parallelisation. These numerical methods date back further than the earliest examples of digital audio synthesis, with its origins in engineering and general physics [14]. The adoption and research of numerical methods for sound synthesis have only recently become practical outside of a theoretical context because of the abundance of computational power that is available in modern personal computing devices.

Simple Harmonic Oscillator

The simple harmonic oscillator is one of the core building blocks of digital audio synthesis. A simple harmonic oscillator can also be formed as a physical model using an ordinary differential equation (ODE). An ODE is a differential equation containing one or more functions of one independent variable and the derivatives of those functions [10]. The ODE for the simple harmonic oscillator is:

$$u_{tt} = -\omega_0^2 u, \tag{1}$$

where state variable $u = u(t)$ describes the displacement of the mass from its equilibrium (in m), $u_{tt} = \frac{\partial^2 u}{\partial t^2}$ is its acceleration and ω_0 is the angular frequency of oscillation (in rad/s). To solve this using finite differences, the state of the mass as well as the derivatives must be approximated, or discretised. Using $t = nT$, with time index $n = 0, 1, \dots$ and time step T, the simple harmonic oscillator in Equation (1) can be discretised to:

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{T^2} = -\omega_0^2 u^n. \tag{2}$$

Here, u^{n+1} refers to the value of the oscillator at the next timestep $n+1$. Using values of the current timestep u^n and previous timestep u^{n-1} along with the size of the timestep T, the value of the oscillator at the next timestep can be computed by rearranging Equation (2) to the following update equation:

$$u^{n+1} = \left(2 - \omega_0^2 T^2\right) u^n - u^{n-1}. \tag{3}$$

By stepping through time and recursively calculating Equation (3), the simple harmonic oscillator is simulated. Figure 2 shows that at each timestep, the explicit scheme depends on u^n and u^{n-1} to calculate u^{n+1}. As u^{n+2} depends on the result of u^{n+1}, the scheme can only be processed serially by a single processor, as there is only one stream of values to simulate that depend on each other.

1D Wave Equation

Partial differential equations (PDE) are an extension to the concept of ODEs, like the simple harmonic oscillator discussed in Equation (1). Where ODEs involved a single independent variable, PDEs involve two or more independent variables [17]. The 1-dimensional wave equation is an example of a partial differential equation:

$$u_{tt} = c^2 u_{xx}, \tag{4}$$

where state variable $u = u(x, t)$ is now dependent on time t as well as spatial coordinate x. Assuming a system of length L (in m), the spatial domain becomes $x \in [0, L]$. Furthermore, c is the wave speed (in m/s).

To approximate Equation (4), one can subdivide the spatial domain into N intervals with equal length X. The spatial coordinate can then be approximated

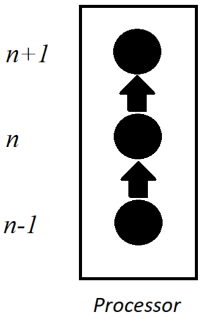


Figure 2: Serial processing of ODE finite-difference schemes.

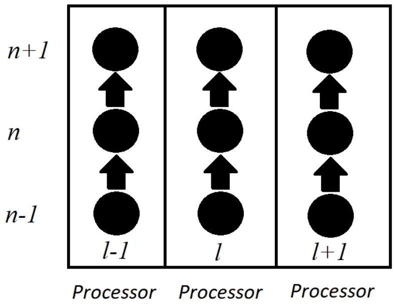


Figure 3: Parallel processing of PDE finite-difference schemes.

according to $x = lX$ with spatial index $l = \{0, \dots, N\}$. Using the spatial index as well as the temporal index n described above, one can define grid function u_l^n which is a discrete approximation to $u(x,t)$.

Approximating u_{tt} and u_{xx} using finite-differences, the following recursively solvable explicit scheme can be formed:

$$u_l^{n+1} = 2u_l^n + \lambda^2(u_{l+1}^n - 2u_l^n + u_{l-1}^n) - u_l^{n-1}, \tag{5}$$

where $\lambda = cT/X$ is the Courant number and needs to abide the condition for the scheme to be stable [13]: $\lambda \leq 1$. To solve this scheme, the simulation now needs to calculate u_l^{n+1} not just for one position, but for $N+1$ positions in the model. All of these positions will need to calculate Equation (5) and store the result in u_l^{n+1} for all l positions as shown in Figure 3. These calculations do not interfere or depend on each other, making each spatial point for the timestep solvable in parallel. Parallel streams of processing like those found on the GPU can each handle a single grid point and provided N is big enough to fully utilise all parallel processors, speed up the calculation of the system for each timestep.

2D Wave Equation

One can extend the 1-dimensional wave equation in Equation (4) to 2 dimensions. The 2-dimensional wave equation is defined as

$$v_{tt} = c^2(v_{xx} + v_{yy}), \tag{6}$$

with wave speed c (in m/s) and where state variable $v = v(x, y, t)$ is defined over two spatial dimensions x and y.[1] For a rectangular system of side lengths L_x (in m) and L_y (in m), $(x, y) \in \mathcal{D}$ where $\mathcal{D} \in [0, L_x] \times [0, L_y]$.

Similar to before, Equation (6) this can be discretised using finite differences. Discretising time as usual ($t = nT$), space can be subdivided into N_x intervals in the x-direction and N_y intervals in the y-direction, yielding $x = lX$ and $y = mX$,[2] with $l \in \{0, \ldots, N_x\}$ and $m \in \{0, \ldots, N_y\}$. Using these definitions, the state variable $v(x, y, t)$ can then be approximated using grid function $v_{l,m}^n$. Discretising Equation (6) and solving for $v_{l,m}^{n+1}$, yields the following update equation:

$$v_{l,m}^{n+1} = 2v_{l,m}^n - v_{l,m}^{n-1} + \lambda^2(v_{l+1,m}^n + v_{l-1,m}^n + v_{l,m+1}^n + v_{l,m-1}^n - 4v_{l,m}^n), \tag{7}$$

where the following stability condition must be satisfied [7]: $\lambda \leq \frac{1}{\sqrt{2}}$.

GPU Acceleration

Modern computer systems are built as heterogeneous platforms [36], meaning systems utilise different processing devices simultaneously for maximum efficiency. The CPU and its typical 4-14 fast processors [38] are usually reserved for most general processing, while the hundreds of parallel processors on the GPU can be used to offload and accelerate suitable tasks such as graphics. Although the CPU and GPU share many similarities, there are key differences that need to be understood to efficiently target both devices. An abstract view of the modern GPGPU architecture is shown in Figure 4 [40]. Here, the CPU interfaces with the GPU unit across a *bridge* and a *host interface* loads instructions and programs onto the GPU [27]. The device then loads the programs across the compute devices; these manage the execution of instructions from the program across their numerous Processing Elements (PE). According to the program instructions, all the PEs then execute the same instructions simultaneously on different sections of data in memory by using the ID of the stream of execution to index into memory. This means that each compute unit supports the single-instruction multiple-data (SIMD) paradigm. Note that the compute device has multiple compute units that can each have different sets of program instructions loaded onto them. This extends the SIMD paradigm to single-instruction Multiple-thread (SIMT) [12] and this arrangement enables the massively parallel processing environment of

[1] Notice that we use state variable v here (instead of u) for consistency with later sections.

[2] We assume the same grid spacing in the x and y-direction.

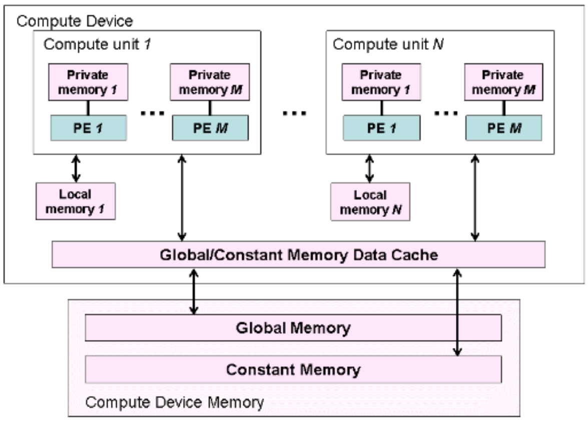


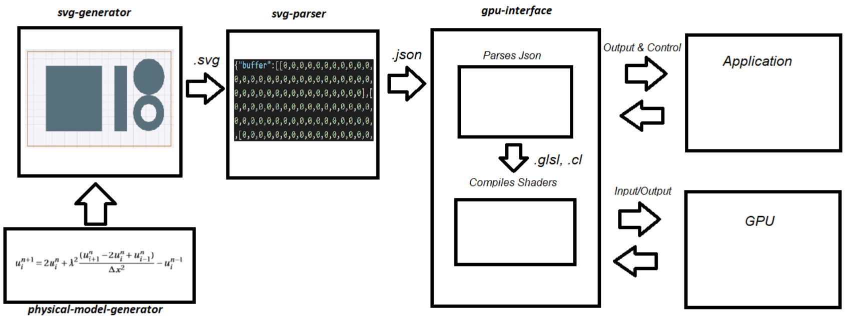
Figure 4: The modern GPU architecture [40].

the GPU. In this paper, the term *core* will be used to refer to streams of execution on the GPU cores and not the physical cores. This abstraction helps describe how data is processed fully in parallel, but in reality, all of this processing might not execute in parallel at once if the number of items to process exceeds the number of physical cores.

The finite-difference methods used in this paper require solving simple linear systems of equations that are entirely independent from one another. This means the entire state of the grid can be contained in GPU global memory meaning each point on the grid can be accessed and processed concurrently, without any explicit synchronisation or communication between PEs. Further, scaling up the number of points does not impact on this advantage. For example, to calculate a point in space and time requires accessing neighbouring values, but the result of the calculation only requires writing to a single, mutually exclusive location. Therefore, when mapping numerical physical models to the GPU, a core can be allocated to solve each equation at each position in space and write the resulting value to memory without affecting other neighbouring cores. This mapping is the case for models based on linear systems, more sophisticated systems, such as non-linear variants are usually not recursively solvable explicit schemes.

HyperModels Framework

The HyperModels framework is outlined in Figure 5. There are four distinct software components: *physical-model-representation*, *svg-generator*, *svg-parser* and then a *gpu-interface*. The *physical-model-representation* provides the mapping between the finite-difference schemes to parallel processing environment. The vector graphic representation of the physical model geometry is described



```
{"buffer":[[0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

Figure 5: Relationship diagram of the HyperModels GPU physical modelling tools.

using an annotated scalable vector graphics (SVG) [16] format in *svg-generator*. The SVG is then parsed in *svg-parser* to produce a two-dimensional grid representation of the models that is suitable for the finite-difference schemes. The physical model is captured inside a JSON object that the *gpu-interface* can load into the GPU and integrated into an application to create an instrument.

Model to GPU Mapping

Mapping finite-difference equations into a parallel processing environment requires defining an appropriate representation and program structure. Here, we define the GPU as system a with a grid of $C_x \times C_y$ cores. A single core is denoted by a_{c_x,c_y} where $c_x \in \{1,\ldots,C_x\}$ and $c_y \in \{1,\ldots,C_y\}$ are the core indices in the horizontal and vertical direction of the GPU grid respectively. The update equation of one grid point can then be assigned to a single core, such that these can be executed in parallel. Considering the 2D system presented in Equation (7), mapping a grid point of a scheme with spatial index (l,m) to a core with index (c_x,c_y) is denoted by $a_{c_x,c_y} \Leftarrow v_{l,m}$.

The cores of the GPU a must then be processed by recursively traversing the two-dimensional system a and calculating the values of a^{n+1} at each position depending on what models are assigned at each position. This process is repeated recursively and once all of a^{n+1} is calculated, time index n can be incremented. A serial representation of this process would be formed using two nested for loops:

Algorithm 1 Serial Representation

```
n = 0
while isSimulation do
    for x = 1 to gridX do
        for y = 1 to gridY do
            if id[x][y] == v then
                a[n+1][x][y]  = 2 * a[n][x][y] - a[n-1][x][y]
                              + $\lambda^2$*(a[n][x+1][y] + a[n][x-1][y] + a[n][x][y+1]
                              + a[n][x][y-1] - 4 * a[n][x][y])
            end if
        end for
    end for
    n = n + 1
end while
```

Here, every position in the grid is visited by iterating over all possible values for x and y. These are first used to check if the position in the two-dimensional grid is inside the model v. If it is, then Equation (7) is used to calculate the value at $a^{n+1}_{c_x,c_y}$. In the equation, components such as $a^{n}_{c_x+1,c_y}$ require accessing neighbouring values of the currently considered position at c_x and c_y by adding 1 to x, leading to a[n][x+1][cy]. This is done for all neighbouring values. This serial approach can be rewritten to the following, so that it is suitable for parallel processing:

Algorithm 2 Parallel Representation

```
n = 0
while isSimulation do
    cx = getGridX()
    cy = getGridY()
    if id[cx][cy] == v then
        a[n+1][cx][cy]= 2 * a[n][cx][cy] - a[n-1][cx][cy]
                      + $\lambda^2$*(a[n][cx+1][cy] + a[n][cx-1][cy] + a[n][cx][cy+1]
                      + a[n][cx][cy-1] - 4 * a[n][cx][cy])
    end if
    n = n + 1
end while
```

Here the nested for loops are implicit, with the loop's indices represented through the identifiers getGridX() and getGridY(), respectively, where $C_x \times C_y$ process streams are dispatched to the processor. Thus, instead of iterating over two nested for loops, the ID of each process stream is used to check which model's equation to use, such as v for accessing a to calculate the next timestep a^{n+1}.

The state of the system a is contained in global GPU memory as it is only directly accessed once by each PE.

Multiple models can be added to the simulation, where the equation at each position in the system a depends on the c_x and c_y coordinates. For example, one can add the 1D wave equation described in Equation (5) as a second model to the GPU. This one-dimensional equation can be mapped into the system a according to $a_{c_x,c_y} \Leftarrow u_l$. The code then includes an additional conditional statement checking if the coordinate (c_x, c_y) identifies u:

Algorithm 3 Parallel Representation

```
n = 0
while isSimulation do
    cx = getGridX()
    cy = getGridY()
    if id[cx][cy] == v then
        a[n+1][cx][cy]= 2 * a[n][cx][cy] - a[n-1][cx][cy]
                    + $\lambda^2$*(a[n][cx+1][cy] + a[n][cx-1][cy] + a[n][cx][cy+1]
                    + a[n][cx][cy-1] - 4 * a[n][cx][cy])
    end if
    if id[cx][cy] == u then
        a[n+1][cx][cy]= 2 * a[n][cx][cy] - a[n-1][cx][cy]
                    +$\lambda^2$* (a[n][cx+1][cy] - 2 * a[n][cx][cy]+ a[n][cx-1][cy] )
    end if
    n = n + 1
end while
```

This mapping has been implemented using a domain specific language (DSL) [34] where finite-difference schemes can be defined. The DSL compiler generates a GPU program that captures the parallel representation. The details of this implementation will be covered in future work.

Vector Based Representation

A method for mapping models such as u into the system a is required for describing the shapes of the models. The SVG format is a compact and extendable vector graphics protocol that supports descriptions of shapes and their properties as XML tags and attribute values. To support the physical models, the standard SVG format is extended to capture important information associated with each shape. An example of an SVG containing a rectangle that includes physical modelling attributes is as follows:

```
<svg viewbox='0 0 12 8' width='12' height='8'
    interface_device='custom' connections=''>
    <rect id='1' interface_osc_address='' interface_type='pad'
        interface_osc_args=''
        width='2' height='10' x='4' y='2'
        style='fill:rgb(88,111,124);'
        physics_program='...'/>
</svg>
```

Inside the SVG *svg* tags, the *viewbox*, *width* and *height* describe the resolution of the entire physical modelled environment. Defining these as *viewbox='0 0 12 8' width='12' height='8'* creates a 12x8 simulation environment. This dictates the possible space to work in and how detailed shapes will be in the simulation. The *connections* field contains a list of coordinates that are connected together between models. The *physics_program* contains the generated GPU program from the *physical-model-representation* stage. Each shape contains two additional attributes, *id* and *physics_program*. The *id* is the unique identifier for each shape, this is used to fill in the *id* of the shape when generating the 2-dimensional grid in the *svg-parser*.

Vector Parser

Simulating the physical models using finite-difference based methods requires a Cartisian grid of points to apply calculations to. Therefore, the SVG vector image format must be used to generate a 2-dimensional grid. The design presented here is based on the SVG parser in [18], with modifications making it appropriate for the physical modelling framework. Figure 6 covers the stages used by the SVG parser. First, the SVG description is tessellated, this produces a set of triangles that when considered holistically, form the original shape. These triangles prepare the shape for rasterisation, where all points inside the respective shape's triangles are populated with the ID of the shape. Here, the *rect* with a height of 10 and width of 2 is tessellated and points inside the triangles are filled with the shape ID inside the 12x8 simulation grid. The output of the parser is then used to populate a JSON object with the following data:

```
{
    "models" : [
    {
        "id": 1,
        "rgb": "rgb(217,137,188)",
        "args": [
        0.39
        ]
    },
    ⊡ ⊡ ⊡
    ],
    "environment": [
        [
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        ]
    ] ,
```

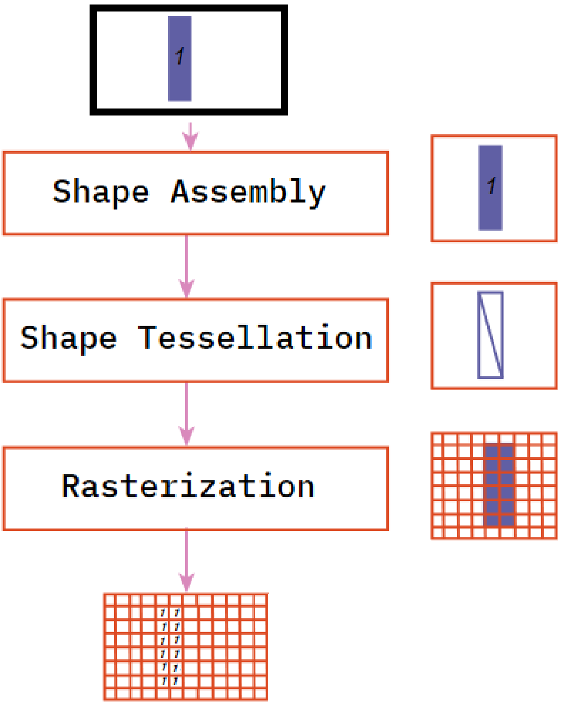


Figure 6: Interface tessellation and rasterisation of single rectangle. [18]

```
    "physics_program": "...",
    "connections": ...
    "interface" : "custom"
}
```

Here the Cartisian grid that contains each model’s ID is contained in *environment*, while a list of *models* contains details for each model, particularly the *id* that indicates what equation will be executed for each point inside the GPU *physics_program*. The *connections* field contains a list of coordinates that are linked together between models.

GPU Interface

The whole physical model environment captured inside the JSON object can now be used by a GPU interfacing API to load the GPU program, prepare the memory modelling the environment and then execute it to generate audio samples. The following C++ function declarations are used in this implementation:

```
void createModel(const std::string aPath);

void step();
void fillBuffer(float* input, float* output, uint32_t numSteps);
void updateCoefficient(std::string aCoeff, uint32_t aIndex, float aValue);

void setInputPosition(int aInputs[]);
void setOutputPosition(uint32_t aOutputs);
```

createModel() takes a file path to the JSON file containing the physical model description. The GPU program for the target GPU interfacing API can then be loaded onto the GPU. *step()* is a private function that is used for advancing the physical model one timestep. This is used by *fillBuffer* to recursively advance the physical model and at each time step extract samples from the output position to fill the *output* buffer. updateCoefficient() is used to change the value of coefficients. The input and output positions for excitation and sample generation respectively can be moved using *setInputPosition()* & *setoutputPosition()*.

The performance of this framework is evaluated by profiling an implementation of the HyperModels framework. This will provide details of the frameworks strengths and limitations that will lead into the development of two example instruments.

Performance Evaluation

The convenience provided by automating development is often contrast with a trade-off with performance. This section provides a brief evaluation of the performance by comparing the auto-generated GPU programs from HyperModels with manually developed equivalents. To create a controlled testing environment, only the GPU programs will be modified. This means the interfacing methods and physical model representation will be the same between versions. Further, measurements taken from equivalent parallel CPU versions have been included for comparison.

Comparative Tests

The benchmarking suite operates by following the real-time profiling technique used in [33]. This approach runs a collection of tests through the following template:

```
void realtimeTest() {
    if (isWarmup) {
        executeTest();
    }
    while (numSamplesComputed < sampleRate) {
        startTimer();
        executeTest();
        endTimer();

        numSamplesComputed += bufferLength;
        checkTestResults(testResults);
    }
    elapsedTimer();
```

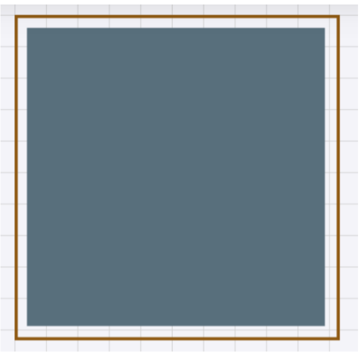


Figure 7: SVG representation of *Simple Single Model* model geometry.

```
    cleanup(hostVariables, deviceVariables);
}
```

Here, the *executeTest* function represents the physical modelling synthesis function that generates *bufferLength* samples. This function is repeatedly called until a seconds worth of samples has been computed at a specified sample rate. This template can then be repeated and the average time to process at the sample rate can be measured. The *isWarmup* flag is used to execute the test once without measuring performance as the first time a GPU program is executed it can be considerably slower than all subsequent executions as the GPU prepares and optimises on the first execution.

In this evaluation, the test will be considered when processing at the minimum acceptable sample rate of 44100Hz [23]. The tests will be repeated for a series of escalating grid dimension sizes, starting at 64x64 increasing by powers of 2 up to 1024x1024. Four tests have been designed to test the basic functionality of the physical models and reveal contextual differences in performance between the automatic and manually written GPU physical model programs. For conciseness in this paper, only the salient results from two of the tests [3] will be considered in this analysis and are as follows:

- *Simple Single Model* - A single 2-dimensional wave equation square physical model. Utilisation=88% (Figure 7)
- *Complex Multiple Models* - Two 2-dimensional square physical models connected by a single string. Utilisation=51% (Figure 8)

Manually writing the GPU program provides opportunities for a competent developer to exploit contextual elements using foresight that automated tools either have difficulty identifying or can not safely implement. Some of the optimisations exclusive to the manual version are full constant folding [25,

[3] The source-code for the benchmarking suite used along with all recorded results are available online: https://github.com/Harri-Renney/HyperModels-benchmarking-suite

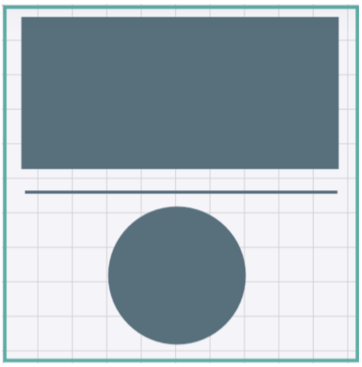


Figure 8: SVG representation of *Complex Multiple Models* model geometry.

p. 329] in equations and grouping models with identical equations into one control dependency. Each of these tests have a utilisation measurement shown as a percentage. This is an important feature of each tests as it denotes how intensive the simulation is to process.

Results

The test results for *simple single model* are shown in Figure 9. Considering the GPU auto-generated and manual versions, grid resolutions between $C_x = C_y = 64$ and $C_x = C_y = 256$ appear to show comparatively marginal difference. However, from $C_x = C_y > 256$, the disparity begins to emerge, where the manually written program begins to perform increasingly better. With this projection, the manually written version can operate at higher resolutions up to around $C_x = C_y = 700$, while the auto-generated version reaches $C_x = C_y = 512$. The limited scope of the *simple single model* test means the only notable optimisation added to the manual version is the constant folding of all redundant calculations. This is where any constant values involved in the equations defined for the model are calculated once on the CPU and uploaded as one constant coefficient to the GPU, instead of redundantly calculating it on the GPU. This optimisation appears effective for this test as it is applied across 88% of the simulated space. Therefore, as expected, optimisations that reduce computation in the model's scheme are effective.

When considering the performance of the GPU against the CPU, the GPU can support far higher resolution models than the CPU in real-time. For this single model test, the manually written CPU version can almost support $C_x = C_y = 128$ which would involve 16384 grid points. The auto-generated GPU version can support $C_x = C_y = 512$ that involves a considerably higher 262144 grid points, 16X more than the CPU can support.

Figure 10 displays the results recorded for the *Complex Multiple Models* test. The results follow a similar trend as the *Simple Single Model* test, however,

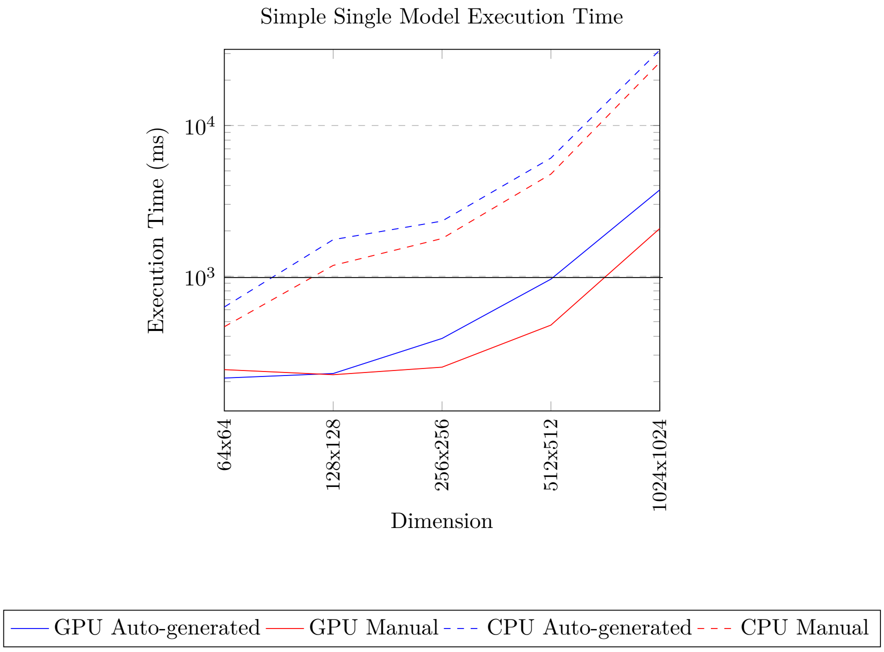


Figure 9: Execution time for a second's worth of sample at 44.1KHz for the *simple single model* test.

the differences between the auto-generated and manual versions are negligible. Both the GPU versions support resolutions up to around $C_x = C_y < 700$. It appears for the test that involves advanced models and connections, the manual optimisations are less effective. This might be partly because the constant folding optimisation is less effective for the smaller grid utilisation of 51%. This test involves two connection points between the three models yet the performance does not appear to be negatively effected by this. This suggests the technique used for the connections in the design is effective, at least for few connection points. However, it can not be assumed to extrapolate as more connections are added, comprehensive experimentation is needed to establish how this scales for hundreds of connections.

The results presented here support the effectiveness of the HyperModels auto-generated programs. For the majority of the tests, the performance difference was a negligible 6% in favour of the GPU manual version over the auto-generated equivalent. Although, under certain contexts the manual version was significantly faster. For example, the manual version of *simple single model* test at $C_x = C_y = 700$ was 50% faster than the auto-generated program.

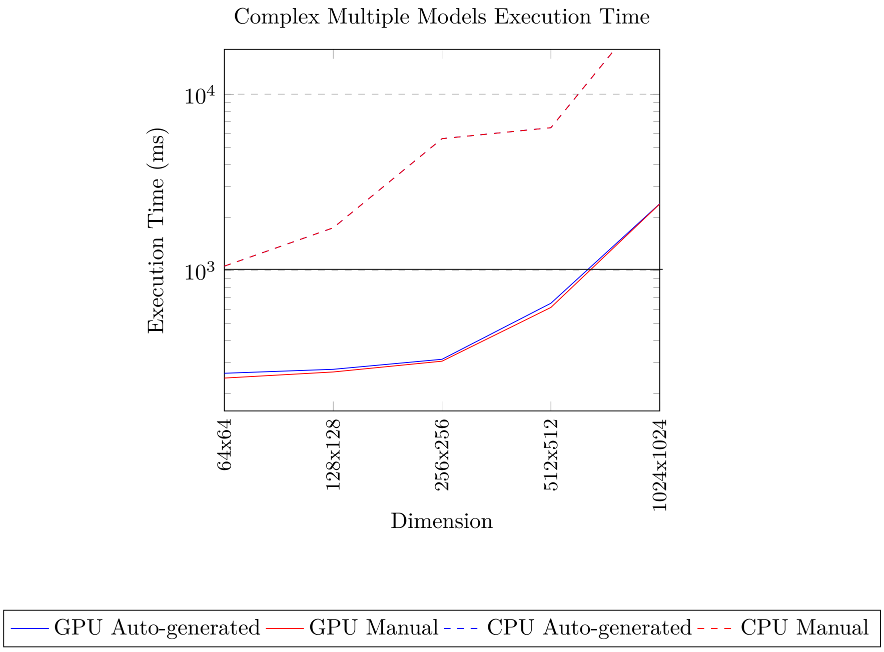


Figure 10: Execution time for a second's worth of sample at 44.1KHz for the *complex multiple model test.*

Case Studies

Two instruments will be presented here to demonstrate the expressiveness of the HyperModels framework. Both instruments will operate within the system a defined in HyperModels that will have a resolution of (C_x, C_y) where models can be defined to operate at specific positions determined by the vector based description of the model shapes.

Instrument 1: Hyper Drumhead

Instrument 1 implements the *Hyper Drumhead* physical model from [47] [46]. In their implementation, the *Hyper Drumhead* is a two-dimensional drumhead simulation that has been accelerated on the GPU using the graphics pipeline. The *Hyper Drumhead* was reported to support resolutions up to 320x320 for most of the systems tested on. In their implementation, Zappi et al. mapped the audio domain of the physical model directly into the graphical domain using the OpenGL graphics rendering API. By conforming to the graphical domain, this design requires an additional field of knowledge and imposes some limitations. For Instrument 1, the *Hyper Drumhead* will be ported to the HyperModels framework with an extended resolution of $C_x = C_y = 512$. The source code and recordings of this instrument are publicly available online [4].

The physical model PDE used in the *Hyper Drumhead* is based on the two-dimensional wave equations with a frequency independent damping component [31]. Equation (7) can be extended to include damping, and the recursively solvable explicit scheme used for the q^{th} model v_q where $q = 1, 2$ will be defined as:

$$\begin{aligned}(1+\sigma_q)v_{q,l,m}^{n+1} &= 2v_{q,l,m}^{n} - (1-\sigma_q)v_{q,l,m}^{n-1} \\ &\quad + (\lambda_q)^2(v_{q,l+1,m}^{n} + v_{q,l-1,m}^{n} + v_{q,l,m+1}^{n} + v_{q,l,m-1}^{n} - 4v_{q,l,m}^{n})\end{aligned} \tag{8}$$

where the Courant number λ_q is defined as in Equation (7), and σ_q is the frequency independent damping coefficient (in s^{-1}). To maintain stability inside the model, the conditions $\lambda \leq \frac{1}{\sqrt{2}}$ and $0 < \sigma < 1$ must be met. The mapping of v_q into the grid of cores in a is illustrated as an SVG in Figure 1b. The physical model described so far is then contained in the *instrument* component in the application visualised in Figure 11. Here, the CPU application program written in the JUCE[5] audio framework interfaces with the GPU instrument program requesting 44100 samples per second. However, the input/output samples between CPU and GPU uses a buffering technique. Therefore, data transfers occur at a rate of $\left\lceil \frac{f_s}{b_s} \right\rceil$. So for a buffer length $b_s = 256$ at $f_s = 44100$, there are $\left\lceil \frac{44100}{256} \right\rceil = 173$Hz data transfers per second. The state of the system a stays in the GPU global memory and is not transferred back to the CPU. Instead, for the on-screen visualisation of the model an OpenGL graphics program is called at a rate of 15Hz. This efficiently maps the state of the instrument into coloured pixels that are then sent directly to a display device. Interactions with the instrument are controlled using two Sensel Morphs [6]. The Sensel morph is a high-resolution pressure sensor that detects the position and amount of pressure of contacts. The two Sensel morph's have been connected to the application, one mapping to model v_1 and the other to v_2 by adding excitation to the system a_{c_x,c_y} at a position c_x and c_y that is detected by the Sensels. The sensel's are polled for contacts at a rate of 150Hz as this provides a maximum detection latency of 6.6ms [42]. A screenshot of Instrument 1's application GUI is shown in Figure 1a.

Instrument 2: String-Plate Connections

Instrument 2 aims to demonstrate that complex, interconnected models can be represented by the GPU accelerated framework. In [42], Willemsen et al. design instruments using strings and plate models that are connected together to form instruments such as the sitar, hammered dulcimer and Hurdy Gurdy. In their implementation, the instruments were executed on the CPU and operated for small resolutions, with strings involving a maximum of 50 points and plates

[4] https://github.com/Harri-Renney/-NIME2022---InstrumentOne

[5] https://juce.com

[6] https://github.com/sensel/sensel-api

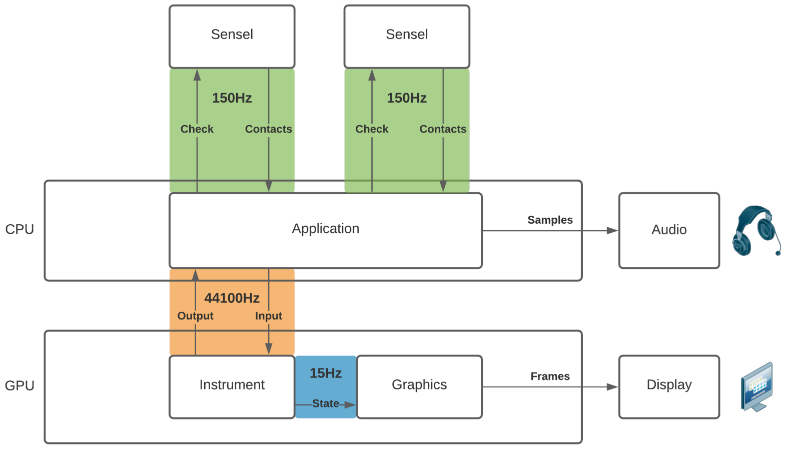


Figure 11: Application overview for Instrument 1: *Hyper Drumhead*.

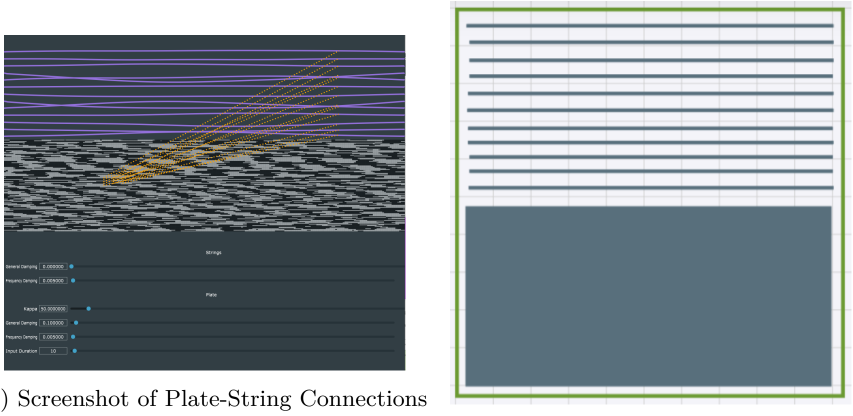
of 20x10. The GPU accelerated implementation will support higher resolution models with multiple strings of 280 points and a plate of 236x121 inside an environment a with $C_x = C_y = 256$. A plate model v and multiple strings u_q will be defined where the subscript q is used to identify each string between $q = 1, \ldots, 13$.

The string models are defined as using the stiff string equation [41, p. 75] along with frequency independent and frequency dependant components [1]. Using finite-differences, the recursively solvable explicit form $u^n_{q,l}$ is formed:

$$\begin{aligned}(1+\sigma_{q,0}T)u^{n+1}_{q,l} = &\left(2 - 2(\lambda_q)^2 - 6(\mu_q)^2\frac{4\sigma_{q,1}T}{X^2}\right)u^n_{q,l} \\ &\left((\lambda_q)^2 + 4(\mu_q)^2 + \frac{2\sigma_{q,1}T}{X^2}\right)(u^n_{q,l+1} + u^n_{q,l-1}) \\ &- (\mu_q)^2(u^n_{q,l+2} + u^n_{q,l-2}) \\ &+ \left(-1 + \sigma_{q,0}T + \frac{4\sigma_{q,1}T}{X^2}\right)u^{n-1}_{q,l} \\ &- \frac{2\sigma_{q,1}T}{X^2}\left(u^{n-1}_{q,l+1} + u^{n-1}_{q,l-1}\right)\end{aligned} \tag{9}$$

with $\mu_q = \frac{\kappa_q T}{X^2}$, stiffness coefficient κ_q, frequency independent damping $\sigma_{q,0}$ and frequency dependant damping $\sigma_{q,1}$.

The linear plate equation [24] uses a similar description of physics as the stiff string including frequency independent and dependant components, but is extended to operate across two-dimensions. By discretising the equation using finite-



Strings

General Damping 0.000000

Frequency Damping 0.005000

Plate

Kappa 50.0000000

General Damping 0.100000

Frequency Damping 0.005000

Input Duration 10

(a) Screenshot of Plate-String Connections application.

(b) Vector graphics and physical model description for u_q and v in system a.

Figure 12: Instrument 2 Plate-String Connections

differences, the following recursively solvable explicit scheme is formed.

$$
\begin{aligned}
(1+\sigma_0 T)v_{l,m}^{n+1} &= (2-20\mu^2-4S)v_{l,m}^{n} \\
&+ (8\mu^2+S)(v_{l+1,m}^{n}+v_{l-1,m}^{n}+v_{l,m+1}^{n}+v_{l,m-1}^{n}) \\
&- 2\mu^2(v_{l+1,m+1}^{n}+v_{l+1,m-1}^{n}+v_{l-1,m+1}^{n}+v_{l-1,m-1}^{n}) \\
&- \mu^2(v_{l+2,m}^{n}+v_{l-2,m}^{n}+v_{l,m+2}^{n}+v_{l,m-2}^{n}) \\
&+ (\sigma_0 T-1+4S)v_{l,m}^{n-1} \\
&- S(v_{l+1,m}^{n-1}+v_{l-1,m}^{n-1}+v_{l,m+1}^{n-1}+v_{l,m-1}^{n-1})
\end{aligned}
\tag{10}
$$

Where parameters are the same as in Equation (9) but with the additional coefficient $S = \frac{2\sigma_1 T}{X^2}$.

The geometry for instrument 2 is displayed in the SVG in Figure 12b. Again, a JUCE application running on the CPU interfaces with the instrument on the GPU at a samplerate of 44100Hz as shown in the instrument overview in Figure 13. The Sensel Morphs are used as input, one being mapped to pluck across the strings u^q and the other to strike the plate v. The key difference in this arrangement is that the visualisation of the instrument is processed on the CPU using the JUCE framework. Therefore, the state of the grid must be transferred from the GPU to the CPU to update the JUCE graphical components to then load onto the GPU at a frame rate of 15Hz. A screenshot of the Instrument 2 application is shown in Figure 12a and the recordings and source code are available online [7].

[7] https://github.com/Harri-Renney/NIME2022---InstrumentTwo

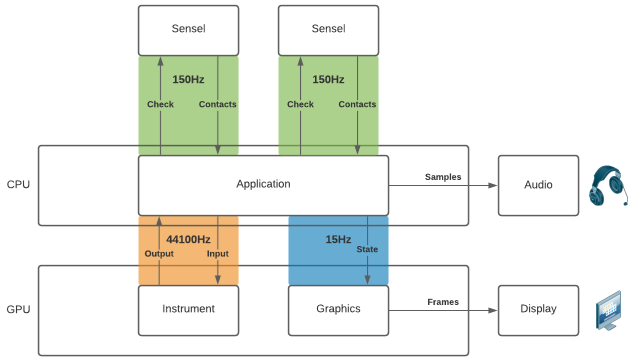


Figure 13: Application overview for Instrument 2: String-Plate Connections application.

Conclusion

This paper has presented an overview of the HyperModels framework for assisting the development of GPU accelerated physical model synthesisers. The performance of HyperModels has been evaluated and shown to outperform CPU versions for resolutions above 64x64, particularly at 512x512 where the auto-generated GPU code was approximately 4X faster than the manually written parallel CPU version. However, the manually written GPU code was consistently faster than the auto-generated equivalent, being around 6% slower for most tests and a maximum of 50% for a particular case. Future work involves optimising the HyperModels implementation further to match the manual program performance and to present the comprehensive details of the entire framework.

Acknowledgments

This work was supported by a grant from the HSA Foundation and the UWE 50-50 PhD fund and has been funded in part by the European Art-Science-Technology Network for Digital Creativity (EASTN-DC), project number 883023.

Ethics Statement

Being a technical paper that presents software designs, the research and contents of this paper involves no ethical considerations and/or implications.

References

[1] Julien Bensa, Stefan Bilbao, Richard Kronland-Martinet, and Julius O Smith III. 2003. The simulation of piano string vibration: From physical models to finite difference schemes and digital waveguides. *The Journal of the Acoustical Society of America* 114, 2 (2003), 1095–1107.

[2] Richard E Berg and David G Stork. 1990. *The physics of sound*. Pearson Education India.

[3] Stefan Bilbao, Brian Hamilton, Alberto Torin, Craig Webb, Paul Graham, Alan Gray, Kostas Kavoussanakis, and James Perry. 2013. Large scale physical modeling sound synthesis. In *Proceedings of the Stockholm music acoustic conference (SMAC2013), Stockholm*. 593–600.

[4] Stefan Bilbao, James Perry, Paul Graham, Alan Gray, Kostas Kavoussanakis, Gordon Delap, Tom Mudd, Gadi Sassoon, Trevor Wishart, and Samson Young. 2019. Large-scale physical modeling synthesis, parallel computing, and musical experimentation: the NESS project in practice. *Computer Music Journal* 43, 2-3 (2019), 31–47.

[5] Stefan Bilbao, Alberto Torin, Paul Graham, James Perry, and Gordon Delap. 2014. Modular physical modeling synthesis environments on GPU. In *ICMC*.

[6] Stefan Bilbao and Craig Webb. 2012. Timpani drum synthesis in 3D on GPGPUs. In *Proc. of the 15th Int. Conference on Digital Audio Effects (DAFx-12), York, United Kingdom*.

[7] Stefan D Bilbao. 2009. *Numerical sound synthesis*. Wiley Online Library.

[8] David Blythe. 2008. Rise of the Graphics Processor. *Proc. IEEE* 96, 5 (2008), 761–778. https://doi.org/10.1109/JPROC.2008.917718

[9] Robert Bristow-Johnson. 1996. Wavetable synthesis 101, a fundamental perspective. In *Audio Engineering Society Convention 101*. Audio Engineering Society.

[10] John Charles Butcher and Nicolette Goodwin. 2008. *Numerical methods for ordinary differential equations*. Vol. 2. Wiley Online Library.

[11] John Chowning and David Bristow. 1986. FM theory and applications. *By Musicians for Musicians* (1986).

[12] NVIDIA Corporation. 2009. NVIDIA Fermi Compute Architecture Whitepaper. (2009).

[13] R. Courant, K. Friedrichs, and H. Lewy. 1928. Über die partiellen Differenzengleichungen de mathematischen Physik. *Math. Ann.* 100 (1928), 32–74.

[14] Richard Courant, Kurt Friedrichs, and Hans Lewy. 1967. On the partial difference equations of mathematical physics. *IBM journal of Research and Development* 11, 2 (1967), 215–234.

[15] Travis Desell, Anthony Waters, Malik Magdon-Ismail, Boleslaw K. Szymanski, Carlos A. Varela, Matthew Newby, Heidi Newberg, Andreas Przystawik, and David Anderson. 2010. Accelerating the MilkyWay@Home Volunteer Computing Project with GPUs. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 276–288.

[16] J David Eisenberg and Amelia Bellamy-Royds. 2014. *SVG essentials: Producing scalable vector graphics with XML.* " O'Reilly Media, Inc.".

[17] Gwynne Evans, Jonathan Blackledge, and Peter Yardley. 2012. *Numerical methods for partial differential equations.* Springer Science & Business Media.

[18] Benedict R. Gaster, Nathan Renney, and Carinna Parraman. 2019. Fun with Interfaces (SVG Interfaces for Musical Expression). In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design* (Berlin, Germany) *(FARM 2019)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3331543.3342579

[19] Brian Hamilton and Craig J Webb. 2013. Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid. *Proc. Digital Audio Effects (DAFx), Maynooth, Ireland* (2013), 336–343.

[20] Bill Hsu and Marc Sosnick-Pérez. 2013. Realtime GPU audio: Finite difference-based sound synthesis using graphics processors. *Queue* 11, 4 (2013), 40–55.

[21] Robert Georg Ilgner. 2013. *A comparative analysis of the performance and deployment overhead of parallelized finite difference time domain (FDTD) algorithms on a selection of high performance multiprocessor computing systems.* Ph.D. Dissertation. Stellenbosch: Stellenbosch University.

[22] Robert H Jack, Adib Mehrabi, Tony Stockman, and Andrew McPherson. 2018. Action-sound Latency and the Perceived Quality of Digital Musical Instruments: Comparing Professional Percussionists and Amateur Musicians. *Music Perception: An Interdisciplinary Journal* 36, 1 (2018), 109–128.

[23] Dan Lavry. 2004. Sampling Theory For Digital Audio. *Lavry Engineering, Inc. Available online: http://www. lavryengineering. com/documents/Sampling_Theory. pdf (checked 24.5. 2010)* (2004).

[24] Philip McCord Morse and K Uno Ingard. 1986. *Theoretical acoustics.* Princeton university press.

[25] Steven Muchnick et al. 1997. *Advanced compiler design implementation.* Morgan kaufmann.

[26] NESS. 2019. faustmanual. https://www.ness.music.ed.ac.uk/project. Accessed: 2019-08-11.

[27] John Nickolls and William J Dally. 2010. The GPU computing era. *IEEE micro* 30, 2 (2010), 56–69.

[28] Marius Gerorge Onofrei, Silvin Willemsen, and Stefania Serafin. 2021. Real-time implementation of a friction drum inspired instrument using finite difference schemes. In *Proceedings of the 24th International Conference on Digital Audio Effects (DAFx20in21)*, Gianpaolo Evangelista and Nicki Holighaus (Eds.), Vol. 2. 168–175. https://dafx2020.mdw.ac.at/ 24th International Conference on Digital Audio Effects, DAFx20in21 ; Conference date: 08-09-2021 Through 10-09-2021.

[29] D. C. Rapaport. 2020. GPU molecular dynamics: Algorithms and performance. *arXiv: Computational Physics* (2020).

[30] Junuthula Narasimha Reddy and David K Gartling. 2010. *The finite element method in heat transfer and fluid dynamics.* CRC press.

[31] Michael Renardy and Robert C Rogers. 2006. *An introduction to partial differential equations.* Vol. 13. Springer Science & Business Media.

[32] Harri Renney, Benedict R Gaster, and Tom Mitchell. 2019. OpenCL vs: Accelerated finite-difference digital synthesis. In *Proceedings of the International Workshop on OpenCL.* 1–11.

[33] Harri Renney, Benedict R Gaster, and Thomas J Mitchell. [n.d.]. There and Back Again: The Practicality of GPU Accelerated Digital Audio. ([n. d.]).

[34] Yasubumi Sakakibara. 1992. Efficient learning of context-free grammars from positive structural examples. *Information and Computation* 97, 1 (1992), 23–60.

[35] JO Smith. 1997. Acoustic modeling using digital waveguides. *Musical Signal Processing* 7 (1997), 221–264.

[36] Olivier Terzo, Karim Djemame, Alberto Scionti, and Clara Pezuela. 2019. *Heterogeneous Computing Architectures: Challenges and Vision.* CRC Press.

[37] Alexis Thibault. 2019. *Wind Instrument Sound Synthesis through Physical Modeling.* Ph.D. Dissertation. ENS Paris-Ecole Normale Supérieure de Paris; Sorbonne Université; Inria . . . .

[38] Jim Turley. 2014. *Introduction to Intel® Architecture.* Technical Report. Intel.

[39] Craig J Webb and Stefan Bilbao. 2015. On the limits of real-time physical modelling synthesis with a modular environment. In *Proceedings of the International Conference on Digital Audio Effects*. 65.

[40] Thomas Weber. 2014. *Micropolygon Rendering on the GPU.* Ph.D. Dissertation.

[41] Silvin Willemsen. 2021. *The Emulated Ensemble: Real-Time Simulation of Musical Instruments using Finite-Difference Time-Domain Methods.* Ph.D. Dissertation. Aalborg University Copenhagen.

[42] Silvin Willemsen, Nikolaj Schwab Andersson, Stefania Serafin, and Stefan Bilbao. 2019. Real-time control of large-scale modular physical models using the sensel morph. In *16th Sound and music computing conference*. Sound and Music Computing Network, 151–158.

[43] Silvin Willemsen, Stefan Bilbao, Michele Ducceschi, and Stefania Serafin. 2021. A physical model of the trombone using dynamic grids for finite-difference schemes. In *Proceedings of the 24th International Conference on Digital Audio Effects DAFx20in21 (Proceedings of the International Conference on Digital Audio Effects, Vol. 2)*, Gianpaolo Evangelista and Nicki Holighaus (Eds.). 152–159.

[44] Silvin Willemsen, Stefan Bilbao, and Stefania Serafin. 2019. Real-time implementation of an elasto-plastic friction model applied to stiff strings using finite difference schemes. In *22nd International Conference on Digital Audio Effects*.

[45] Silvin Willemsen, Stefania Serafin, Stefan Bilbao, and Michele Ducceschi. 2020. Real-time Implementation of a Physical Model of the Tromba Marina. In *17th Sound and Music Computing Conference*. 161–168.

[46] Victor Zappi. 2017. *The Hyper Drumhead: Making Music with a Massive Real-time Physical Model.* Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

[47] Victor Zappi, Andrew Allen, and Sidney S Fels. 2017. Shader-based physical modelling for the design of massive digital musical instruments.. In *NIME*. 145–150.