PERFORMANCE PORTABILITY FOR ROOM ACOUSTICS SIMULATIONS

Larisa Stoltzfus

School of Informatics
The University of Edinburgh
Edinburgh, UK
larisa.stoltzfus@ed.ac.uk

Alan Gray

EPCC
The University of Edinburgh,
Edinburgh, UK
a.gray@epcc.ed.ac.uk

Christophe Dubach

School of Informatics
The University of Edinburgh
Edinburgh, UK
christophe.dubach@ed.ac.uk

Stefan Bilbao

Acoustics and Audio Group,
The University of Edinburgh
Edinburgh, UK
s.bilbao@ed.ac.uk

ABSTRACT

Numerical modelling of the 3-D wave equation can result in very accurate virtual auralisation, at the expense of computational cost. Implementations targeting modern highly-parallel processors such as NVIDIA GPUs (Graphics Processing Units) are known to be very effective, but are tied to the specific hardware for which they are developed. In this paper, we investigate extending the portability of these models to a wider range of architectures without the loss of performance. We show that, through development of portable frameworks, we can achieve acoustic simulation software that can target other devices in addition to NVIDIA GPUs, such as AMD GPUs, Intel Xeon Phi many-core CPUs and traditional Intel multi-core CPUs. The memory bandwidth offered by each architecture is key to achievable performance, and as such we observe high performance on AMD as well as NVIDIA GPUs (where high performance is achievable even on consumer-class variants despite their lower floating point capability), whilst retaining portability to the other less-performant architectures.

1. INTRODUCTION

The finite difference time domain method (FDTD) is a well-known numerical approach for acoustic modelling of the 3D wave equation [1]. Space is discretised into a three-dimensional grid of points, with data values resident at each point representing the acoustic field at that point. The state of the system evolves through time-stepping: the value at each point is repeatedly updated using *finite differences* of that point in time and space. The so-called *stencil* of points involved in each update is determined by the choice of discretisation scheme for the partial differential operators in the wave equation [2]. This numerical approach is relatively computationally expensive, but amenable to parallelisation. In recent years, there has been good progress in the development of techniques to exploit modern parallel hardware. In particular, NVIDIA GPUs have proven to be a very powerful platform for acoustic modelling [3][4]. Most work in this area has involved writing code using the NVIDIA specific CUDA language, which is tied to this platform. Ideally, however, any software should be able to run in a portable manner across different architectures, such that the performance of alternatives can be explored, and different resources can be exploited both as and when they become available. It is non-trivial, however, to develop portable application source code that can perform well across different architectures: this issue of *performance portability* is currently of great interest in the general field of High Performance Computing (HPC) [5].

In this paper, we explore the performance portability issue for FDTD numerical modelling of the 3D wave equation. To enable this study, we have developed different implementations of the simulation, each performing the same task, including a CUDA implementation that acts as a "baseline" for use on NVIDIA GPUs, plus other more portable alternatives. This enables us to assess performance across multiple different hardware solutions: NVIDIA GPUs, AMD GPUs, Intel Xeon Phi many-core CPUs and traditional multi-core CPUs. The work also includes the development of a simple, adjustable abstraction framework, where the flexibility comes through the use of templates and macros (for outlining and substituting code fragments) to facilitate different implementation and optimisation choices for a room acoustics simulation. Both basic and advanced versions of an FDTD algorithm that simulates sound propagation in a room (i.e. a cuboid) are explored.

This paper is structured as follows: in Section 2, we give necessary background information on the computational scheme, the hardware architectures under study and the associated programming models; in Section 3, we describe the development of a performant, portable and productive room acoustics simulation framework; in Section 4 we outline the experimental setup for investigating different programming approaches and assessing performance; in Section 5 we present and analyse performance results; and finally we summarise and discuss future work in Section 6.

2. BACKGROUND

This paper is focused on assessing different software implementations of a room acoustics simulation across different types of computing hardware. While this project focuses strictly on single node development, the ideas in this work could easily be extended for use across multi-node platforms by coupling with a message-passing framework. In this section we give some necessary background details. We first describe the FDTD scheme used in this study. We then give details on the hardware architectures that we wish to assess and the native programming methods for these architectures. Finally we describe some pre-existing parallel programming frameworks that provide portability.

2.1. FDTD Room Acoustics Scheme

Wave-based numerical simulation techniques, such as FDTD, are concerned with deriving algorithms for the numerical simulation of the 3D wave equation:

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi \tag{1}$$

Here, $\Psi(\mathbf{x}, t)$ is the dependent variable to be solved for (representing an acoustic velocity potential, from which pressure and velocity may be derived), as a function of spatial coordinate $\mathbf{x} \in \mathcal{D} \in \mathbb{R}^3$ and time $t \in \mathbb{R}^+$.

In standard finite difference time domain (FDTD) constructions, the solution Ψ is approximated by a grid function $\Psi^n_{l,m,p}$, representing an approximation to $\Psi(\mathbf{x} = (lh, mh, ph), t = nk)$, where l, m, p, n are integers, h is a grid spacing in metres, and k is a time step in seconds. The FDTD scheme used for this work is given in [6], and is the standard scheme with a seven-point Laplacian stencil. According to this scheme, updates are calculated as follows:

$$\Psi^{n+1}_{l,m,p} = (2 - 6\lambda^2)\Psi^n_{l,m,p} + \lambda^2 S - \Psi^{n-1}_{l,m,p} \tag{2}$$

where

$$\begin{aligned} S = {} & \Psi^n_{l+1,m,p} + \Psi^n_{l-1,m,p} + \Psi^n_{l,m+1,p} \\ & + \Psi^n_{l,m-1,p} + \Psi^n_{l,m,p+1} + \Psi^n_{l,m,p-1}, \end{aligned} \tag{3}$$

The constant $\lambda = ck/h$ is referred to as the Courant number and must satisfy the stability condition $\lambda \leq 1/\sqrt{3}$. It can be seen that each grid value is updated based on a combination of two previous values at the same location, and contributions from each of the six neighbouring points in three dimensions (giving the six terms in S - see Figure 1). The benchmarks used in this study are run over

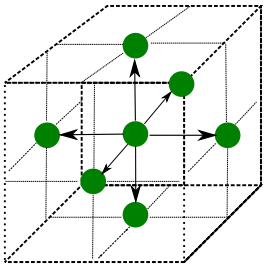

Figure 1: *Figure of a 7-point stencil on a three dimensional grid.*

13–105 million grid points at a sample rate of 44.1kHz to produce 100ms of sound. The boundary conditions used are frequency-independent impedance boundary conditions. We also include, towards the end of the paper, some results for comparison where a larger stencil is used, as described in [3][7].

2.2. Hardware and Native Programming Methods

The hardware architectures used in this work are multi-core CPU, GPU, and many-core CPU (Intel Xeon Phi) platforms. CPUs have traditionally been the main workhorses for scientific computing and are still targeted for the majority of scientific applications. GPUs have been gaining traction in the scientific computing landscape over the last decade, and can offer significant performance improvements over traditional CPUs for certain algorithms including wave-based ones like acoustic simulation. Xeon Phi chips are essentially multi-core CPUs, but with a large number of cores each with lower clock speeds and wider vector units. In this section, we describe these architectures in more detail and discuss how they are typically programmed.

2.2.1. Traditional Multi-Core CPUs

Computational simulations (like acoustic models) have historically been run on CPU chips. However, these architectures were originally optimised for sequential codes, whereas scientific applications are typically highly parallel in nature. Since the early 2000s, clock speeds of CPUs have stopped increasing due to physical constraints. Instead, multi-core chips have dominated the market meaning CPUs are now parallel architectures.

OpenMP [8] is one framework designed for running on shared memory platforms like CPUs. It is a multi-threading parallel model which uses compiler directives to determine how an algorithm is split and assigned to different threads, where each thread can utilise one of the multiple available cores. OpenMP emphasises ease of use over control, however there are settings to determine how data or algorithmic splits occur.

2.2.2. GPUs

GPUs were originally designed for accelerating computations for graphics, but have increasingly been re-purposed to run other types of codes [9]. As such, the architecture of GPUs has evolved to be markedly different from CPUs. Instead of having a few cores, they have hundreds or thousands of lightweight cores that operate in a data-parallel manner and can thus do many more operations per second than a traditional CPU. However most scientific applications, including those in this paper, are more sensitive to memory bandwidth: the rate at which data can be loaded and stored from memory. GPUs offer significantly higher memory bandwidths over traditional CPUs because they use graphics memory, though they are not used in isolation but as "accelerators" in conjunction with "host" CPUs. Programming a GPU is more complicated than a CPU as the programmer is responsible for offloading computation to the GPU with a specific parallel decomposition as well as managing the distinct memory spaces.

In this paper we investigate acoustic simulation algorithms using NVIDIA and AMD GPUs. NVIDIA GPUs are most commonly programmed using the vendor-specific CUDA model [10], which extends C, C++ or Fortran. CUDA provides functionality to decompose a problem into multiple "blocks" each with multiple "CUDA threads", with this hierarchical abstraction designed to map efficiently onto the hardware which correspondingly comprises multiple "Streaming Multiprocessors" each containing multiple "CUDA cores". CUDA also provides a comprehensive API to allow memory management on the distinct CPU and GPU memory spaces. CUDA is very powerful, but low level and non-portable. AMD GPUs are similar in nature to NVIDIA GPUs, but since there is no equivalent vendor-specific AMD programming model, the most common programming method is to use the cross-platform OpenCL, which we discuss in Section 2.3.1. For simplistic purposes, we will use the same terminology to describe the OpenCL framework as for CUDA.

2.2.3. Intel Xeon Phi Many-core CPU

The Xeon Phi was developed by Intel as a high performance many-core CPU for scientific computing [11]. One of the main benefits of the Xeon Phi is that it uses the same instruction set (X86) as the majority of other mainstream CPUs. This means that, theoretically, 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. There is also a wider vector instruction set on the Xeon Phi, which means that more instructions can be run in parallel per core than on a CPU or GPU. Depending on the algorithm, this can provide a boost in performance. The Xeon Phi currently straddles both the architecture and the performance of the CPU and GPU. The same languages and frameworks that are used for programming CPUs can be used on Xeon Phis.

2.3. Existing Portable Programming Methods

In this section we review existing portable parallel frameworks and APIs: OpenCL, a low-level API designed for use on heterogeneous platforms; TargetDP, a lightweight framework that abstracts data-parallel execution and memory management syntax in a performance portable manner, and a range of other frameworks offering higher levels of abstraction and programmability. These frameworks and APIs are designed to allow computational codes (like room acoustics) to be portable across different architectures, however they can be difficult to program in and they do not all account for performance across hardware.

2.3.1. OpenCL

OpenCL [12] is a cross-platform API designed for programming heterogeneous systems. It is similar in nature to CUDA, albeit with differing syntax. Whereas CUDA acts as a language extension as well as an API, OpenCL only acts as the latter resulting in the need for more boilerplate code and provides a more low-level programming experience. OpenCL can, however, be used as a portable alternative to CUDA, as it can be executed on NVIDIA, AMD and other types of GPUs, as well as manycore CPUs such as the Intel Xeon Phi. OpenCL is compatible with the C and C++ programming languages.

2.3.2. targetDP

The targetDP programming model [13] is designed to target data-parallel hardware in a platform agnostic manner, by abstracting the hierarchy of hardware parallelism and memory systems in a way which can map on to either GPUs or multi/many-core CPUs (including the Intel Xeon Phi) in a platform agnostic manner. At the application level, targetDP syntax augments the base language (currently C/C++), and this is mapped to either CUDA or OpenMP threads (plus vectorisation in the latter case) depending on which implementation is used to build the code. The mechanism used is a combination of C-preprocessor macros and libraries. As described in [13], the model was originally developed in tandem with the complex fluid simulation package *Ludwig*, where it exploits parallelism resulting from the structured grid-based approach. The lightweight design facilitates integration into complex legacy applications, but the resulting code remains somewhat low-level so it lacks productivity and programmability.

2.3.3. Other Methods

Higher level approaches focus more on distinct layers of abstraction that are far removed from the original codes. These approaches include: parallel algorithmic skeletons, code generators, DSLs (Domain Specific Languages), autotuners, combinations thereof and others. Higher-level frameworks can also provide 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. Many of these higher-level frameworks build on the work of skeleton frameworks, which focus on the idea that many parallel algorithms can be broken down into pre-defined building blocks [14]. 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 the stencils found in room acoustics models. These frameworks then simplify the process of writing complex parallel code by providing an interface which masks the low-level syntax and boilerplate. A code generator can either be a type of compiler or more of a source to source language translator. Other higher-level approaches include functional DSLs with auto-tuning [15], rewrite rules [16], skeleton frameworks combined with auto-tuning [17] 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 [18]. Exastencils 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 [19].

3. PORTABLE AND PRODUCTIVE FRAMEWORK DEVELOPMENT

The room acoustics simulation codes used in this work were previously tied to a specific platform (NVIDIA GPUs) through their CUDA implementation. In this study we compare the CUDA (pre-existing), OpenCL, and targetDP frameworks (all with C as a base language). In addition, we introduce the newly developed abstraction framework titled *abstractCL* (with C++ as a base language). In this section we describe our approach in enhancing performance portability and productivity through the development of this new framework for investigating room acoustics simulation codes. First, details about the implementation are discussed followed by the benefits of creating such a framework for the field of acoustics modelling.

3.1. Overview

abstractCL was created to make room simulation kernels on-the-fly, depending on the type of run a 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 and definitions in the main algorithm itself. The data abstractions and optimisations investigated for this project include: thread configuration settings, memory layouts and memory optimisations. Algorithmic changes can be introduced by adding new classes to the current template for more complicated codes.

3.2. Functionality

abstractCL works through the use of flags which determine what version should be run. Certain functions must always be defined as dictated by a parent class. However, those functions' implementations can be pulled in from different sources and concatenated together to create the simulation kernel before compilation. This framework runs similarly to the other benchmark versions, apart from that the kernel is created before the code is run (which creates more initial overhead). It was developed in C++ (due its built-in functionality for classes, templates, inheritance and strings) as well as OpenCL.

3.3. Advantages

One of the main benefits of creating an abstraction framework in this manner is that room acoustics simulation codes would not need to be rewritten to test out new optimisations. This makes it easier to use than a normal OpenCL implementation. Optimisations can be swapped in and out from the same point, limiting the room for error. Additionally, abstractions and performance can often be at war with each other when developing codes. abstractCL provides the opportunity to explore this tension at the most basic level for these simulations by allowing the data type representing the grid (and grid points) to be implemented in different ways that can be changed easily. For example, when accessing a data point, it could be stored in a number of different places in different memories. Using abstractions that mask implementation, the performance effects of these different implementations can then be investigated and compared. Though the optimisations in this project focus primarily on GPU memories, the framework 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).

4. EXPERIMENTAL SETUP

In this section we describe our setup for investigating alternative implementations of room acoustics benchmarks and our means of assessing their performance on different architectures. First we introduce the environment used in this study, including the separate platforms and benchmarks. Then the analysis including metrics and domain sizes used is described.

4.1. Environment

4.1.1. Hardware

The platforms used for this study are specified in Table 1. Included are two NVIDIA GPUs, two AMD GPUs, an Intel Xeon Phi manycore CPU and a traditional Intel Xeon CPU. Of the two NVIDIA GPUs, the consumer-class GTX-780 has much reduced double precision floating point capability over the high-end K20 variant, but offers higher memory bandwidth. Of the AMD GPUs, the R9 259X2 has higher specifications than the R280 and produces the best results overall.

4.1.2. Memory Bandwidth Reference Benchmark

As described in Section 2, memory bandwidth can be critical to obtaining good performance (and this will be confirmed in Section 5). It is therefore important to assess our results relative to what is expected given the memory bandwidth capability of a particular architecture. However, the peak values presented in Table 1 are rarely achievable in practice. STREAM [20] is an industry standard benchmark which was run on each architecture to provide a reference instead (through simple operations requiring data access from main memory).

4.2. Analysis

4.2.1. Metrics

The different versions of codes were compared using performance timings (time run in seconds), megavoxels/second, and data throughput (in GB/s). Time is determined by running the application with timing calls in place at key points in the code (to determine how much time is spent in the main computational kernel, the secondary kernel, for data IO and miscellaneous time). Megavoxels per second is found by multiplying the volume of the room by the number of time steps simulated and dividing by the simulation run time in seconds. The data throughput is calculated by multiplying the size of the room by the number of bytes accessed for every grid point.

4.2.2. Acoustical Model Sizes

The "room" used in the comparison runs is a rectangle box. Two different sized boxes (rooms) were used in the simulation runs: 256 x 256 x 202 points and 512 x 512 x 402 points. The purpose of using two room sizes is to see what kind of impact there is from increasing the domain space. 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 then scales with the audio sample rate chosen (in this study this is set to 44.10kHz). All versions use double precision as single precision can incur rounding errors in certain cases.

5. RESULTS

In this section we present the results of comparing the different room acoustics model implementations described previously as run on the selected hardware platforms. We first present the best performing results on each architecture to provide an overall assessment of the hardware. We then compare the applicability, performance and portability of these different runs. We go on to show that memory bandwidth is critical to achieving good performance. Finally, we describe the effect of optimisations and extend results to a more complex version of the codes.

5.1. Overall Performance Comparison Across Hardware

In this section we give an overview of performance achieved across the different hardware platforms, to assess the capability of the hardware for this type of problem. Figure 2 shows the time taken for the test case where we choose the best performing existing software on each architecture. It can be seen that the GPUs show similar times, with the AMD R9 295X2 performing fastest. This result translates to 8466 megavoxels updates per second. Another point of interest is that the consumer-class NVIDIA GTX780 is significantly faster for the CUDA implementation than the K20, even though it has many times lower double precision floating point capability. This is because, as discussed further in Section 5.3, memory bandwidth is more important that compute for this type

Table 1: *Specification of Different Hardware Architectures Used. Note that the "Ridge Point" is the ratio of Peak Gflops to Peak Bandwidth (in the terminology of the* ROOFLINE *Model).*

Platform	**Number of Cores/ Stream Processor**	**Peak Bandwidth (GB/s)**	**Peak GFlops** (Double Precision)	**Ridge Point (Flops/Byte))**	**Memory (MB)**
AMD R9 259X2	2816	320	716.67	2.24	4096
AMD R280	2048	288	870	3.02	3072
NVIDIA GTX 780	2304	288.4	165.7	0.57	3072
NVIDIA K20	2496	208	1175	5.65	5120
Xeon Phi 5110P	60	320	1011	3.16	8000
Intel Xeon E5-2620	24	42.6	96	2.25	16000

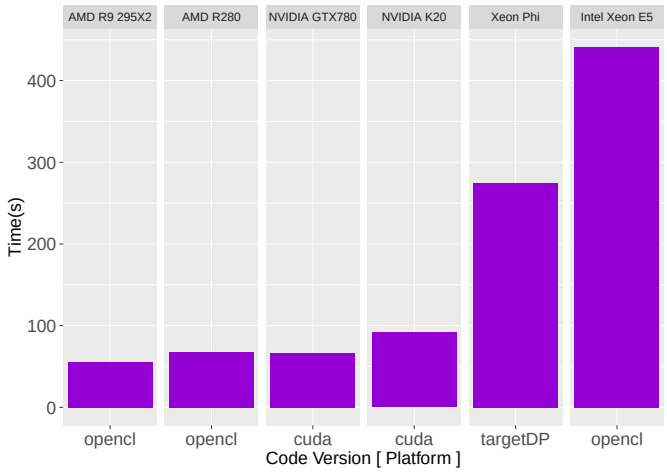

Figure 2: *Original fastest (optimised) versions across platforms for simple room acoustics simulation of room size 512×512×404. The timings shown produce 100ms of sound.*

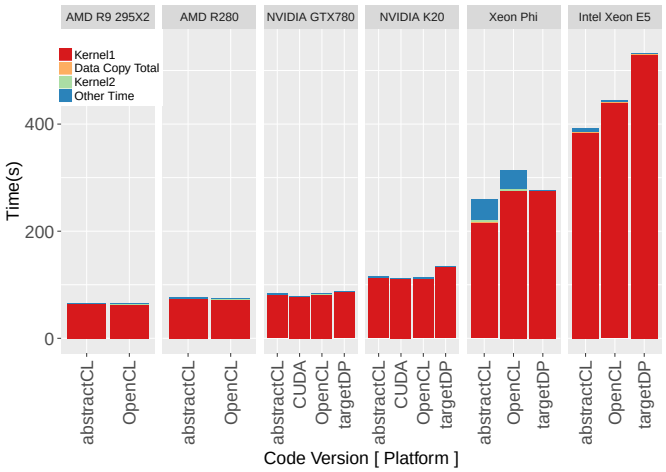

Figure 3: *Original timings of the simple room acoustics simulation for room size 512×512×404 on all architectures tested. The timings shown produce 100ms of sound.*

of problem. The Xeon Phi is seen to offer lower performance that the GPUs, but remains faster than the traditional CPU.

5.2. Performance Comparison of Software Versions

In this section we analyse the differences in performance resulting from running different software frameworks on a particular platform. In Figure 3, it can be seen how the performance depends on the software version. The main result is that for each architecture the timings are comparable, in particular in comparison to the *abstractCL* version. On the NVIDIA GPU, there is a small overhead for the portable frameworks (OpenCL, abstractCL and targetDP) relative to use of CUDA, but we see this as a small price to pay for portability to the other platforms. In particular, the newly developed framework abstractCL shows comparable performance to the original benchmarks and those written in OpenCL, indicating that it is possible to build performant, portable and productive room acoustics simulations across different hardware.

5.3. Hardware Capability Discussion

In this section, we further analyse the observed performance in terms of the characteristics of the underlying hardware. The ROOFLINE model, developed by Williams et al. [21], can be used to determine for a given application the relative importance of floating point computation and memory bandwidth capabilities. It uses the concept of "Operational Intensity" (OI): the ratio of operations (in this case double precision floating point operations) to bytes accessed from main memory. The OI, in Flops/Byte, can be calculated for each computational kernel. A similar measure (also given in Flops/Byte and known as the "ridge point"), exists for each processor: the ratio of peak operations per second to the memory bandwidth of the processor. Any kernel which has an OI lower than the ridge point is limited by the memory bandwidth of the processor and any which has an OI higher than the ridge point is limited by the processor's floating point capability.

The OI for the application studied in this paper is 0.54, which is lower than the ridge points of any of the architectures (given in Table 1), indicating that this simplified version of the application is memory bandwidth bound across the board (and thus not sensitive to floating point capability). This explains why the NVIDIA GTX780 performs so well despite the fact that it has very low floating point capability: its ridge point of 0.57 is still (just) higher than the application OI.

The blue columns in Figure 4 give observed data throughput (volume of data loaded/stored by the application divided by runtime, assuming perfect caching), for each of the architectures. The black lines give the peak bandwidth capability of the hardware (as reported in Table 1). It can be seen that, for all but the Xeon Phi architecture, the measured throughput varies in line with the peak

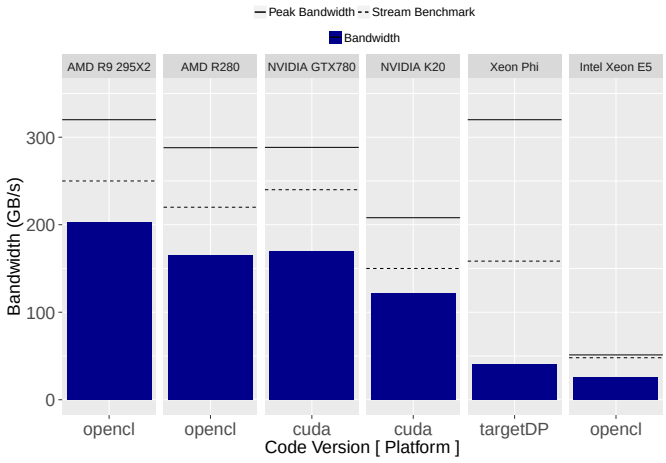

Figure 4: *Data Throughput for different versions of the room acoustics benchmark across platforms for room size 512×512×404.*

bandwidth, evidence that the overall performance of this application can largely be attributed to the memory bandwidth of that architecture. Since it is often very difficult to achieve near peak performance, we also include (with dashed horizontal lines) STREAM benchmark results (see Section 4.1.2), which give a more realistic measure of what is achievable. It can be seen that our results are, in general, achieving a reasonable percentage of STREAM but we still have room for improvement. In addition to running STREAM, profiling was also done on a selection of the runs. These results showed higher values than our measured results, indicating that our assumption of perfect caching (see above) is not strictly true and there may be scope to reorganise our memory access patterns to improve the caching. The Xeon Phi is seen to achieve a noticeably lower percentage of peak bandwidth relative to the other architectures, and this warrants further investigation.

5.4. Optimisation

Two optimisation methods were explored for the GPU platforms: shared and texture memory. Texture 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). The use of texture memory is relatively straightforward in CUDA, requiring only an additional keyword for input parameters. OpenCL is restricted to an earlier version on NVIDIA GPUs which does not support double precision in texture memory, so these results are not included. Shared memory is specific to one of the lightweight "cores" of a GPU (streaming multi-processor on NVIDIA) and can only can be shared between threads utilising that core, so can be useful for data re-use. A 2.5D tiling method was used to take advantage of shared memory [6].

Figure 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 shared and texture memory (for the CUDA version), *shared* uses only shared memory and *none* uses no memory optimisations. As before, different platforms are indicated by separate segments of the graphs. The reason for comparing to an optimised thread configuration version instead of the original run was to isolate what effect the memory optimisations had. Both room sizes (large and small) were run, but the large rooms had more significant differences in performance so are the only results shown. Overall the abstractCL version showed the most consistent improvement, however in this graph it is more clear that it is not quite as fast as the OpenCL version due to the overhead of using a more productive framework. All versions showed some improvement with this use of shared memory, but this amount varied per version and per room size across the different architectures. One of the reasons these results do not show more improvement is because the codes are already close to peak bandwidth as is discussed in Section 5.3.

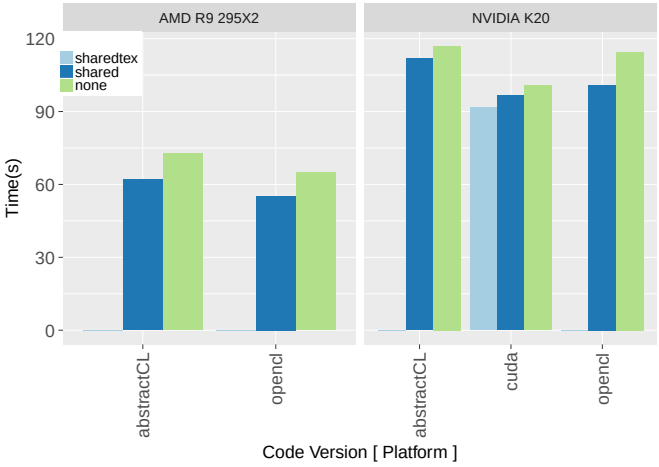

Figure 5: *Memory optimisations for CUDA, OpenCL and abstractCL versions of the room acoustics benchmarks run for room size of 512×512×404 on an NVIDIA and AMD GPU. The timings shown produce 100ms of sound.*

5.5. Advanced Simulations

Results in this paper have thus far only been presented for a very simplified problem: wave propagation is assumed lossless and the simplest FDTD scheme (employing a seven-point stencil) is used. It is thus of interest to explore more complex models of wave propagation, as well as improved simulation designs. Two new features were investigated for these so-called "advanced" codes: air viscosity (see [6]) and larger stencil schemes of leggy type (see [3]). The main algorithmic difference in adding viscosity is that another grid is passed into the main kernel and more computations are performed. Schemes operating over leggy stencils require access to grid values beyond the nearest neighbours. For this investigation, 19-point leggy stencils were introduced (three points in each of the six Cartesian directions as well as the central point, see Figure 1 in [3]). The comparison was done on a smaller scale, however, with the intention only of giving a general idea of whether or not the codes performed similarly. Thus, only CUDA and OpenCL versions were tested. The variations 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 amongst the different advanced versions, as a comparison between versions on the same platform and also comparisons of the same version across platforms. Graphs in Figure 6 show the performance timings and the memory bandwidth of the advanced codes for the various implementations for the larger room size. In these graphs, the following versions are included: cuda (the original version), cuda_adv (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 set up in a similar way as those found in Figures 3 and 4, where the versions of the code run along the x-axis and the performance is on the y-axis (in seconds) and the separated parts of the graph indicate the platform it was run on. Here, the top graph shows performance and the bottom shows bandwidth.

Generally the performance profile of the advanced codes looks similar to what can be seen for the original codes in Section 5.1: the codes still run fastest on AMD R9 259X2 and slowest on the Xeon Phi. The versions run on the AMD R9 259X2 in comparison to the same 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. The combination advanced codes (*_leggyadv) are significantly slower across the board, particularly on the Xeon Phi.

This purpose of this analysis is to see what effect algorithmic changes (ie. number of inputs or floating point operations) in the same benchmark have when run on the same platform. How the performance changes with differences to the models of 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.

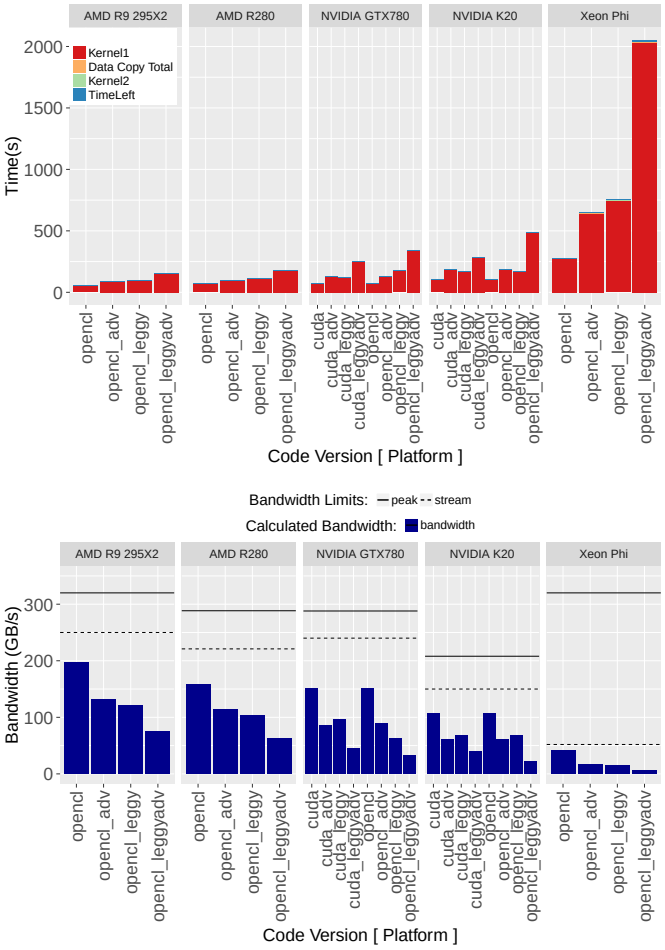

Figure 6: *Timing (top) and Bandwidth (bottom) for different versions of the advanced room acoustics benchmarks across platforms for room size 512×512×404. Smaller room results are not included as they show similar pattern.*

6. SUMMARY AND FUTURE WORK

In this paper we have shown that it is possible to implement room acoustics simulations in a way that allows the same source code to execute with good performance across a range of parallel architectures. Prior to this work, such simulations were predominantly tied to NVIDIA GPUs and we have now extended applicability to other platforms that were previously inaccessible. We have found that the main indicator of how the application will perform on a given architecture is the memory bandwidth offered by that architecture, due to the fact that the algorithm has low operational intensity. The best performing platforms are AMD and NVIDIA GPUs, due to their high memory bandwidth capabilities. The AMD R9 259X2 has the highest peak bandwidth of the GPUs tested, and was correspondingly found to be the best performing platform for the room acoustics simulation codes. In addition, we found the consumer-class NVIDIA GTX780 outperforms the HPC-specific NVIDIA K20 variant despite the fact it has many times lower floating point capability, due to insensitivity of the application to computational capability, as well as higher memory bandwidth of the former. Traditional CPUs have much lower memory bandwidth than GPUs, and measured performance was correspondingly low. The Intel Xeon Phi platform offers a high theoretical memory bandwidth, but we were unable to achieve a reasonable percentage of this in practice.

Performance-portable frameworks, including the abstractCL framework designed to allow flexibility in implementation options, were able to achieve similar performance to native methods, with only relatively small overheads (that we consider a small price to pay for the benefits that are offered by portability). However, relatively low-level programming is still required for these frameworks (including explicit parallelisation and data management). An ideal framework would offer performance portability, whilst allowing an intuitive definition of the scientific algorithm. Future work will

adapt a higher level framework named LIFT [22], currently under research and development at the University of Edinburgh, to enable it for 3D wave-based stencil computations. This framework aims, through automatic code generation, to allow execution of an application across different hardware architectures in a performance portable and productive manner. Future work will also look into extending abstract frameworks such as LIFT to modelling these codes across multiple parallel devices.

7. ACKNOWLEDGEMENTS

We thank Brian Hamilton and Craig Webb for providing the benchmarks used in this project and for answering questions about acoustics. We would also like to thank Michel Steuwer for providing invaluable advice about GPUs and OpenCL. 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.

8. REFERENCES

[1] Dick Botteldooren, "Finite-Difference Time-Domain Simulation Of Low-Frequency Room Acoustic Problems," *The Journal of the Acoustical Society of America*, vol. 98, no. 6, pp. 3302–3308, 1995.

[2] Stefan Bilbao, Brian Hamilton, Alberto Torin, et al., "Large Scale Physical Modeling Sound Synthesis," in *Stockholm Musical Acoustics Conference (SMAC)*, 2013, pp. 593–600.

[3] Brian Hamilton, Craig J Webb, Alan Gray, and Stefan Bilbao, "Large Stencil Operations For GPU-Based 3-D Acoustics Simulations," *Proc. Digital Audio Effects (DAFx),(Trondheim, Norway)*, 2015.

[4] Niklas Röber, Martin Spindler, and Maic Masuch, "Waveguide-Based Room Acoustics Through Graphics Hardware.," in *ICMC*, 2006.

[5] "Compilers and More: What Makes Performance Portable?," https://www.hpcwire.com/2016/04/19/compilers-makes-performance-portable/.

[6] Craig Webb, *Parallel Computation Techniques for Virtual Acoustics and Physical Modelling Synthesis*, Ph.D. thesis, University of Edinburgh, 2014.

[7] Jelle Van Mourik and Damian Murphy, "Explicit Higher-Order FDTD Schemes For 3D Room Acoustic Simulation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 12, pp. 2003–2011, 2014.

[8] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.5," November 2015, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf Accessed: 2016-08-12.

[9] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[10] NVIDIA, "Programming Guide: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2016-08-12.

[11] James Reinders, "An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessor," https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf. Accessed: 2016-08-05.

[12] Khronos OpenCL Working Group, "OpenCL Specification. Version 2.2," 2016, https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf Accessed: 2016-08-12.

[13] Alan Gray and Kevin Stratford, "A Lightweight Approach To Performance Portability With TargetDP," *The International Journal of High Performance Computing Applications*, p. 1094342016682071, 2016.

[14] Murray Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Ph.D. thesis, University of Edinburgh, 1989.

[15] Yongpeng Zhang and Frank Mueller, "Autogeneration and Autotuning Of 3D Stencil Codes On Homogeneous And Heterogeneous GPU Clusters," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 3, pp. 417–427, 2013.

[16] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel, "Operator Language: A Program Generation Framework For Fast Kernels," in *Domain-Specific Languages*. Springer, 2009, pp. 385–409.

[17] Cédric Nugteren, Henk Corporaal, and Bart Mesman, "Skeleton-based Automatic Parallelization Of Image Processing Algorithms For GPUs," in *Embedded Computer Systems (SAMOS), 2011 International Conference On*. IEEE, 2011, pp. 25–32.

[18] Joshua Auerbach, David F Bacon, Perry Cheng, et al., "Growing A Software Language For Hardware Design," *1st Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32, pp. 32–40, 2015.

[19] Christian Lengauer, Sven Apel, Matthias Bolten, et al., "Exastencils: Advanced Stencil-Code Engineering," in *European Conference On Parallel Processing*. Springer, 2014, pp. 553–564.

[20] John D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec 1995.

[21] Samuel Williams, Andrew Waterman, and David Patterson, "Roofline: An Insightful Visual Performance Model For Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[22] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach, "Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions To High-Performance Opencl Code," *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 205–217, 2015.

[23] George Teodoro, Tahsin Kurc, Jun Kong, et al., "Comparative Performance Analysis Of Intel Xeon Phi, GPU And CPU," *arXiv preprint arXiv:1311.0378*, 2013.