Performance, Portability and Productivity for Room Acoustics Codes

Larisa C. Stoltzfus

Master of Science
School of Informatics
University of Edinburgh
2017
Abstract

The parallel programming landscape is becoming so vast and complex that developing codes which are performant, portable and easily programmable tends to require compromising somewhere. Current solutions have focused on abstracting away parallelisation instructions from algorithms using libraries to hide details of hardware specific code and low-level syntax. However, many of these solutions have been tested only on simple benchmarks, are too low-level for productive use or focus more on hardware specific optimisations than specific codes. This project proposes a solution developed from the bottom up, beginning with the algorithm of interest - a room acoustics simulation for modeling sound in an enclosed 3D space - and developing a performance portable and productive framework from it.

Such a framework has been developed with the flexibility to swap in optimisations and data abstractions. The performance, portability and productivity of this framework improves on or is comparable to native and new implementations of the original benchmark it is based on. Tests were run across six different platforms (CPU, two NVIDIA GPUs, two AMD GPUs and Xeon Phi) comparing four implementations (CUDA, OpenCL, targetDP library and the new abstracted framework). Results showed that the abstract framework is performant within 23% of the best version on each of the platforms in the worst instance. It also runs 40% faster (45% when optimised) on a previously inaccessible platform as compared to the original benchmark on the platform it was optimised for. Additionally, comparisons of more advanced versions of the benchmarks with the original versions determined that there is a similar performance profile when the room acoustics algorithms become more complex.

This research has shown that it is possible to develop more productive and performance portable codes for room acoustics simulations which simplify the problems of writing programmable and performant code across different platforms. This work can be extended to other physical simulations using similar algorithms. Going forward, research could also be done to extend this work into a DSL for grid-based simulations (like room acoustics). Furthermore, this DSL could be brought into existing frameworks to compile HPC simulations to architecture specific optimised code.
Acknowledgements

First of all, a very big thank you to all of my supervisors: Alan Gray, Christophe Dubach and Stefan Bilbao. I am also grateful for all the help I’ve received from the NESS group, in particular Brian Hamilton and Craig Webb for providing benchmarks, answering questions and discussing previous work on the room acoustics simulations. Thank you very much to Michel Steuwer for his help on many things especially OpenCL, skeleton frameworks, C++ and general academic advice. Thanks also goes to Alberto Magni for help with profiling on AMD machines as well as to Amanda Grimm and Murray Cole for all the thankless things they’ve done to make this year go smoothly. Finally, thank you to Hugh and Qubit, for reminding me to step away from my computer now and again and remember the other important things in life.

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Larisa C. Stoltzfus)
# Table of Contents

1 Introduction ........................................... 1

2 Background ........................................... 5
   2.1 Physics of Room Acoustics .......................... 5
   2.2 Simulating Room Acoustics and Stencils .......... 6
   2.3 Relevant Hardware and Software ................. 9
      2.3.1 Hardware .................................. 9
      2.3.2 Software .................................. 11
   2.4 Case Study: NESS ................................ 12
   2.5 Related Work .................................... 13
      2.5.1 Low-Level Work ............................. 13
      2.5.2 Higher-Level Approaches .................... 14
      2.5.3 Limitations of Current Approaches ........ 15

3 Composable Room Acoustics Stencils .................. 17
   3.1 Framework Design ................................ 17
   3.2 Exploring the Design Space ...................... 19
      3.2.1 Thread Configuration ....................... 19
      3.2.2 Abstracting Memory Layouts ............... 20
      3.2.3 Memory Optimisations ...................... 20

4 Methodology ........................................... 23
   4.1 Languages, Frameworks, Tools and Benchmarks ... 23
      4.1.1 Languages and Frameworks .................. 24
      4.1.2 Tools ..................................... 24
      4.1.3 Benchmarks ................................ 24
   4.2 Code Structure .................................. 25
   4.3 Specific Platform, Compiler and Drivers Overview 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Comparison Analysis</td>
<td>28</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Metrics</td>
<td>28</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Scripts</td>
<td>29</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Room Sizes</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Testing</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Code Development</td>
<td>30</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Simple Room Benchmarks</td>
<td>30</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Developing abstractCL</td>
<td>33</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Advanced Room Benchmarks</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Memory Optimisation Work</td>
<td>41</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Image Memory Development</td>
<td>41</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Local Memory Development</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Experimental Results</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Data Layouts</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Simple Benchmark Comparison</td>
<td>47</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Performance</td>
<td>47</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Portability</td>
<td>52</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Productivity</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Optimisations</td>
<td>53</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Thread Configuration Settings</td>
<td>54</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Local Memory Experimentation</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Bandwidth and Computing Power Evaluation</td>
<td>60</td>
</tr>
<tr>
<td>5.5</td>
<td>Advanced Codes Comparison</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Critical Analysis</td>
<td>68</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Project Improvement</td>
<td>68</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Outstanding Issues</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Future Work</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td>73</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Modern computing has become so pervasive that it has begun influencing and directing nearly all areas of life. Computer simulations are now required to predict the behaviour of people, machines, physical processes and more. Where high performance computing (HPC) was once a niche field predominantly utilised by scientists, techniques in this area are now becoming commonplace. Computers are moving towards parallel architectures as greater performance can no longer be achieved through a single core. As such, the ability to program simulations that can perform well across different architectures is of growing importance.

The types of architectures available are changing and increasing to meet demands for performance. To avoid rewriting codes, portability across old and new platforms must be considered. Currently, most HPC codes compile down to run on multiple CPU cores, but in order to take advantage of new, more performant hardware groups may need to maintain multiple code bases. Furthermore, even where codes can port from one architecture to another, there is no guarantee they will retain the same performance on the new platform. Rewriting and retuning codes every time a more performant platform becomes available is not a long term strategy in any field.

Developing a method that would allow any codes to run performant and in parallel (not to mention be easily programmable) on any platform would be a monumental task. Instead, the purpose of this project will be to evaluate the scope of this problem and develop a solution using a subset of codes. The idea is that by looking at the problems of performance portability and productivity from a bottom-up approach, tools could be developed with specific algorithms in mind which could then be expanded to accommodate a wider range of codes. This project will focus on room acoustics simulations, which use algorithms representative of a large number of other physical models.
Chapter 1. Introduction

Room acoustics simulations were developed to model sound waves in an enclosed three dimensional space. The simulations use first physical principals to represent the properties of sound as it moves through space and time. These “room codes” use grids as data types that update values via very a commonly used nearest-neighbours technique. Their output can provide composers or architects the ability to hear what a composition or noise would sound like in a space without actually being there or having built it.

This project will undertake the development of a simple, flexible abstraction framework based on an acoustical simulation benchmark. The flexibility comes through the use of templates and macros to swap in and out different implementation and optimisation choices. Other implementations of the benchmark will be developed (or updated) for comparison to this abstracted framework and for testing performance portability and programmability across different platforms. Advanced versions of the same benchmark will also be run across a subset of the same platforms to provide information about the effects of changing the basic algorithm of the original simulation.

Results will show that a portable and more productive version of a simple room acoustics simulation is possible. It will be shown that such a framework is comparable in performance to other native versions, running at worst 23% slower than the most optimal version of the code tested on any given platform. This difference will drop to 7% when considering larger room sizes (ie. more typical of a real simulation). Across platforms, the abstracted framework version developed will show an improvement of 40% in performance in comparison to the original benchmark on the platform is was optimised for. This improvement will increase to 45% when considering the optimised version of the abstracted framework on the new platform. Other results will show that data layouts and other optimisations can have an impact on performance, although the codes generally were already quite close to utilising most of the memory bandwidth available. Advanced versions of the simple benchmark will also show a similar performance profile to the simpler ones.

This paper will first cover the necessary background for this project, including the physics of room acoustics, physical simulation modeling, hardware platforms, relevant parallel frameworks and previous work in Chapter 2. Chapter 3 will present the new tunable abstract framework, which allows for different abstractions and optimisations to be swapped in and out. Methodology will be outlined in Chapter 4, which covers languages, libraries, tools, benchmarks, code structure, platform details, testing frameworks and code development. Chapter 5 will analyse the results for each of
the comparisons between versions of the code, as well as discuss the abstraction, optimisation and advanced codes investigations done. Finally, the paper will conclude in Chapter 6, giving a brief summary of findings, reflecting critically on the project and contemplating future work.
Chapter 2

Background

This project involves running variations of different implementations of specific room acoustics simulations codes across different types of computing hardware, but will build on ideas that have been brought forward already. In order to understand these benchmarks, the physics of room acoustics is introduced with a brief overview of what principles are involves in simulating sound. Bridging the gap between the physics and the computation is also discussed, as well as a brief foray into the group who has developed the simulations themselves. As well as the focus on the acoustic side, it is also important to understand what kinds of different hardware will be used for this project in addition to commonly used languages and frameworks available to program them. Previous work has shown it is possible to develop portable and productive abstractions for grid based codes, but so far results for large-scale HPC simulations have been limited.

2.1 Physics of Room Acoustics

Sound is a property created when a physical object vibrates. Energy from the vibrations propagate through the air in the form of longitudinal waves, which is what our ears pick up on and interpret as sound. These waves can be modelled using first physical principals[1][2]. The 3D wave equation can be seen in Equation 2.1 and it is a partial differential equation describing the movement of sound in three dimensions. For this project, the movement of the sound waves is restricted to contained rooms. So in such a case, the \( x,y,z \) coordinates indicate the spatial coordinates inside a room, \( t \) indicates the time, \( \psi \) represents the amplitude of the wave which is proportional to the acoustic pressure (the local difference from ambient pressure) and \( c \) is the speed of a
sound wave through air in the local conditions. Values must be updated depending on circumstances, for instance when encountering obstacles or walls. This can be done with boundary conditions at specific spatial coordinates of the room.

\[
\frac{\partial^2 \psi}{\partial t^2} = c^2 \left( \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} + \frac{\partial^2 \Psi}{\partial z^2} \right) \quad (2.1)
\]

The 3D wave equation on its own would only be useful for modelling sound at very small distances. For larger distances, energy is lost from sound waves as they travel through air, due to a property known as viscosity in non-ideal gases. Generally, the main variable that affects energy loss in air is temperature. Equation 2.2 shows the effect that viscosity have on the 3D wave equation. There, both \(c\) and \(\alpha\) are constants, the combination \(c\alpha\) is a “damping component” the derivation of which is beyond the scope of this paper[1][2]. In this project, simulations were run using both the simple 3D wave equation (see Section 4.6.1) and more advanced simulations with viscosity (see Section 4.6.3).

\[
\frac{\partial^2 \psi}{\partial t^2} = c^2 \nabla^2 \psi + c\alpha \nabla^2 \frac{\partial \psi}{\partial t} \quad (2.2)
\]

### 2.2 Simulating Room Acoustics and Stencils

In order to model the acoustics in a room, the physical behaviour of the system (ie. the solutions to the 3D wave equation) must be broken down into a form where small, distinct sub-sections of the room can be computed on. That is, what is a continuous equation must be reformed as a discrete one with distinct points. This is done through a process known as discretisation. Given that the 3D wave equation is a partial differential equation, there are numerous numerical methods that can be used to solve the solution for a three dimensional space. One of these solutions is the finite difference method, which uses small approximations of rates of change around a point of interest[3]. Grid spacing determines how small the granularity of the points representing the room are, which also constrains the accuracy of the finite difference approximations. Room acoustics simulations are only one example of a wide range of physical processes that can be modelled this way, including electromagnetics[4], lattice quantum chromodynamics[5][6] and fluid dynamics[7].

As the sound waves in the room travel outward from their source in different directions during each timestep, the grid must be updated to reflect the changing conditions. Points in the grid are updated according to behaviour of the wave’s traversal, which
means that a grid point depends on its neighbouring grid points to pass information as time moves forward. This process of updating points from nearest neighbours, known as a “stencil,” can be seen more clearly in Figure 2.1, which depicts a seven point stencil where the values of the blue points are added to the pink point in the centre.

The discretised version of the 3D wave equation is shown in Figure 2.3[8]. In this equation, \( \psi_{n,x,y,z} \) represents the amplitude of the pressure of the sound wave at the current timestep \( n \) at the grid point \( x,y,z \). The next timestep is represented by \( n+1 \), while \( n-1 \) represents the previous one.

\[
\psi_{n+1,x,y,z} - 2\psi_{n,x,y,z} + \psi_{n-1,x,y,z} = L^2(\psi_{n,x-1,y,z} + \psi_{n,x+1,y,z} + \psi_{n,x,y-1,z} + \psi_{n,x,y+1,z} + \\
\psi_{n,x+1,y,z+1} + \psi_{n,x,y,z+1} - 6\psi_{n,x,y,z})
\] (2.3)

Equation 2.4 shows equation 2.3 rewritten in the form where the latest grid timestep is being updated from the two previous ones. This is the form the algorithm takes in the room acoustics simulation codes used in this project.

\[
\psi_{n+1,x,y,z} = (2 - 6L^2)\psi_{n,x,y,z} + L^2(\psi_{n,x-1,y,z} + \psi_{n,x+1,y,z} + \psi_{n,x,y-1,z} + \psi_{n,x,y+1,z} + \\
\psi_{n,x+1,y,z+1} + \psi_{n,x,y,z+1} - 6\psi_{n,x,y,z})
\] (2.4)

Stencils come in many different shapes and sizes. They can be summations involving only those values next to a particular point (ie. six neighbours in the three dimensional case) or have more length, direction or weights. Length and weight values can also vary by direction. In Figure 2.3, the stencil is \( \psi_{n,x-1,y,z} + \psi_{n,x+1,y,z} + \psi_{n,x,y-1,z} + \psi_{n,x+1,y,z+1} + \psi_{n,x,y,z+1} - 6\psi_{n,x,y,z} \)
\[ \psi_{x,y+1,z}^n + \psi_{x,y,z-1}^n + \psi_{x,y,z+1}^n. \]

Here it can clearly be seen how the state of the simulation at one grid point influences its neighbours at a given timestep. For many physical simulations, stencils can provide more accuracy or convergence more quickly, but this is also dependent on the grid spacing[9][8]. That is, for grids with smaller spacing, larger stencils may not provide a large improvement in accuracy. Although stencil applications have much in common and simple abstractions can be made for them, for many simulations their use is complicated slightly through differences in the algorithms using the stencil. In the case of room acoustics, multiple timesteps and boundary conditions (as well as other physically influenced variations) complicate the abstraction of these stencils.

In figures 2.2 and 2.3, examples of stencils in algorithm form are shown. Figure 2.2 shows a six point stencil used for a three dimensional grid. In the figure, the variable \( cp \) is a flattened index of a 3D grid. The values \( tdx,tdy,tdz \) represent the 3D grid indices. \( grid \) represents the grid in its current state and \( gridTS1 \) represents the grid in the previous timestep. Here the stencil is using a six point sum, which mean it adds up its six neighbouring points in each direction on a 3D grid.

6 Point Stencil for a 3D grid

\[
grid[cp] = gridTS1[tdx+1][tdy][tdz] + gridTS1[tdx-1][tdy][tdz] \\
+ gridTS1[tdx][tdy+1][tdz] + gridTS1[tdx][tdy-1][tdz] + \\
gridTS1[tdx][tdy][tdz+1] + gridTS1[tdx][tdy][tdz-1]
\]

Figure 2.2: Six Point Stencil Example in 3D

Figure 2.3 shows a leggy stencil accessing three points in each direction for a two dimensional grid. In the figure, \( grid \) is a 2D array which is using a leggy stencil. Similar to Figure 2.2, \( cp \) is a flattened index and the values \( tdx,tdy \) represent the 2D grid indices. This stencil accesses neighbouring points up to three points in all cardinal directions of the grid. As can be seen, leggy stencils add many more memory accesses to the algorithm.
2.3 Relevant Hardware and Software

The selection of hardware for this project focuses on CPUs and accelerators and the languages and frameworks used are to program these architectures. Within the realm of accelerators, both GPUs and the Xeon Phi were selected for use as they provide different benefits and limitations. The focus also is much more on these accelerators than CPUs as accelerators have shown more performance improvement than single CPU chips (even multi-cores). The frameworks used vary more for GPUs than the Xeon Phi as the Xeon Phi architecture is much closer to the CPU. However, the languages used are the same across all platforms: C/C++.

2.3.1 Hardware

The types of hardware used in this project include: CPU, GPU, and Xeon Phi. CPUs have historically been the main chips used for running HPC simulations. Their architectures though were originally optimised for sequential codes. Since the early 2000s, clock speeds of CPUs have stopped increasing due to physical constraints and instead multi-core chips have dominated the market[10]. Tangentially, other architectures have either been repurposed or come onto the scene, in particular accelerators such as GPUs and Xeon Phi chips.

2.3.1.1 GPUs

Graphics Processing Units, as the name suggests, were originally designed for accelerating computations for displaying graphics. However, in the last decade they have

\[
\text{grid}[cp] = \text{gridTS1}[tdx+1][tdy] + \text{gridTS1}[tdx-1][tdy] \\
+ \text{gridTS1}[tdx][tdy+1] + \text{gridTS1}[tdx][tdy-1] \\
+ \text{gridTS1}[tdx+2][tdy] + \text{gridTS1}[tdx-2][tdy] \\
+ \text{gridTS1}[tdx][tdy+2] + \text{gridTS1}[tdx][tdy-2] \\
+ \text{gridTS1}[tdx+3][tdy] + \text{gridTS1}[tdx-3][tdy] \\
+ \text{gridTS1}[tdx][tdy+3] + \text{gridTS1}[tdx][tdy-3]
\]
increasingly been repurposed to run other types of codes in what is known as GP-GPU - “general purpose” GPU - programming[10]. Because of their original specific purpose though, the architecture of GPUs have evolved to be markedly different from CPUs. Instead of having a few cores, they have hundreds of what are known as “compute units” (or streaming multiprocessors), which contain multiple stripped down cores that run in parallel working on the same code (with different data). They have much higher memory bandwidth than CPUs, however each of their compute units has a much smaller cache. CPUs are built for low latency whereas GPUs are built for high throughput while masking the higher latency with more threads. One major limitation of GPUs is that data must be copied to them over a PCI bus and then back when finished, which can be a slow process. This means that simulations that need to frequently copy data to and from main memory will not fare well. However, for computationally intensive simulations which compute over the same data and may have memory bandwidth constraints, GPUs tend perform quite well[11][10][12].

Programming a GPU is more complicated than a CPU as large numbers of threads are used to compute over the data and the correct configuration for this must be manually selected prior to running a particular code. The compute units work like SIMD (“Single Instruction Multiple Data”) chips, where groups of threads execute the same line of code in parallel[13]. The total number of threads must be mapped appropriately for a given kernel (similar to a function) so that as many compute units can be used as possible and can run in a way that both covers the data and exploits the memory layout. This is done through workgroup and workitem settings.

The overall idea is that a kernel that is spawned on a GPU consists of an array of workgroups and a workgroup consists of a group of workitems and the workitems map down to individual threads. Workitems must span the entire domain space and workgroups are made up of groups of these workitems. These overall settings define how many threads are allocated per group and in which “direction” they access memory from. They are also constrained by architecture. When data is retrieved from global memory, it is pulled consecutively across lines in the same direction as the data is stored. As memory is retrieved in chunks, it is important to retrieve data nearby to other data that will be used soon after in a computation in order to avoid delays in having to perform more memory accesses than necessary. This process is known as memory coalescing. Correct settings for how many workitems are assigned to a workgroup can ensure memory access are coalesced if their settings are optimised to be larger in the same direction as the memory being accessed.
There are several types of memory in a GPU, the main ones used in optimisations include global, image, local and register memory (or for NVIDIA architecture: global, texture, shared and private)[13]. Global memory is the main memory that data is stored in. When data is copied over from the CPU, it is copied to global memory. Image memory is a special type of memory that is behaves more like a cache and is only accessible for read-only or write-only data, depending on the version and the framework being used. Accesses to this type of memory are separate to other caches and can be faster due to the exploitation of locality and the fact that it uses fewer GPU cycles. Local memory is local to a compute unit, so threads that are in a workgroup on the compute unit can all access it. This allows for optimisations like tiling. Finally, register memory is memory that is private to a workitem and will not be discussed further in this project.

### 2.3.1.2 Xeon Phis

The Xeon Phi was developed by Intel as an alternative accelerator to the GPU[14]. One of the main benefits of the Xeon Phi is that it uses the same instruction set (X86) as the majority of other CPUs. This means that (in theory) HPC codes developed to run on CPUs could be more easily ported to Xeon Phi chips. There are fewer cores on the Xeon Phi than on a GPU, however data does not need to be transferred to and from separate memory. Another big difference between the architectures is that there is a wider vector instruction set on the Xeon Phi. The Xeon Phi currently straddles both the architecture and the performance of the CPU and GPU. Codes that can be vectorised and scale to a large number of cores will see a performance boost on the Xeon Phi, but even with good use of vectorisation the performance will not likely increase as much as could be gleaned from a GPU[11]. There is no concept of workgroup and workitem settings on a Xeon Phi (just number of threads) and the same languages and frameworks that are used for programming CPUs can be used, so long as there are compilers that support the platform.

### 2.3.2 Software

There are a vast array of parallel frameworks available, but most are not portable across platforms. There are two main frameworks that can be used with codes to enable programming on GPUs: CUDA and OpenCL. CUDA is developed by NVIDIA and only works on their GPUs. The NESS codes have originally only used CUDA for
their GPU ported simulations. CUDA is much more high-level, accessible and well supported in comparison to OpenCL, but OpenCL is far more portable - able to be used across GPUs, Xeon Phis, CPUs and other architectures. OpenCL suffers from a large amount of boilerplate code and is therefore more complicated to program in. On NVIDIA GPUs, OpenCL is also restricted to version 1.2, even though a much higher version exists. For CPUs there is a much richer array of frameworks available, the two main ones being MPI and OpenMP. As the Xeon Phi has a more similar architecture to CPU chips, MPI and OpenMP can be used there as well. For this project, only OpenMP was explored with CPU and Xeon Phi platforms. All of these frameworks - CUDA, OpenCL and OpenMP - must be used in conjunction with another language: this project primarily uses C, except for the abstracted version which was written in C++.

2.4 Case Study: NESS

One research group struggling with the issues of managing multiple code bases and creating performance portable and programmable code is NESS, or the “Next gEneration Sound Synthesis” project at the University of Edinburgh. The aim this project is to develop a physical sound synthesis model for musical instruments in different acoustic settings[15]. NESS provides an improvement over previous models, which lack flexibility or use sampled data and thus have a poorer quality of sound. The model can be used to simulate musical instruments, room acoustics or combinations of both. For simulating only instruments the model can be run on a CPU. However, room acoustics modeling is much more computationally expensive and is currently ported separately to run on GPUs. Ideally the model could be run for either of these scenarios from the same code base whilst retaining performance across different architectures and without having to be updated for new architectures.

Currently the code base is divided into two main branches: room acoustics codes and instruments codes. Instruments codes are written in MATLAB and mainly run on the CPU. Room acoustics codes are only ported to run in C and CUDA on NVIDIA GPUs. Additionally, in order to parallelise the code in CUDA, the NESS team liaised with members of EPCC. This project is an attempt to address these issues of having to maintain multiple code bases and reliance on outside expertise for re-tuning to new, more performant architectures. In doing so, the project will focus exclusively on the room acoustics side of the NESS codes and the benchmarks used are derived from their
2.5 Related Work

Much work has already been done to develop strategies to tackle the performance portability problem, without compromising productivity. At the low-level end, meaning libraries or codes that are closer to machine level and use fewer abstraction layers, simple substitutions can swap into existing codes which have already been created to allow portability across platforms that were previously inaccessible. On the higher level end, where high indicates a level of abstraction much more removed from the details of computation or architecture, there are a number of frameworks, libraries, languages and tools available. No work in either of these areas specifically targets room acoustics codes as many of the frameworks are much less specialised or are still in the early stages of development with limited testing.

2.5.1 Low-Level Work

On the low-level end, EPCC has developed a library to abstract away thread and instruction level parallelism for lattice-based codes using C preprocessor directives and macros[6]. This library - known as “targetDP” - can be integrated into applications to allow them to run on NVIDIA GPUs, CPUs and Xeon Phis. It is currently being implemented into the complex fluid simulation package Ludwig, which uses grids in a similar way to room acoustics simulations. The main appeal of this framework is that it is lightweight (only three extra files are needed) and maintains a close syntax to C codes, thus pre-existing codebases can be easily updated to use it.

The library essentially has a separate implementation with the same syntax for each of the versions necessary to run on multiple architectures and uses separate flags in the Makefile to inform which version to compile. On CPUs and Xeon Phi chips, the codes run using OpenMP threads. On NVIDIA GPUs, the codes use CUDA. The code can be swapped between the two using pre-defined macros that access different definitions according to compilation instructions. This allows for more optimised code for NVIDIA GPUs because only CUDA will be used, however makes the library less portable to non-NVIDIA GPUs.
2.5.2 Higher-Level Approaches

Higher level approaches focus more on distinct layers of abstraction that are far removed from the original codes and which generally also aim to support a higher level of programmability. These approaches include: parallel algorithmic skeletons, code generators, DSLs, autotuners, combinations therein and other types of frameworks. Many of these frameworks also focus on decoupling layers of functionality, which allows for more flexibility with different implementations and architectures. As many of these frameworks are still in early stages of development, there are limitations in using them with pre-existing code bases.

2.5.2.1 Algorithmic Skeletons

Skeleton frameworks are a large subgroup of abstraction frameworks developed for enhancing productivity. These skeletons focus on the idea that many parallel algorithms can be broken down into pre-defined building blocks[16]. Thus, an existing code could be embedded into a skeleton framework that already has an abstraction and API built for that algorithm type, such as a stencil. These frameworks then simplify the process of writing complex parallel code (like OpenCL) by providing an interface which masks the low-level syntax and boilerplate. A number of these skeleton frameworks have been developed, many of which are intended for grid-based applications, however few have been tested on larger simulation models. Additionally, many of them lack 3D functionality, such as Skepu[17] and SkelCL[18] which only support 1D and 2D stencils. Some of the libraries also only target particular architectures - for example PSkel, which does support 3D stencils, but only ports to NVIDIA GPUs[19].

2.5.2.2 Code Generators

A code generator can either be a type of compiler or more of a source to source language translator. They are a promising area in this field of research given that their modularity allows for flexibility between languages and platforms. A group in the University of Edinburgh’s Informatics department has developed a modularised code generation system using a collection of re-write rules which automatically tunes for a particular architecture[20]. This framework results in target architecture tailored OpenCL code re-written from the top level down. Petabricks is another example of a code generator, which is actually a language and a compiler capable of auto-tuning over multiple pre-existing implementations of algorithms to tailor to a specific
2.5. Related Work

hardware[21]. Kokkos and SYCL are two other code generators that are gaining popularity, which compile down to CUDA and OpenCL respectively, and are tailored more towards general use and ease of programmability[22][23]. Kokkos also has a sophisticated memory mapping pattern for optimising codes. SYCL focuses more on bringing a simplified interface to OpenCL in C++.

2.5.2.3 Other Higher-Level Approaches

Other higher-level approaches include functional DSLs with auto-tuning[24], rewrite rules[25], skeleton frameworks combined with auto-tuning[26], code generators and many others including the examples below. Liquid Metal is a project started at IBM to develop a new programming language purpose built to tailor to heterogeneous architectures[27]. Exastencil is a DSL developed by a group at the University of Passau that aims to create a layered framework that uses domain specific optimisations to build performant portable stencil applications[28]. Halide is another functional DSL with auto-tuning that specialises in abstracting stencils by separating algorithm from execution[29]. Both Exastencil and Halide could be potentially be quite useful for room acoustics simulations.

2.5.3 Limitations of Current Approaches

While low-level and high-level frameworks take fairly different approaches, neither are without problems. For TARGETDP, implementation is fairly straightforward, however its use of preprocessor directives means code readability and ease of debugging suffers. Productivity while developing these codes with this implementation will also likely not show much improvement, as the syntax of the framework is still quite low-level. Skeletons enable ease of programmability and portability, but they alone are not enough to produce performance-portable code especially as they are often tied to a framework or architecture. A framework that remains de-coupled from specific implementations would avoid this problem. For the higher-level code generation frameworks, results look promising in comparison to hand-tuned optimisations, though generally only small benchmarks have been tested or broad domains focused on. Some of the stencil-specific frameworks (like Halide) also focus mainly on images, simple stencils or other non-HPC specific domains. Many also often focus on getting good performance of particular hardwares instead of the algorithms that would use the framework. These limitations mean that there currently are no DSLs or frameworks developed that
have been shown to give room acoustics simulations good performance, portability and productivity.
Chapter 3

Composable Room Acoustics Stencils

The focus of this project is on investigating the feasibility of developing a performance portable and programmable framework for a room acoustics simulation. The development of this new framework will be built from the bottom up using a room acoustics benchmark. The new framework will have a performance portable abstraction layer, which will allow for flexibility of design by creating kernels for the room acoustics simulation on the fly. It will be tested through a comparison with other implementations of the same room acoustics code. The framework will then also be used to test out different memory layouts and optimisations.

3.1 Framework Design

As can be seen in Table 3.1, the current status of the methods of available for running codes on the platforms used in this project (CPUs, GPUs, Xeon Phi) vary in performance portability and productivity. The CUDA + NVIDIA GPU entry is highlighted in red to indicate that it was the only version in existence before this project. No versions are productive and portable across all architectures. A flexible abstraction framework has been developed to address this shortcoming. This framework allows for different types of grid implementations and optimisations to be run without having to completely rewrite the code each time. Additionally, it can be easily extendible to different types of room codes with varying algorithms, for example advanced codes with viscosity (though abstractions that could accommodate both simple and advanced codes were not actually implemented). The framework works by having a broad API available for different abstractions and implementations and pulling together those that are of interest or suite a particular platform on demand. In this way, it can “compose” a room
acoustic stencil on the fly. Optimisations can also be tested out easily in the framework to see what effect different changes had for different versions and platforms. Being able to run the same code over multiple platforms without writing and tuning multiple copies of the codes would improve productivity overall.

<table>
<thead>
<tr>
<th>Device</th>
<th>Performance Portable</th>
<th>Productive</th>
</tr>
</thead>
<tbody>
<tr>
<td>C + OMP</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C + CUDA</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C + OpenCL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>targetDP</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.1: Current status of a subset of implementations available for room acoustics codes. No framework currently exists that is both performance portable across platforms and productive. The CUDA + NVIDIA GPU combination is highlighted in red to indicate that it was the only version available before this project.

The framework was developed directly from the original room acoustics benchmark and then extended for different variations, versions and optimisations. The main benchmark used to tailor this framework models a simple 3D room with one source and one receiver. The simulation progresses through a set number of timesteps and simulates the propagation of sound waves as they move from a source to a receiver. Figure 3.1 shows a very simplistic version of the main algorithm in the benchmark. The benchmark calls the `UpdateRoom` function, which does the sound modeling in a room. This function gets the correct index of the point in the room, checks if the point is within the boundary of the room (ie. not a wall point) and then calculates the pressure of the neighbourhood using a stencil and updates the current point (at current index in the grid) accordingly. The function is called for as many timesteps as the user requires.
3.2 Exploring the Design Space

Figure 3.1: A simplified version of the original room acoustics benchmark, used as a starting point for the abstracted framework. The code loops over this `UpdateRoom` function accessing all points in the room for a set number of timesteps the sound is simulated for.

The actual framework version implementation is not too different from the pseudocode shown above in Figure 3.1 as it limits the impact of new changes given that they must conform to the abstract form. The design choice is intentional so that it maintains the same structure for all versions, but allows for flexibility in their implementation. That is, the main abstracted implementation will always look the same, but underneath the hood different implementations will use different functionality. This development method forces problems with abstracting these types of algorithms to emerge.

3.2 Exploring the Design Space

One of the main benefits of creating a tunable, flexible abstraction framework is that room acoustics codes would no longer need to be rewritten to test out new optimisations. Moreover, the optimisations could be swapped in and out from the same point limiting the room for error. A number of different data abstractions and optimisations were investigated with this new framework. The optimisations explored include: thread configuration settings, memory layouts and memory optimisations. Thread configuration is the optimisation of workgroup and workitem sizes, memory layouts dictate how a data type is designed and memory optimisations look at the use of non-global memory for improved performance.

3.2.1 Thread Configuration

Data accesses patterns can play a large role in performance of a code. Optimising workgroup and workitem settings can minimize performance hits by accessing memory contiguously. Using the abstraction framework here is less significant than other
potential optimisations as all that is being swapped in and out are values, however it is still a simple and useful setting to tune. A quick change of these values can test what effect different settings have to ensure memory is being accessed in a performant way.

### 3.2.2 Abstracting Memory Layouts

Abstractions and performance can often be at war with each other when developing codes. On the one hand, a code can be tuned for maximum performance, but the readability and programmability will suffer. On the other hand, a code could be very easy to implement and run, but would not be easily optimised due to layers of abstraction that obstruct flexibility. So compromises often need to be made when developing the appropriate data abstraction for a particular algorithm. One well-documented case of this is the comparison between an “array of structs” versus “structs of arrays”[30].

If data is arranged in a consecutive way using built-in types then performance when accessing these types from memory is optimal. However, often types like structs use padding which means that even if only a single value is extracted from a struct, the data type will be taking up more space than it normally would, so fewer values can be read at a time. However, some algorithms have unconventional memory accesses which might be better served by a struct that forces together values that need to be accessed together. Thus, whether an array of structs or a struct of arrays is more optimal depends on the algorithm and how its accessing the data.

The abstract framework provides the opportunity to explore this tension at the most basic level for room acoustics codes by allowing the data type representing the grid (and grid points) to be implemented in different ways that can be changed easily. Using abstractions that mask implementation, the performance effects of these different implementations can then be investigated. Experimenting with different data types can help determine what abstractions provide the most optimal memory accesses, as well as give a more in depth view of changes that may look simple on the surface, but have performance repercussions. Compromises may need to be made further down the line in developing flexible abstractions that do not hinder performance, particularly for more advanced room acoustics codes.

### 3.2.3 Memory Optimisations

Through the use of abstractions, it is easy to mask where data is actually being stored. When accessing a data point, it could be stored in a number of different places in dif-
3.2. Exploring the Design Space

different memories. Using an abstracted framework, it is straightforward to write a range
of different implementations for accessing data stored in different memory locations.
Though the optimisations in this project focus primarily on GPU memories, the frame-
work could be extended to include optimisations specific to other platforms that are
swapped in and out on a larger scale or for more complex layouts (ie. combinations
of memories used). This design allows for platform specific optimisations to be tested
out quickly and easily.
Chapter 4

Methodology

The methodology of this project describes how the structure of the project was built and fits together. This includes: languages and tools used, the code base layout, a brief overview of the hardware, a description of how the comparisons are done and the scripts were developed to glue the project together; and finally details about the room acoustics codes themselves as well as the optimisations developed for them. As one of the main goals of this project is to look at portability, a number of different frameworks are used, as well as different tools and benchmarks. The code structure is set up to be as simple and flexible as possible, allowing for batch runs, analysis and testing to happen from the same parent directory. The hardware platforms were selected to represent a variation, but also with slight variation on brand (ie. two different NVIDIA and AMD GPUs are used). The versions of the implementations themselves were written or updated in the various frameworks selected, details of the abstraction framework development are discussed as well as details provided about the data layouts and optimisations tested out.

4.1 Languages, Frameworks, Tools and Benchmarks

Results for this project use a number of different versions of the same room acoustics benchmark and thus require a number of external frameworks, scripts, tools and benchmarks for development and comparison. The languages and frameworks used were chosen based on their portability or accessibility for the platforms selected for this project. Scripting languages were chosen for running versions, aggregating and analysing data. Tools were used to find out more about how the codes run as well as provide a means of storing code changes. A well-known external benchmark was also
chosen for comparison on how well the room acoustics codes could be expected to perform.

### 4.1.1 Languages and Frameworks

Languages and frameworks were selected for the different versions of the room acoustics code and scripting languages were chosen to develop scripts to run batch processes as well as trim and analyse data. The frameworks versions selected for this project include CUDA, OpenCL, targetDP and the newly developed abstracted framework dubbed abstractCL. As frameworks cannot provide a stand alone piece of code, the languages C and C++ were used in conjunction with the frameworks. The CUDA, OpenCL and targetDP versions all use C in combination with their frameworks. The abstractCL version was written in a combination of C and C++ and utilises the C++ OpenCL bindings, which differ slightly from the OpenCL ones. Scripts for running versions were written in bash. Scripts for collecting, processing and displaying data were written in bash and R and are discussed in more detail in Section 4.4.2.

### 4.1.2 Tools

Two main external tools were used with this project: profiling applications and source control. Two profilers were used to extract information about runs: APP Profiler on AMD GPUs and nvprof on NVIDIA GPUs[31][32]. These profilers run from the command line and obtain information about codes running on GPUs, such as how much data is being read from and written to global memory, percentage of cache misses, etc. These profilers were used primarily to confirm how much memory bandwidth was actually achievable by codes run on the GPUs using CUDA and OpenCL. Git was used as a source control and regular code updates were checked in to track version changes. When version runs are completed, they are written to a data directory with the git ID of the last check-in. This allows runs to be easily traced to which version of the code made them.

### 4.1.3 Benchmarks

As well as profilers, a benchmark called “stream” was also used to give an approximation of the highest actual bandwidth possible (as opposed to the peak bandwidths advertised for architectures which are almost never obtainable) for the codes on par-
4.2 Code Structure

Stream is a very simple benchmark developed to give a rough idea of the best possible bandwidth a code could theoretically achieve without more in-depth optimisations. The benchmark itself consists of simple codes for adding, copying and multiplying from multiple arrays and returns the memory bandwidth calculated for each. Values in this report use the highest values of memory bandwidths returned by the benchmark.

4.2 Code Structure

The structure of the codebase was developed with batch processing in mind. Each version of the room simulation benchmark implementation was written in its own directory under src with its own Makefile. This can be seen in Figure 4.1 on the right, where abstract, opencl, cuda and targetDP represent the abstractCL, OpenCL, CUDA and targetDP versions of the room acoustics benchmark accordingly. The source directory then fits under the main project directory room_code, an overview of which can be seen on the left in the same figure.
Figure 4.1: Overall project structure (left) and version source code structure (right). The project structure is outlined on the left for the *room_code* directory and just the benchmark implementation source structure is on the right under *src*

Running an implementation (or batch running multiple implementations) is a straightforward process. From the directory of interest (for example, *src/cuda*) calling the Makefile places a compiled version of the binary executable in the separate agnostic directory *bin*. Separate implementation code directories all link to the same in-
clude files in the `include` directory at the same main level as `src` and `bin`. Calling 
/bin/BasicRoom then runs the code. Codes can be run in batch using scripts in the 
`scripts` directory, for example running `scripts/runAllAMD.sh` will run all those 
implementations that can be run on AMD platforms (ie. OpenCL and abstractCL) for 
the number of iterations specified in the script. There are scripts for running the same 
version multiple times as well as multiple versions any number of times and different 
scripts to target different platforms. When a code is run, it outputs timing information 
as well as room and receiver data into a directory in `data` (at the main level). In the 
data directory, the executable will create a directory based on the git ID and place 
output data in this directory. In this manner, codes can always be traced back to which 
version they are running. In addition to this, the output files include version information 
about the platform they are running on and their names include information about 
the room size, receiver location, version and other variables. An example of an output 
name for a room simulation would be: `5208efcbae/targetDP_CUDA-nvidia_kepler-
room-A256x256x202-S120x120x60x-R50x50x50-NF4410.bin`. For this run, the git ID 
is `5208efcbae`, the version is `targetDP-CUDA`, the platform is the NVIDIA Kepler K20, 
the room dimensions are `256x256x202`, the source is at `x=120, y=120, z=60` and the 
receiver is at `x=50, y=50, z=50` and the number of timesteps is `4410`. The file then 
contains the current values from the input sound wave at all points within the room. 
Another file will contain the values of the sound wave captured at the receiver point. 
Unit tests for the codes can be found in the `testing` directory at the main level. More 
about the complete directory structure and running the codes can be found in the code 
attached to the project.

## 4.3 Specific Platform, Compiler and Drivers Overview

The specific platforms the codes ran on include: Intel Xeon E5 CPU, NVIDIA Kepler 
K20 GPU, NVIDIA GTX 780 GPU, AMD R280 GPU, AMD R9 259X2 GPU and Intel 
Xeon Phi 5110P. Specification details about these platforms can be found in Table 1 
in the Appendix. The codes were originally developed to run on and were optimised 
for the NVIDIA Kepler K20 platform. Only the new OpenCL and abstractCL versions 
are capable of running across all platforms. Two NVIDIA GPUs and two AMD GPUs 
were chosen for the comparison as they have very different specifications. The Kepler 
K20 has a very high peak GFlops and relatively low (compared to the other GPUs 
tested in this project) memory bandwidth, while the GTX 780 has a high memory
bandwidth and relatively low GFlops. The AMD R9 259X2 has the highest memory bandwidth available and the AMD R280 has more similar specs to the GTX 780. AMD GPUs are built quite differently to NVIDIA ones[34][35] and the results (see Figure 5.2 for example) show that there are performance differences to be had between these brands. Previous experiments have shown that results on the Xeon Phi fall somewhere between CPU and GPU as an “accelerator”[11]. More specifics about compilers and device drivers used for each platform and language can be found in in Table 2 in the Appendix.

4.4 Comparison Analysis

Different versions of the codes were analysed using three main metrics (performance timings, memory bandwidth and computing power) and graphs were made of the results for two different room sizes using scripts. Timings indicate how long a run took, memory bandwidth is the rate at which data that can be fetched from global memory and computing power is how many operations can be performed in a given amount of time. Scripts were written to collect, trim and display resulting data from different versions of runs. Finally, different sized room simulations were run to give a comparison of the effect of domain size.

4.4.1 Metrics

The codes were compared using performance timings (time run in seconds), memory bandwidth (in GB/s) and computing power (FLOPS). Time is calculated by running the application with timing calls in place at key points in the code. The value of the memory bandwidth was calculated with the following formula:

\[
\text{Bandwidth} = \frac{((\text{dimensions of room}) \times \text{sizeof(double)} \times 1^{-9} \times 3)}{\text{total time of kernel/number of iterations}} \tag{4.1}
\]

This equation shows that the bandwidth is calculated by multiplying the size of the room by the size of a double value, then multiplying by the constant $1^{-9}$ to format the end result to gigabytes and finally multiplying by another constant indicating the number of read/writes to global memory. This value is then divided by the average time the kernel takes to run. The value of three in this formula assumes perfect caching on the part of the hardware, which is not the case. This is why memory bandwidth values from profilers and the stream benchmark return higher values than the calculated ones, as discussed further in Section 5.4.
The value of computing power was calculated in GFLOPS (Giga[ Billion] Floating Point Operations Per Second) using:

\[
\text{GFLOPS} = \frac{\left(\text{dimensions of room} \times 1^{-9} \times 13\right)}{\left(\text{total time of kernel/number of iterations}\right)}
\]  

This equation shows that Gflops is calculated by the size of the room multiplied by the number of operations (13) as well as \(1^{-9}\) to format to a billion operations, then divided by the average time the kernel takes to run. Codes are generally either limited by the peak computing power or the peak memory bandwidth of a machine and this should be kept in mind when optimising. Graphs and analysis of these values can be seen in Section 5.4.

### 4.4.2 Scripts

Scripts were written in bash and R to ease the collection and analysis of data from the various versions run. Bash scripts were written primarily to manage different versions, trim unwanted data or aggregate values into easier to read formats, while R was used for pulling disparate data together for graphing. For a particular version of the code, the executable was run ten times and the median was selected from these values.

### 4.4.3 Room Sizes

Two different sized rooms were used in the simulation runs: 256x256x202 points and 512x512x402 points. The purpose of using two room sizes is to see what kind of impact there is from increasing the domain size. These sizes do not indicate the actual size of the room, just the number of points in the grid representing the room. The physical size of the room can then vary according to the physics of the simulation.

### 4.5 Testing

Unit tests were written for ensuring correct results from the different benchmark runs and can be found in the testing directory along with comparison data. The tests compare output data from the different versions to the data from the original benchmarks, which is known to be correct to an acceptable degree. The tests compare values of doubles stored in binary data files. These tests compare these values by percentage difference (down to .000001%) as opposed to a delta, which might return an incorrect
inequality for extremely small values. The tests are called during a batch script after each run of a different version and will output an error if they don’t pass. The data directory also contains original files about timings for comparison of performance.

4.6 Code Development

Three different codes were used in this project: simple room acoustics benchmark implementations, the abstracted framework and advanced room acoustics benchmark implementations. The simple benchmark simulates the acoustics in a room with a single source and single receiver based on an original CUDA version from the NESS group[12]. Three other versions of this benchmark were developed for performance comparisons based on this original. The abstracted framework was built around this original benchmark, but bolsters it with several new classes to accommodate the abstractions required. The advanced codes drew from two different advanced benchmarks provided by the NESS group, one for investigating the effects of larger stencils and one for viscosity[9][12].

4.6.1 Simple Room Benchmarks

The original simple room acoustic benchmark was written by the NESS group in C and CUDA and subsequent implementations were written based off of this one using C or C++ and one of the following: OpenCL, OpenCL with C++ bindings and the targetDP library. The CUDA version used in this project was updated from the original version to run more easily in batch and output data in the same format as other versions. Additionally, the optimisations that were originally part of that version were retained but not duplicated in other versions. Table 4.1 shows which platforms each of these versions are capable of running on. Those versions marked in red are the updated versions of the original codes whereas all other versions are new versions. From the table it is clear why other versions were necessary for comparison, as the original version can only run on NVIDIA GPUs. All other versions offer more portability across different platforms.
4.6. Code Development

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>OpenCL</th>
<th>targetDP</th>
<th>abstractCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NVIDIA K20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NVIDIA GTX 780</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMD R280</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>AMD R9 259X2</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 4.1: Simple Benchmark Version Comparison Matrix. Red indicates benchmarks which existed already.

The crux of the main algorithm from the original benchmark using CUDA can be seen in Figure 4.2. This code snippet shows the main GPU algorithm which updates values in the room according to sound waves moving through space over time from a single source. The parameters passed into the function (grid and gridTS1) are pointers to data representing the room grid for timestep \( n \) and timestep \( n-1 \), which are calculated in a loop outside of this function. The grid point indices \((X, Y, Z)\) are mapped to the thread IDs (from the workgroup and workitem settings). Whether the points are at the halo (ie. the very edge of the room) is checked in the if statement on lines 2-3. Since the arrays are passed in as one-dimensional pointers, the “flattened” index \((cp)\) is calculated on line 4. Boundary conditions are calculated, but not shown. The six-point stencil is calculated on lines 6-7 and the final update to the original grid using this stencil and the boundary conditions is on line 8.
Chapter 4. Methodology

Room Update GPU Kernel

```c
__global__ void UpdateRoom(double *grid, double* gridTS1){
    if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1))
        && (Z>0) && (Z<(Nz-1)) ){
        int cp = Z*area+(Y*Nx+X); // flattened index
        ...
        double S = gridTS1[cp-1]+gridTS1[cp+1]+gridTS1[cp-Nx]+
                    gridTS1[cp+Nx]+gridTS1[cp-area]+gridTS1[cp+area];
        grid[cp] = cf*( (2.0-K*cf_d[0].l2)*gridTS1[cp] + cf_d[0].l2*S
                        - cf2*grid[cp] );
    }
}
```

Figure 4.2: Original simple benchmark CUDA kernel for updating points in a room. The code is called from a main loop that iterates over the number of timesteps selected. The `grid` and `gridTS1` pointers are swapped after each iteration.

Other versions of the code have similar algorithms, apart from targetDP, which required explicit vectorisation for the Xeon Phi. This can be seen in Figure 4.3, for a small section of the benchmark. VVL is the number of doubles able to be vectorised at a time (this is value is architecture-dependent, for the Xeon Phi it is eight). `targetTLP` defines the thread level parallelism, which would be the main loop that ends up being supplanted in the GPU stencil and accessed indirectly through the thread IDs. The variables are defined as arrays of size eight instead of single values, allowing for multiple values to be computed at a time. `targetILP` defines the instruction level parallelism loop, which allows the array of values to be updated at the same time where vectorisation exists on a platform. In this example, the array indices (ie. `X[vi]`) are getting updated eight at a time. On most machines this won’t make a difference, but on the Xeon Phi the compiler cannot currently always find the vectorisation correctly if it is not explicitly defined in this manner.
4.6. Code Development

```c
#define VVL 8
__targetTLP__(idx, VOLUME){
    int revisedIdx[VVL];
    int X[VVL];
    int Y[VVL];
    int Z[VVL];
    __targetILP__(vi) revisedIdx[vi] = idx + vi;
    __targetILP__(vi) X[vi] = revisedIdx[vi] % Nx;
    __targetILP__(vi) Y[vi] = (revisedIdx[vi] / Nx) % Ny;
    __targetILP__(vi) Z[vi] = (revisedIdx[vi] / (Nx*Ny));
```

Figure 4.3: Snippet of targetDP implementation showing explicit vectorisation for updating index values eight at a time using temporary arrays.

### 4.6.2 Developing abstractCL

The development of abstractCL involved abstracting away the original benchmark as well as developing a method for swapping in different data types and optimisations. The overall work on the framework kept the basic layout close to the original benchmark while allowing for behind-the-scenes tweaking for different implementations. There are a number of data layout abstractions developed already to be used with the framework and more can be added easily. Optimisations can also be swapped in and out in a similar fashion as the data layouts. An example of how to swap in a data layout is walked through for clarity.

#### 4.6.2.1 Development Overview

abstractCL was developed to create room simulation kernels on-the-fly, depending on the type of run user wants to do. The type of variations can be between different data layouts of the grid passed in to represent the room, hardware-specific optimisations or both. This is done through swapping in and out relevant files that include overloaded functions or definitions in the main algorithm itself. New classes can also be developed based on the simple benchmark for more complicated codes (like the advanced benchmarks) by extending or inheriting from the current parent template.

This framework runs similarly to the other versions, apart from that the kernel used is created before the code is run, which creates more overhead. It was developed
Chapter 4. Methodology

in C++ due its built-in functionality available for classes, templates, inheritance and strings. Additionally, OpenCL (with C++ bindings) was used because of its inherent portability. abstractCL works by concatenating strings output by functions in files chosen by macros which define what version should be run. Certain functions must always be defined as dictated by a parent template class. However, their implementations can be pulled in from another source and concatenated to the file with the abstracted kernel before being submit as a complete program to the compiler.

The code in Figure 4.4 shows the layout of the abstracted kernel. In comparison to Figure 4.2, it should be straightforward to discern where abstractions have been pulled out of the benchmark. The normal grid and previous time step grid (grid and gridTS1) are now represented by the data type grid_data. Further down in the code, there are no longer any direct accesses to this new type - they are all behind the scenes. Additionally, all functions that act on the grid or get values from it are abstracted out. isConnected(...) replaces the if statement on lines 2-3 in the original version, CalculateIndex(...) calculates the flattened index of the array, CalculateStencil(...) does the nearest neighbours calculation and UpdateGrid(...) sets the new value in the grid. These functions then get defined elsewhere and can have multiple different definitions across files that get swapped in and out according to implementation.

Abstracted Room Update Kernel

```cpp
__kernel void UpdateRoom(__global grid_data *grid,
                             __global grid_data *gridTS1)

    if(isOnGrid(X,Y,Z))
    {
        int idx = CalculateIndex(X,Y,Z);
        ...
        value sum = CalculateStencil(idx, gridTS1);
        int numNeighbours = getNumberOfNeighbours(X,Y,Z);
        UpdateGrid(idx, numNeighbours, sum, grid, gridTS1, &wallCoeffs);
    }
```

Figure 4.4: A simplified version of the abstracted kernel code showing the room update function. This is comparable to the original version shown in Figure 4.2.
4.6.2.2 Data Layouts

Table 4.5 describes the different versions of abstractions that can be implemented and compared, although others could easily be added. Seven different versions were developed for testing, though some are quite similar to each other on the surface. The intention was to see what substitutions would have the most effect, even where values may not be used. As such, a variety of layouts using different numbers of variables and layers were tried out. The main differences between the versions is through the use of structs and how data is mapped to or with them. For the first two versions compare and none, there are no structs used so the data is mapped as it would be normally for doubles, the only difference between the two being the use of typedef in none. In one_layer and full, the data type is a struct of arrays (in this case only one array), which means while the array itself inside may be aligned nicely, the struct will have padding around it so will not necessarily fit neatly into memory. For full, there is also another struct inside the main struct, so here the memory is even less neatly aligned than it would be for a built-in type as there is another struct requiring memory. Still, this is preferable to arrays of structs, as in the case of structarr. Here the padding will be added for each of the grid data points. The types two_layer and twotwo_layer both are structs of arrays of structs, the first of which has one data point in the grid point struct and the latter with two. As the layers increase, the complexity of the memory layout increases and thus will likely hinder performance.
Figure 4.5: Table of Data Abstractions. These abstractions are for representing points on a 3D grid representing a room. \texttt{grid\_data} is the type representing the room itself and \texttt{grid\_data\_value} (where applicable) is for the grid point in the room.

4.6.2.3 Example Walk-through: Data Layout Swap

As an example, the process of swapping out a different memory layout for the grid data types is walked through. The first step involves setting up separate struct and class files for the layout version and then using macros to pull out which layout to use. That is, all class names use the same name for the grid data type in the abstracted version of “\texttt{grid\_data}” (as described in Table 4.5). Then the correct classes can be brought in by including the appropriate file. The reason for having a separate struct and class file is because in the main code, the host data needs to be the same type of object as what is seen in the “on-the-fly” kernel that gets built, so the data layouts must be defined for host as well as device (GPU). This process can then be automated for a batch run comparison using scripts that edit files on the fly (ie. can swap in and out the line where the macro used defined). This can be seen in Figure 4.6, where \texttt{TWOLAYER} is the name of the data abstraction of interest. This is defined and then includes the two headers \texttt{grid\_structs\_twolayer.h} and \texttt{CLGridSetGetTwoLayer.h}. The
first of these files defines the data layouts themselves as can be seen in Figure 4.7. Here, the struct grid_data_value is defined for points on the grid and the struct grid_data is defined as the grid object. Then in the second file, the actual getters and setters for data values in the grid are defined and returned as strings (however only the getter is shown in Figure 4.8). In this example, the function GetGetString builds a string of the function to return the value of a grid point. It takes in the index of the value and the grid_data object as parameters and then returns the value at that index - except this is all encapsulated in a string as it will be combined with other files to create a mega header file. The functions themselves are called from other functions when accessing points on the grid, however this has not been shown for brevity (but can be found in the attached source code). The parent class can define what getters and setters must be defined and the strings of functions which are output get concatenated together to form one large header file that gets included with the abstracted kernel.

```c
#define TWOLAYER

#ifdef TWOLAYER
#include grid_structs_twolayer.h
#include CLGridGetSetTwoLayerabstracted.h
#endif
```

Figure 4.6: Macros for swapping in and out data layout files. The layout in use is first defined. The first file is the definition of the structs, which will always have the same names as what's in the abstracted kernel as well as the host code. The second file creates the accessor functions for the structs used, which are concatenated with other strings that are then included with the kernel.
#ifndef GRID_STRUCTS_H
#define GRID_STRUCTS_H

typedef double value;

typedef struct grid_data_value // point on grid
{
  value val;
} grid_data_value;

typedef struct grid_data // the grid grid
{
  grid_data_value data[VOLUME];
} grid_data;
#endif

Figure 4.7: grid_structs_twolayer.h file showing how data structs are declared for the abstracted version TWOLAYER

std::string GetGetString()
{
  return 'value getValueAtPointWithIndex(int idx, __global grid_data *grid)\n    {\n      return grid->data[idx].val; \n    }';
...

Figure 4.8: GetSetTwoLayer.hpp file defines functions for accessing particular data abstractions. These are returned as strings because the kernel that includes these definitions is built on the fly. Here only the getter is shown.

4.6.3 Advanced Room Benchmarks

Experimenting with advanced codes provides insight into codes that are closer to what an actual room acoustics simulation run by the NESS group would be like. The simple benchmarks updated and re-implemented in Section 4.6.1 are only able to convey a
very simplistic view of what is going on for a very limited type of room acoustics. Two new features were investigated to make these “advanced” codes: viscosity and larger (leggy) stencils. The main algorithmic difference in adding viscosity is that another grid is passed into the main kernel and more computations are performed. Leggy stencils involve a larger number of neighbours to calculate from in a given direction. For this situation, only 18-point leggy stencils were used (three points in each of the six directions). Analysis about differences in accuracy of the simulation from the use of these versions is not included in this project.

Three separate versions of these additions were used in this investigation of advanced benchmarks: one using viscosity, one using leggy stencils and one with a combination of both features. Both the viscosity and “leggy” versions were based on original NESS codes, the combination one was developed. Unlike the simple benchmarks, only two versions of these codes were investigated: CUDA and OpenCL (with C++ bindings). The CUDA versions were largely untouched from their original state, however boundary conditions were taken out of the viscosity and original versions to be more comparable with the leggy version, which had no boundary conditions to begin with.\footnote{Removing boundary conditions from the original codes only made a marginal difference ($\sim 3\%$) in performance} Abstractions for the advanced codes were not investigated, but the abstraction framework was developed to be extendable with this future development in mind.

The advanced codes do not differ greatly from the simplistic ones, but each has key algorithmic changes to them. For the viscosity benchmarks, the main algorithmic difference is that three arrays are passed in instead of two (as there is an extra timestep used for the grids - two timesteps back instead of just one) and there are two stencil computations for the two different previous timestep grids. This can be seen in Figure 4.9. On lines 1-2 the new arrays are passed in, on lines 4-5 there are now two stencils where there previously was one and finally the update of the grid point on line 7 is slightly more computationally intensive than it was previously.
Vicosity Implementation

```c
__global__ void UpdateRoom(double *grid, double *gridTS1,
    double *gridTS2)

    double N1 = gridTS1[idx-1]+gridTS1[idx+1]+
        gridTS1[idx-Nx]+gridTS1[idx+Nx]+gridTS1[idx-AREA]+
        gridTS1[idx+AREA];
    double N2 = gridTS2[idx-1]+gridTS2[idx+1]+
        gridTS2[idx-Nx]+gridTS2[idx+Nx]+gridTS2[idx-AREA]+
        gridTS2[idx+AREA];

    grid[idx] = (2.0-K*l2-K*lvah)*gridTS1[idx] + (K*lvah
        -1.0)*gridTS2[idx] + (l2+lvah)*N1 - lvah*N2;
```

Figure 4.9: CUDA version of key components of the advanced code with the algorithm updated to account for viscosity.

The leggy code has three additional neighbouring points accesses in each stencil direction, which means a lot more memory accesses and computation, though the number of grids remains the same. This can be seen in Figure 4.10. On lines 3-12 the stencil is much more complicated than before not only through number of memory accesses, but also with coefficients that fall out from using finite difference methods. A version was also made including both of these amendments, however correct results were unable to be obtained. The version is however still included in the experiments with this caveat.
4.7 Memory Optimisation Work

There are two main types of memory optimisations that are often investigated when looking to improve performance on GPUs: image and local memory. Image memory is a read-only or write-only memory that is distinct from global memory and uses separate caching allowing for quicker fetching for specific types of data (in particular, ones that take advantage of locality). Local memory is specific to a compute unit so only can be shared between threads in a workgroup, but can be useful for data re-use within the workgroup. Whether or not these types of memory optimisations show any improvement is very algorithm dependent.

4.7.1 Image Memory Development

Developing a working version that utilised image memory in the abstract framework was investigated first. However, NVIDIA GPUs are restricted to using only version 1.2 of OpenCL, meaning that abstractCL must be limited to working with this version for portability considerations. One big hindrance of this version is that it does not support...
double precision in image memory, however a bespoke data type could be developed that would “fake” double precision. The simple acoustics benchmark uses two grids: one that is read-write and one that is read-only and these grids get swapped after every iteration. Thus, in its current state, only the read-only grid could take advantage of image memory. However this would be again be difficult in OpenCL given that a new type of object is created for each of the grids they would need to be swapped around after every iteration.

On the other hand, image memory is relatively easy to implement in CUDA: the modifiers `const` and `restrict` are simply added to the array parameters that will use image memory. This is also currently restricted to read-only memory on NVIDIA devices, however it was tested out to see what kind of performance increase was possible. Preliminary tests showed that there would not be a great enough improvement to warrant an in depth development of a version in OpenCL. It was also attempted to split the read-write array in the original code to be two arrays: one read-only and one write-only, resulting in three arrays: two read-only and one write-only. Using this approach would make it easier to swap the arrays in OpenCL. However, this proved to cause worse performance than the original setup, so no further development was done.

### 4.7.2 Local Memory Development

Previous work with these room acoustics benchmarks showed that there was potential for speed-up through the use of local (or shared) memory, but results were mixed[12]. Local memory optimisations were implemented in CUDA in combination with image memory (results without image memory were not included) and tested out. In this study, three different algorithms were tried:

- implementation with local memory for array accesses, except for global accesses for boundaries
- implementation using only local memory for all array accesses
- tiling across the XY-planes in the Z direction and utilising local memory for the tiles with global accesses for the boundaries

More information about how these algorithms were designed can be found in [12].

First these algorithms were re-implemented in CUDA and then also implemented in OpenCL with C++ bindings and tested out. Their thread settings were then optimised, the result of which none of these algorithms showed significant speed-up apart
from the XY-tiling one. Only local memory was tested out on AMD GPUs, however on NVIDIA, the combination of local and image memory was also looked at. The experimental results will only discuss the versions developed using the XY-tiling algorithm.
Chapter 5

Experimental Results

Experiments were run in four areas: data layouts, simple benchmark comparison across platforms, optimisations, and finally advanced codes investigations. Data layout comparisons were done to optimise the grid data type for the abstract framework. The simple room acoustics benchmark comparisons were performed to determine the performance, portability and productivity of various implementations of the same room acoustics code. Optimisations were investigated to test out the use of the abstract framework as well as pushing the benchmark to find its optimal settings on GPUs. The advanced codes serve as a companion comparison to the simpler codes in showing whether the performance patterns are retained when key components of the algorithm change.

5.1 Data Layouts

The memory layout of an object can have a significant effect on performance. Preliminary testing of the abstractCL version showed that its initial implementation was much slower than its counterparts. Upon investigation, it was clear that the reason behind this was due to the choice of the data layout for the grid data type being used. The framework was then first optimised for the most performant (and productive) layout to use before being compared with additional versions. The following results show this investigation into the effects of different data layouts, where even slight changes in how the data type is setup can affect performance. This occurs when memory accesses aren’t coalesced or aligned coherently, limiting the number of data values that can be accessed at the same time.

Seven different layouts were tested out on three different platforms: AMD R9
Chapter 5. Experimental Results

259X2, NVIDIA K20 and Xeon Phi. Code descriptions of each of these layouts can be found in Table 4.5. The performance varied more on certain platforms than others, but some layouts were consistently worse than others. Timing performance results of the different versions across platforms can be seen in Figure 5.1. The difference in colour indicates the part of the code that is running: the room updates kernel is in light blue (and takes up the majority of the computation time) and dark blue represents everything else, such as IO, setup, etc. The values across the horizontal axis indicate which memory layout is being used in the code and timing is shown on the vertical axis in seconds. The graph is split into segments indicating which platform the codes were run on at the top of each segment.

![Figure 5.1: Comparison of performance of different data structures across different platforms for a room size of 256x265x202. A guide to the different layout versions can be found in Table 4.5.](image)

From the graph it is clear that the `twotwolayer` layout is the least performant version, being more than twice as slow on two of the architectures as the comparison version (using only an array of doubles for the grid type). This is followed by the `full`
version which is also noticeably slower on all architectures. The \textit{twotwolayer} layout involves a struct which has an array of structs with two data points inside it. Given that the second point is not currently used, this means that half the memory that would normally be contiguous is wasted. With the extra layer of structs involved, there is additional padding required. This explains why the performance is so much worse for this version. For the \textit{full} version, a similar structure is used except there is only one data point in the grid points struct, but there is an additional struct for boundary conditions still being included in the grid data type and accessed in the code. Thus, while not more than half of memory is going wasted, there is still extra padding for the simpler struct across the whole array of points for the grid. Additionally, the addition of another struct inside the struct means that there is more padding than necessary in the main struct as well. On the Xeon Phi, the \textit{structarr} and \textit{two\_layer} versions were also significantly slower than other versions. \textit{structarr} uses an array of structs, whereas \textit{two\_layer} uses a struct of an array of structs. On the other platforms, these versions compare similarly to the versions with little to no struct usage, so perhaps it is a compiler dependent issue on the Xeon Phi causing the slowness. Also on the Xeon Phi, much more “other time” is being spent than on other platforms. The percentage of time being spent on “other” on the K20 is still larger than it is on the AMD R9 259X2, but not significantly so. In conclusion, it seems possible to use abstractions for memory layouts that make an acceptable amount of performance hit, but caution must be used in selecting which as different compilers can optimise abstractions differently.

5.2 Simple Benchmark Comparison

The simple benchmark versions were analysed and run across the six platforms in order to collect data for comparison about performance, portability and productivity. For performance, several metrics were captured, but timing will be the only one discussed here. For portability, the codes were analysed according to which platforms they could access. Productivity could be measured by looking at lines of code as well as the more subjective measure of how straightforward it is to program different versions.

5.2.1 Performance

The four main implementations of the simple acoustics benchmark were run at two different room sizes over six different platforms collecting data about timing perfor-
formance, memory bandwidth and number of floating point operations. The four implementations include CUDA, OpenCL, targetDP and abstractCL and these are the names used subsequently to discuss each. The different platforms include a CPU (Intel Xeon), two NVIDIA GPUs (Kepler K20 and GTX 780), two AMD GPUs (4470K and R280) and the Xeon Phi. For the GPUs, the number of work items per workgroup settings was set at 34x4x2. On the Xeon Phi 236 OMP threads and a vector array size of 8 were used. On the CPU, the number of OMP threads was limited to the number of cores: 32. The main kernel iterates in a loop over the number of timesteps chosen by the user. This was restricted to 4410 for this project, however normally this value is much higher. Ultimately most of the versions ran quite similarly on the same hardware, but this is useful as it shows that it should be straightforward to implement a performance portable version. The larger room sizes showed similar but more consistent performance across the platforms and the subsequent discussion of results will focus on these results over the smaller room sizes.

There are two main comparisons that can be made between the various versions run on different platforms. The first is to analyse how performance of one particular version compares with each of the others on the same platform. The second is to examine how performance of a particular version compares across different platforms. From these two perspectives, there are other conclusions which can be drawn combining information from both sides. Discussion on why performance is different for the latter (across different platforms) is discussed in more detail in Section 5.4.

The graphs in Figure 5.2 show performance timings of the two room sizes. The various colours indicate how much time was spent in which part of the code: red for the time in the room update kernel, orange for the time spent in aggregating values at the receiver point kernel, green for the time spent copying data around and blue for all other miscellaneous time. The graph is split up into six sections, one for each of the platforms run on, which is indicated at the top of each section of the graph. Then, the bars in each section show the performance of the version of the code that has produced the result. Subsequent graphs for memory bandwidth and GFlops will follow a similar layout (as will those for advanced codes).
5.2. Simple Benchmark Comparison

![Benchmark Comparison Diagram]

Figure 5.2: Performance timings for simple room acoustics benchmark implementations across various platforms using room sizes of 256x256x202 (top) and 512x512x404 (bottom).
One of the main purposes of this research was to develop a performant portable and more productive version of the room acoustics benchmark, able to run comparably across all the platforms selected and results show that this has been achieved. When isolating the main kernel, the abstractCL version is faster than or within 10% of the performance of all other versions, whereas when accounting for total time this difference drops to a 23% difference in performance time at worst. However, this difference is on the smaller room size and given that the room simulations were not run for as many timesteps as they would normally be (where the overhead of starting up versions would decrease in ratio over time), this value is less significant than for that of the larger rooms which take longer to run. The largest difference between abstractCL and other versions for the larger room size is on the NVIDIA GTX 780 at around 7% and is even faster than other versions in some instances. Values for all of the performance differences between the abstractCL and the best version run on a platform can be found in Table 5.1 for the smaller room size and Table 5.2 for the larger one.

Comparisons between abstractCL and the native versions (ie. OpenCL and CUDA) individually provide some insight into its performance capabilities and portability. In comparison to just the OpenCL version, there is at most 2.5% difference in comparison to the abstractCL version and occasionally the main kernel in the abstractCL version runs slightly faster. One reason why this may happen is that abstractCL is written in C++ and uses the C++ bindings of OpenCL. In comparison to the CUDA versions, the abstractCL version is slower on NVIDIA GPUs (also true for the OpenCL version), but on the AMD R9 295X2 was able to obtain a 40% increase in performance in comparison to the original benchmark run on the platform it was optimised for originally (the NVIDIA Kepler K20). This is a sizeable difference in the timings, especially given that these versions of the codes were not specifically tuned on any of the other architectures as they were for the original.
Overall the performance for other versions also does not differ greatly per version per platform. Generally, OpenCL has slightly more overhead and does take longer on NVIDIA platforms than the native CUDA versions, though this is less pronounced at larger rooms and on the Kepler K20. For the smaller room, the difference in performance for the main kernel is 9.3% between the OpenCL and CUDA versions on the GTX 780, whereas this drops to 4.5% for the larger room size. The difference between these two runs on the Kepler K20 is less than 1% on both platforms for both room sizes, decreasing only marginally for the larger room. However, these differences go up to 16% on the GTX 780 for the smaller room version when the total performance time is taken into account. The targetDP-CUDA version is markedly slower on both NVIDIA platforms, although less slow on the GTX 780 than the Kepler K20 (9.8% for small rooms versus 16.7% for larger rooms). This library uses slightly different workgroup and workitem settings, which may account for this difference in performance.

The results for other platforms were poorer than for the GPUs, but this is what was expected. The CPU run was much slower, but of interest for comparison to see how much of a performance gain the accelerators can offer. For small rooms, abstractCL was 6.2x faster on the fastest GPU (AMD R9 295X2) than the CPU and for the large
rooms this decreased to 5.7x faster. Performance on the Xeon Phi was in between the CPU and GPU, which is similar to performance reported elsewhere[11]. The abstractCL version was about 1.8x faster on the fastest GPU in comparison to the Xeon Phi for small rooms and 3.8x faster for large rooms. Interestingly, for smaller rooms the targetDP-C version is the fastest on the Xeon Phi, whereas the abstractCL version takes over for larger ones. However, for the rest of the results, the ordering of fastest to slowest remains consistent with other platforms. Timings for the Xeon Phi were much more varied generally, so it’s difficult to draw too many conclusions from its results.

5.2.2 Portability

Although maintaining performance portability for versions is of key interest, portability on its own is also an important measure. From the graphs in Figure 5.2, it is easy to discern that the targetDP, OpenCL and abstractCL versions are among the most portable. The OpenCL and abstractCL versions technically can also run on the CPU, but were not included in this investigation. However, a code should be portable without requiring rewriting. This was not the case for the targetDP-C version on the Xeon Phi which needed hand-tuned vectorisation to get good performance. However, it only needed be rewritten once and now the vector size can be tweaked for other architectures with vectorised instruction units.

5.2.3 Productivity

For an objective analysis of productivity, LOC (or lines of code) can be used as an approximate measure. The Table 5.3 shows this value for each of the versions. The values are comparable with what is expected, in that OpenCL has the most lines of code and one of the more productive versions (targetDP) has the least. As the abstractCL version is still quite tied the OpenCL, it is not surprising that it only comes in third.
### 5.3 Optimisations

Two GPU optimisations were investigated with two versions of the simple benchmark: thread configuration values and local memory. Thread configuration involved optimising the workgroup and workitem settings for enabling an optimal thread layout for data throughput and memory access speed. Local memory was then used in conjunction with a tiling algorithm to push the limits of the benchmark. Both of these optimisations resulted in more performant versions of the benchmark.

<table>
<thead>
<tr>
<th>Version</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>273</td>
</tr>
<tr>
<td>OpenCL</td>
<td>421</td>
</tr>
<tr>
<td>targetDP</td>
<td>261</td>
</tr>
<tr>
<td>abstractCL</td>
<td>305</td>
</tr>
</tbody>
</table>

Table 5.3: Productivity Table For Different Implementations. *Lines of Code* indicates how many lines there are only in the main loop and stencil kernel (ie. not including any library code)

From a subjective point of view, none of these codes in their current form are particularly productive. In the sense that they can easily run different versions, then targetDP and the abstractCL versions are semi-productive. CUDA in comparison to OpenCL is productive, but this is not particularly telling. OpenCL is easily the worst in productivity of all the versions. Apart from the large amounts of boilerplate code, it is also tedious to put things like simple timing calls in. The C++ bindings for OpenCL do improve programmability somewhat, however. Once targetDP was setup, it was relatively straightforward to run different versions, but the syntax is still quite low-level (very close to C). It is also relatively easy to swap between abstractCL versions, but again the syntax is also still low-level. The framework is limited in its utility and extending it is not currently straightforward.
5.3.1 Thread Configuration Settings

Optimal thread configuration was determined for the two versions CUDA and abstractCL. This was investigated in order to maximise the amount of data being processed at a time as well as assigning the appropriate amount of workitems in a workgroup in a given direction to maximise memory access coalescing. A script was written to iterate over the settings defining the number of workitems assigned to a workgroup - or a “block size” - for values of 2-64 iterating by powers of two in the X and Y directions (workgroup settings can be one to three dimensional - in this project they are three dimensional, but for the room sizes used the Z direction is not a power of two and is held at 2). This was done for two versions to see if there was any difference between the two. The optimal workgroup settings for CUDA and abstractCL versions were found to be slightly different, however the three fastest configurations were the same across both versions for both room sizes.

There are two types of graphs which show the dependence of the timing on the block size along the X and Y dimensions respectively, each run for both small and large room sizes. In the entry of the first graph, the values of the X dimension are held constant at X=2, while the values of Y are iterated between 2-64. This is then repeated for X=4, X=8, all the way up to X=64 for subsequent entries in the graph. The shapes along the vertical axis that is shown for each of these values on the horizontal axis indicates the amount of variation in timings there were for different values of Y (from 2-64). In the case of holding the X value steady and changing the Y, this indicates how much dependence the X value has on the choice of the Y value. These graphs are shown in Figure 5.4 for small (256x256x202) and large (512x512x404) room sizes. It is clear from these graphs that there is some dependence on the choice of Y value for small values of X, but as the size of X increases, the range converges on much smaller range of performance timings so the dependence on Y goes down. For brevity, only the abstractCL version of this experiment is shown, however behaviour of the CUDA experiments showed were shown to be similar.

The opposite was then done for the X and Y values, where values of Y were held constant along the horizontal axis while the variation of X is graphed along the vertical axis. In the graphs in Figure 5.3 these can be seen to vary much more significantly than the ones in Figure 5.4. It still holds that the smaller the value of Y is the larger the variation there is in performance timings for different X values. However, as the value of Y goes up, this variation only decreases a small amount. From these graphs we can
determine that the X dimension has a much greater effect on performance timings and indeed that performance timings are better for large values of X (where the Y values do not matter as much). This makes sense given that memory is aligned in the X direction in main memory. Additionally, the timing ranges are larger for larger room sizes, but the overall shapes are nearly identical for the two rooms so this trend is also not dependent on the problem size.
Figure 5.3: abstractCL block size dependence for the Y direction for room sizes 256x256x202 (top) and 512x512x404 (bottom). The plots are violin plots, which show a combination of a box plot and a kernel density plot. The shape indicates what the distribution is (mirrored on each side). The white dot indicates the median and the black line indicates what would be the whiskers in a box plot (variability).
Figure 5.4: abstractCL block size dependence for the X direction for room sizes 256x256x202 (top) and 512x512x404 (bottom). An explanation of how these plots can be interpreted can be found in Figure 5.3.
5.3.2 Local Memory Experimentation

Local memory optimisations were explored using an XY-tiling method that iterates over the Z dimension of the room. This means that the Z index of a grid point is held constant while X and Y indices are calculated for a tile spanning the XY plane. For subsequent iterations, the Z index is incremented and the next tile is updated. This algorithm was adopted from the same method proposed by Craig Webb in his PhD thesis[12]. The XY-tiling method was run using optimal thread configurations per framework and platform in CUDA, OpenCL and abstractCL for small and large rooms across three different platforms (two NVIDIA GPUs and one AMD GPU). As the NVIDIA GTX 780 and AMD R280 gave similar results in Section 5.2, the codes were only run on the NVIDIA GTX 780. The CUDA version was also run using image memory and all results were compared with codes not using local memory. Only GPUs are used as they are the only platforms with these types of memory.

Figure 5.5 shows the results for this experiment, where the optimised versions are the fastest versions found in this project per version per platform. Three different types of codes were run: sharedtex uses local and image memory (only the CUDA version), shared uses only local memory and none uses no memory optimisations (only thread configuration optimisation). The bar plots for these versions are coloured in yellow, green and blue respectively. The reason for comparing to an optimised thread configuration version instead of the original run was to isolate what effect the memory optimisations had. As before, different platforms are indicated by separate segments of the graphs. The results shown are only intended get a rough idea of capable improvement and as such have omissions.
5.3. Optimisations

Figure 5.5: Local memory optimisations for CUDA, OpenCL and abstractCL versions of the room acoustics benchmarks run for room sizes of 256x256x202 (top) and 512x512x404 (bottom).
All versions showed some improvement with this use of local memory, but this amount varied per version and per room size across the different architectures. The simplest case is the abstractCL version which showed the most consistent improvement: 15% for small and large rooms on the AMD R9 259X2 chip and 4% for the small and large rooms on the NVIDIA Kepler K20. For the OpenCL version, the local memory optimisation improved performance by $\sim 9\%$ for the small room and $\sim 15\%$ for the large room on the AMD R9 259X2. On the NVIDIA K20, this improvement was $\sim 6\%$ for the small room and $\sim 12\%$ for the large room. For the CUDA version on the K20, the performance increase was $\sim 7\%$ for the small room and $\sim 9\%$ for the large room using image and local memory. The performance increase is about half those values when using only local memory. On the NVIDIA GTX 780, the CUDA version went $\sim 6.7\%$ faster for the small room and $\sim 8\%$ faster for the large one using both memories.

Overall, it is clear that the OpenCL version was able to see more improvement across the platforms using local memory, especially as it was not using image memory as well. However, the abstractCL version was on par with this version on the AMD chip, where both showed around 15% improvement. The improvement differences between these two versions level out somewhat when compared with original timing values (ie. runs in Section 5.2) for the simple benchmark runs that were not using tailored thread configuration optimisations. Presumably, with more time to optimise the abstracted version, the differences between it and the OpenCL version would converge as well. Finally, this optimised abstractCL version on the AMD R9 259X2 now sees a performance improvement of 45% (as opposed to 40% unoptimised) when compared with the original CUDA benchmark run on the Kepler K20. This increase is more like 50% for the OpenCL version, showing that the abstractCL framework has room for optimisation. One of the reasons that these results do not show more improvement using local memory optimisations is because the codes are already not too far off peak bandwidth as is discussed further in the following section, 5.4. Another reason might be because there is not much locality to exploit in a six point stencil.

### 5.4 Bandwidth and Computing Power Evaluation

As seen in Section 5.2, the platform a code is run on can make a big difference to performance. Two of the biggest factors for why this can be are: memory bandwidth and computing power. It is often the case that codes are limited by one of these two
factors in obtaining more performance. Memory bandwidth determines how fast data can be brought in from main memory. Computing power determines the number of operations a chip can perform over a period of time, usually described with FLOPS.

The top graph in Figure 5.6 shows the memory bandwidth values of the larger room sizes (smaller room results not shown because the profiles are similar). The blue columns represent the memory bandwidth as calculated by Equation 4.1. The columns are again split according to platform and the different bars indicate code version. There are also three horizontal lines shown on the graph: one for the peak bandwidth in black (exact values for these can be found in Table 1 in the Appendix), one for the profiled value (if applicable) in green and another for the best value obtained by the stream bandwidth. It would be inconceivable for a code to ever actually reach the peak memory bandwidth advertised, which is why the STREAM benchmark was also run on all the chosen architectures for reference[33]. Comparable (or surpassing) values to the STREAM benchmark would indicate that the codes were making optimal memory accesses. This only seems to occur on the Kepler K20. On the NVIDIA GTX 780, the profiled version is significantly slower than the stream benchmark at \( \sim 13\% \) and on the AMD R280 it is only behind by \( \sim 7\% \). However, even with these differences, all versions are only \( \sim 30\% \) slower than the peak bandwidth.
Figure 5.6: Bandwidth (top) and Gflops (bottom) for different versions of the simple room acoustics benchmark across platforms for room size 512x512x404. Smaller room results are not included as they show similar pattern.
While the cross-platform results were mostly as expected, one of the most striking differences when running the versions on all the platforms is that the OpenCL and abstractCL versions run nearly twice as fast on the AMD R9 259X2 as on the NVIDIA Kepler K20. The memory bandwidth is nearly twice as high on the AMD R9 259X2, which explains part of it (320 GB/s versus 208 GB/s). However, the OpenCL versions run on AMD R280 around 10% faster than they do on NVIDIA GTX 780 even though their advertised peak bandwidth is roughly the same (~288 GB/s).

Overall, profiled memory bandwidth values were much higher than calculated ones and closer to the STREAM benchmark. Kernel run time did not change noticeably between the 1st and 4000th iteration on either platform, which could effect the accuracy of the calculated values. So presumably it is only the limitation of the number of memory accesses in the bandwidth equation used (see Equation 4.1) which does not reflect what is really going on in the hardware through caching. Codes were profiled with varying room sizes as well as iterations to give more accurate values of the memory bandwidths. Interestingly, the number of iterations and the size of room increases the bandwidth on AMD GPUs, but not on NVIDIA ones.

The local memory optimised versions of the code were also profiled and these results provide a few reasons why the performance did not improved for codes when using local memory. Looking at the cache hits on the profiler, it appears that actually the cache is already doing a fair job. There are only 30% cache misses on original code and this drops to only 20% on the local memory code. There is also a difference in the number of global reads, but on average it works out to be only ~9% fewer on the local memory optimised versions. Larger stencils are more likely to get large performance gains using local memory, here only 6 points would be potentially benefiting from reuse.

Figure 5.6 (bottom) also shows the GFlops versus room code version for the larger room size. The green columns represent the GFlops as calculated by Equation 4.2. The black line indicates where the peak GFlops could be obtained in theory. One of the first things that is noticeable is that no version comes within half the peak computing capability. Comparing this with the memory bandwidth graphs in the same figure, values can come much closer to the peak, making it clear that these codes are restricted more by memory accesses than raw computation.
5.5 Advanced Codes Comparison

The goal of running the advanced versions of the room acoustics benchmarks was to determine if the same performance behaviour was exhibited between codes and across different platforms (and additionally between sizes of rooms) when the main algorithm changes. The comparison was done on an intentionally on a smaller scale, however, intending only to give a general idea of whether or not the codes performed similarly. Two changes were made to the benchmarks including adding viscosity and the use of a leggy stencil in place of the original six point stencil and this was done for a CUDA and an OpenCL version. These new additions were also run together to see if the combination of the two had any additional effects. All versions were run on the two NVIDIA GPUs, the two AMD GPUs and the Xeon Phi for both small and large room sizes. Results of the performance of these advanced codes can be discussed in a number of ways including: in comparison to the simpler codes, as a comparison among different advanced versions (ie. using viscosity, using leggy stencils), as a comparison between versions on the same platform and also comparisons of the same version across platforms. The abstractCL version was not extended for any of the advanced codes in this project.

Graphs in Figure 5.7 show the performance timings and the memory bandwidth of the advanced codes for the various versions for the larger room size. In these graphs, the following versions are included: cuda (the original version), cuda先进的 (cuda with viscosity), cuda_leggy (cuda with leggy stencils) and cuda_leggyadv (cuda with leggy stencils and viscosity). The comparable OpenCL counterparts of these versions were also run. These graphs are setup in a similar way as those found in Figure 5.2. However, here there are more types of the same framework and fewer platforms were tested (CPU was not included).
### 5.5. Advanced Codes Comparison

<table>
<thead>
<tr>
<th>Code Version [Platform]</th>
<th>AMD R280</th>
<th>AMD R9 259X2</th>
<th>NVIDIA GTX780</th>
<th>NVIDIA K20</th>
<th>Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Copy Total</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kernel1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kernel2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TimeLeft</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bandwidth Limits:** — peak — stream

**Calculated Bandwidth:**

<table>
<thead>
<tr>
<th>Code Version [Platform]</th>
<th>AMD R280</th>
<th>AMD R9 259X2</th>
<th>NVIDIA GTX780</th>
<th>NVIDIA K20</th>
<th>Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (GB/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
</tbody>
</table>

Figure 5.7: Advanced codes performance timings (top) and bandwidth (bottom) for various different algorithms using the same framework for room size 512x512x404. Smaller room results are not included as they show a similar pattern.
Generally the performance of the advanced codes looks similar to what can be seen for the original codes in Section 5.2. That is to say, that the codes still run fastest on AMD R9 259X2 and slowest on the Xeon Phi (probably the CPU would be slower, but these runs were not included). The versions on the AMD R9 259X2 in comparison to the versions on the K20 hover between being 43%-54% faster. For both large and small rooms, the leggy codes are slower than the viscosity codes for both OpenCL and CUDA on everything except the K20 where the leggy codes are faster. The combination advanced codes (*_leggyadv) are significantly slower across the board, particularly on the Xeon Phi.

One of the big differences in this analysis is that for the advanced codes is that it is now possible to see what effect algorithmic changes in the same version have on the same platform. It turns out that how the performance changes with changes in the rooms can vary quite a bit across platforms, which echoes the results seen when comparing local memory optimisations. When comparing original versions to the leggy, viscosity or combination versions, OpenCL codes are 1.4-6.4x slower for the combination versions. When this is limited to AMD GPUs, the difference is only 1.4x slower - for NVIDIA GPUs, 3-3.6x slower. In comparison, the CUDA version on the NVIDIA GPUs varies from 1.6-2.4x slower for the combination version. For the stand-alone leggy and viscosity versions, this difference is much less pronounced, however the same trend remains: OpenCL versions retain better performance with changes on AMD platforms and significantly worse than CUDA versions on NVIDIA GPUs. These differences cannot wholly be attributed to specification differences given that the difference exists for all these versions between the AMD R280 and NVIDIA GTX 780, which share some similar specifications. This can be seen most clearly in the memory bandwidth graph (Figure 5.7 bottom) comparing the OpenCL versions on the NVIDIA GTX 780 to the AMD R280. Both platforms have roughly the same peak bandwidth, but the more advanced versions do not get as close to reaching this on NVIDIA as they do on the AMD.
Chapter 6

Conclusions

This report concludes with an overview of some of the main results found during this investigation, analysis of improvements and failures of the project, as well as a look at where this work can go next. A new framework was developed for a simple room acoustics benchmark which showed comparable or improved performance, portability and productivity in comparison to native versions across platforms. However, a few noticeable improvements could be made and a small number of questions remain unanswered. For future work, this project has provided the foundations to continue investigating how to develop abstractions that will accommodate HPC grid-based physical simulations. Long term these abstractions could form a layer above an existing framework capable of tuning codes specific to architectures.

6.1 Summary

Results showed that a performant, portable and more productive version of a simple room acoustics simulation was possible and that this room acoustics simulation is now capable of being run with good performance on platforms that were previously inaccessible to it. An abstracted framework developed (abstractCL) was able to get within 23% of the most optimal versions of the same benchmark across platforms for small rooms and within 7% for large ones. This difference shrinks to less than 10% difference at worst (ie. for small rooms) when only the main room simulation algorithm is compared. In comparison to the original benchmark optimised on the Kepler K20, abstractCL was able to achieve a ∼40% speedup when run on the AMD R9 259X2. This increases to 45% when the benchmark is optimised for the AMD R9 259X2 platform. The best versions of the GPU codes could also get within about 30% of the peak
bandwidth advertised on the platforms selected for this project, which means not much more improvement can be squeezed out of those GPUs.

There were a number of other smaller results that came out of the investigations undertaken for this project. Inter-platform performance of the same version had a larger variability than intra-platform performance between versions of the code. The largest difference in this case was between the CPU and the AMD R9 259X2, where the AMD versions were 6.8x faster for the small rooms. For intra-platform comparison, the largest difference was 25% between the targetDP-C and OpenCL versions on the Xeon Phi for the small rooms. Additionally, it can be seen that these codes are memory bandwidth bound given that they are much closer to reaching the peak memory bandwidth of platforms than they are to reaching compute power peaks. In addition, the choice of memory layout for the grid data types was shown to have a significant effect on performance, however it was possible to create flexible abstractions that did not greatly affect performance. Advanced codes followed similar performance behaviours to the simple ones, where runs on the AMD R9 259X2 were able to get the highest performance and the Xeon Phi results were slowest. It was also shown that performance was hit the least on AMD platforms when codes were updated with new algorithms.

6.2 Critical Analysis

Given the limited amount of time for development, there are a number of things that could be improved in this work as well as issues that remain unsolved. Major improvements will continue to be investigated as the project progresses and evolves. However, some minor oversights were noted and are included here but will not be dealt with further. Remaining mysteries are also relatively minor and will just be documented here.

6.2.1 Project Improvement

The main improvement for this project would be a more flexible version of the abstractCL framework. As this one was written bottom up, it is very heavily tied to a specific benchmark and will have to be extended any time a new algorithm or major change is introduced. This might not be a large problem for room acoustics codes, which will be relatively similar, but for different simulations might require more ad-
6.2. Critical Analysis

advanced abstractions. Ultimately there is a tension between writing a framework that is specific and easy to use versus being flexible and abstract and this framework falls into the former category. The framework could also be optimised more. It runs comparably to the OpenCL counterparts in the main kernel, but there is a fair amount of overhead in starting it up.

Some other smaller issues in this project could also be improved upon. While unit tests were written, no regression tests were included which would have helped with optimisation comparisons. There were also greater performance differences between both the simple and advanced room acoustics codes than there might have been if workgroup and workitem sizes were tuned specifically per platform. This discrepancy was noticed too late during the analysis phase of the local memory optimisations, which did use platform and framework specific tunings. Additionally, for the data layout investigation, it would have helped analysis to look at the assembler to see what different compilers had actually done. Another oversight is that the OpenCL and abstractCL framework should have been run on the CPU. It would also have made explanations clearer if more research had been done on what makes the AMD and NVIDIA GPU platforms different, in order to explain some of the performance differences. Given more time, doing more analysis on the Xeon Phi performance would also have contributed more to the analysis part of the project. Finally, it was also intended to implement at least one version in one of the higher level frameworks Kokkos or SYCL, but time ran out. However, this could be done in future work.

6.2.2 Outstanding Issues

There are a few mysteries left unsolved from this project. One issue is why the targetDP-CUDA version is slower than just the CUDA version on NVIDIA platforms. It is possible that this is brought about because targetDP defines workgroup and workitem settings only in the X direction (the array is flattened out) whereas the CUDA version preserves the three dimensions and also uses three dimensional workgroup and workitem sizes. As well as this, it is unclear why the leggy stencil codes running faster than the advanced viscosity codes only on the NVIDIA K20 GPU (consistent across multiple runs and both room sizes). Additionally, it was occasionally difficult to get consistent timings on the Xeon Phi and it was not clear why this was the case.
6.3 Future Work

Future work will investigate further the use of abstractions and optimisations for the advanced room acoustics codes as well as other similar HPC physical simulations. As a starting point, the project could evolve to include more simple, flexible abstractions which can cover both simple and advanced codes. Additionally, all of this work has only been done on one chip or GPU and it would be useful to investigate how adding multiple chips would affect performance as many simulations have larger domain sizes than can be accommodated by a single chip or GPU. Further down the line, for more complicated rooms, meshing may need to be used. It is not clear if any current abstractions could manage this and if they do how well they would accommodate HPC codes.

The long term goal would be to develop or build upon a decoupled framework which pulls together the best stencil appropriate DSLs or abstraction layers for HPC codes to compile them down to hardware specific optimised code. There are a number of DSLs already in existence - some that even focus on stencils - that would be a good starting point to investigate[29][28]. The next aim would be evaluating their capability of implementing HPC codes using them, including possibly extending them. At a later stage, this abstraction layer or DSL could be linked to another framework that does the optimisation and hardware specific tuning.
Appendix

Tables

<table>
<thead>
<tr>
<th>Platform</th>
<th>Number of Cores/Stream Processor</th>
<th>Peak Bandwidth (GB/s)</th>
<th>Peak GFlops (Double Precision)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA K20</td>
<td>2496</td>
<td>208</td>
<td>1175</td>
<td>5120</td>
</tr>
<tr>
<td>NVIDIA GTX 780</td>
<td>2304</td>
<td>288.4</td>
<td>165.7</td>
<td>3072</td>
</tr>
<tr>
<td>AMD R280</td>
<td>2048</td>
<td>288</td>
<td>870</td>
<td>3072</td>
</tr>
<tr>
<td>AMD R9 259X2</td>
<td>2816</td>
<td>320</td>
<td>716.67</td>
<td>4096</td>
</tr>
<tr>
<td>Xeon Phi 5110P</td>
<td>60</td>
<td>320</td>
<td>1011</td>
<td>8000</td>
</tr>
<tr>
<td>Intel Xeon E5-2670</td>
<td>32</td>
<td>51.2</td>
<td>166.4</td>
<td>126000</td>
</tr>
</tbody>
</table>

Table 1: Specification Table. This table shows the specifications of the various platforms used in this project, including: number of cores, peak memory bandwidth, computing power (ie. FLOPS) and memory.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Platform</th>
<th>Device Driver</th>
<th>Version</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>NVIDIA K20</td>
<td>352.93</td>
<td>7.5</td>
<td>nvcc V7.5.17</td>
</tr>
<tr>
<td>CUDA</td>
<td>NVIDIA GTX 780</td>
<td>331.79</td>
<td>6.01</td>
<td>nvcc V6.0.1</td>
</tr>
<tr>
<td>OpenCL</td>
<td>NVIDIA K20</td>
<td>352.93</td>
<td>1.2</td>
<td>gcc 4.4.7</td>
</tr>
<tr>
<td>OpenCL</td>
<td>NVIDIA GTX 780</td>
<td>331.79</td>
<td>1.1</td>
<td>gcc 4.8.1</td>
</tr>
<tr>
<td>OpenCL</td>
<td>AMD R280</td>
<td>1800.5</td>
<td>2.0</td>
<td>gcc 4.8.1</td>
</tr>
<tr>
<td>OpenCL</td>
<td>AMD R9 259X2</td>
<td>1598.5</td>
<td>2.0</td>
<td>gcc 4.8.1</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Xeon Phi</td>
<td>1.2</td>
<td>1.2</td>
<td>gcc 4.4.7</td>
</tr>
<tr>
<td>C + OMP</td>
<td>Intel Xeon</td>
<td>NA</td>
<td>OMP 3.0</td>
<td>gcc 4.4.7</td>
</tr>
<tr>
<td>C + OMP</td>
<td>Xeon Phi</td>
<td>NA</td>
<td>OMP 3.0</td>
<td>gcc 4.4.7</td>
</tr>
</tbody>
</table>

Table 2: Device Table. This table shows which specific compilers, device drivers and framework versions used to obtain the results shown in this project for a specific framework on a specific platform.
Bibliography


