|epcc|

Modelling Room Acoustics with a GPU

Jacob Josiah Webber

18 August 2017

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2017

Abstract

The modelling of room acoustics is useful to determine how an environment will affect sound. These models can be done computationally by solving the wave equation in three dimensions with finite-difference time-domain techniques (FDTD).

This work replicates the work done in [1] with a slightly more complex set of boundary conditions applied as set out in [2]. In addition, a new technique and data structure were proposed and implemented that improved performance.

The nature of FDTD models and their application to room acoustics requires that a stencilling operation is performed on a very large grid representation of the room being modelled. With irregularly shaped rooms, at present, a *bounding box* method is used. This requires grid points that are outside the room to be stored, even though their values are not relevant. When GPUs are used for acoustic models, this grid is decomposed into small 3D blocks that are each processed by a compute kernel. In the new method outlined here, only those blocks that contain any points that are internal to the room are stored. These are stored in an array of structs, each struct of which contains the grid points for the block, as well as the locations of the neighbouring blocks.

This method significantly reduced the amount of unnecessary data storage compared with the bounding box technique - in some cases by more than a factor of 30. In addition, the stencilling performance was improved by an amount equivalent to the data savings, as the rate at which data was processed remained relatively constant.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I would like to extend my sincere thanks to Paul Graham and Dr Brian Hamilton for agreeing to supervise this project and all their help along the way.

Chapter 1

Introduction

"There is no competition of sounds between a nightingale and a violin."
Dejan Stojanovic

The way we perceive sound depends on the environment in which we hear it, as well as how the sound was created. Listening to a sound in a space like a church or a cathedral is very different from listening in a bedroom. Being able to predict how a given environment will sound has practical use in the design of new buildings, particularly those where acoustic performance is critical. A great deal of effort goes into the design of new concert halls. The results could enhanced by accurate acoustic models of spaces. The aim then of research in this area is to provide an accurate prediction of how sound will manifest itself given the physical properties of the space in which it is heard.

The key technique currently used for these simulations is the finite-difference time-domain (FDTD) method. This is used to find approximate solutions to the wave equation across a three dimensional space. The FDTD process requires that a stencil operation be performed. This stencil process must be iterated across a vast number of points in memory. The requirement for high memory bandwidth means that the problem is well suited to the use of graphics processing units (GPUs). These have high band width graphics memory and can operate on very many points at once in a massively parallel fashion.

In the work leading up to this report, some currently state-of-the-art GPU acoustics techniques as outlined in [1] and [2] were recreated. In addition to this, a novel technique that performs these simulations was devised and implemented. The proposed approach is designed to make best use of graphics memory when rooms are irregularly shaped. This is important because graphics memory is relatively scarce.

GPU threads can be controlled in three dimensional blocks. On NVIDIA devices these blocks can contain up to 1024 threads. When performing FDTD stencils of irregularly shaped rooms, current methods will result in there being many blocks assigned to regions that are completely outside the room being simulated. The new method proposed here decomposes rooms into blocks in space that correspond to blocks of threads. This means that space and compute is not wasted on empty blocks. These blocks are large enough that they remove some of the

typical drawbacks of unstructured data approaches but small enough that they avoid storing a significant amount of empty space. The fact that data is organised similarly to computation results in less branching logic. Branching can cause performance issues on GPUs.

In addition, the fact that these blocks are smaller means that neighbouring points in space will be closer together in memory. This means that the method is well suited to devices with high bandwidth memory. Such devices cache large amounts of data every time they access memory.

This report includes some background on the theory of acoustics and difference schemes, a brief background in GPUs and an overview of the new techniques that were devised and trialled in the work leading up to this report. This is followed by description of the details of the implementation in CUDA C as well as software development, profiling and debugging strategies and tools. There is then a comparative analysis of existing and novel techniques both in terms of performance in compute time and memory use. The report finishes with some conclusions and ideas for further work.

Chapter 2

Background and Literature Review

"He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast."

Leonardo da Vinci

2.1 Room Acoustics

2.1.1 Classical Acoustics

Sound is modelled using the wave equation which in one dimension is:

$$\frac{\partial^2 P}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} = 0 \tag{2.1}$$

where scalar field P is the deviation from the ambient air pressure, and c is the speed of sound. This result is derived by, among others, Richard Feynman in [3]. It is trivial to generalise this for 3 dimensional systems, which yields

$$\nabla^2 P - \frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} = 0 \tag{2.2}$$

where

$$\nabla \equiv \hat{\mathbf{i}}\frac{\partial}{\partial x} + \hat{\mathbf{j}}\frac{\partial}{\partial y} + \hat{\mathbf{k}}\frac{\partial}{\partial z} \tag{2.3}$$

Intuitively we know that a room's acoustic properties are dependent on its size and what materials form the walls and other surfaces. This is reflected in the solutions to the wave equation, where almost all environmental data is encoded in boundary conditions imposed on the P

scalar field. In real world applications these boundary conditions are too sophisticated for the wave equation to be solved analytically. This sophistication stems from both the shape of the room and the reflective properties of its boundaries. These reflective properties are a function of the frequency of the sound reflected.

2.1.2 Finite Difference Schemes

Finite difference schemes are therefore used to solve the wave equation for complex room shapes. These can provide an approximate numerical solution to the continuous wave equation.

A simple equivalence between first and second order derivatives and their corresponding central difference schemes is shown in 2.4 and 2.5.

$$\frac{df(u)}{du} \approx \frac{f(u+h) - f(u-h)}{2h} \tag{2.4}$$

$$\frac{d^2f(u)}{du^2} \approx \frac{f(u+h) - 2f(u) + f(u-h)}{h^2} \tag{2.5}$$

This can be easily discretised if h is set to be the distance in space or time between samples.

When discretising the space within the room, the scalar field of pressure within room is represented as a three dimensional array of floating point numbers. The letters l, m, and p are chosen to represent the samples index in each spacial dimension within the array, with n giving the time step. This means $P^n_{l,m,p}$ gives an approximation to P at the point (x, y, z) at time t where $x = lH$, $y = mH$, $z = pH$ and $t = nT$. T and H are the sample rates in time and space respectively.

When equation 2.2 is expanded out, it is shown that in order to compute a numerical approximation of the wave equation, discrete equivalents are required to the derivatives in time, and in each of the three spacial dimensions.

$$\frac{\partial^2 P}{\partial t^2} = c^2 \left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2} \right) \tag{2.6}$$

To replace these continuous partial derivatives a new finite difference operator, δ, is introduced. From 2.5 the following relations can be derived by simply replacing h with H:

$$\delta_t^2 P^n_{l,m,p} \equiv \frac{1}{T^2} \left(P^{n+1}_{l,m,p} - 2P^n_{l,m,p} + P^{n-1}_{l,m,p} \right) \approx \frac{\partial^2 P}{\partial t^2} \tag{2.7}$$

$$\delta_x^2 P^n_{l,m,p} \equiv \frac{1}{H^2} \left(P^n_{l+1,m,p} - 2P^n_{l,m,p} + P^n_{l-1,m,p} \right) \approx \frac{\partial^2 P}{\partial x^2} \tag{2.8}$$

$$\delta_y^2 P_{l,m,p}^n \equiv \frac{1}{H^2}\Big(P_{l,m+1,p}^n - 2P_{l,m,p}^n + P_{l,m-1,p}^n\Big) \approx \frac{\partial^2 P}{\partial y^2} \tag{2.9}$$

$$\delta_z^2 P_{l,m,p}^n \equiv \frac{1}{H^2}\Big(P_{l,m,p+1}^n - 2P_{l,m,p}^n + P_{l,m,p-1}^n\Big) \approx \frac{\partial^2 P}{\partial z^2} \tag{2.10}$$

It is now possible to use these operators to discretise 2.6.

$$\delta_t^2 P_{l,m,p}^n = c^2\big(\delta_x^2 P_{l,m,p}^n + \delta_y^2 P_{l,m,p}^n + \delta_z^2 P_{l,m,p}^n\big) \tag{2.11}$$

Equation 2.11 can then be expanded using the results of equations 2.7-2.10. This yields

$$\begin{aligned} P_{l,m,p}^{n+1} - 2P_{l,m,p}^n + P_{l,m,p}^{n-1} = \frac{c^2T^2}{H^2}\big(&P_{l+1,m,p}^n + P_{l-1,m,p}^n + P_{l,m+1,p}^n + P_{l,m-1,p}^n \\ &+ P_{l,m,p+1}^n + P_{l,m,p-1}^n - 6P_{l,m,p}^n\big) \end{aligned} \tag{2.12}$$

Letting the Courant number be defined as

$$\lambda \equiv \frac{cT}{H} \tag{2.13}$$

and our stencil

$$S \equiv P_{l+1,m,p}^n + P_{l-1,m,p}^n + P_{l,m+1,p}^n + P_{l,m-1,p}^n + P_{l,m,p+1}^n + P_{l,m,p-1}^n \tag{2.14}$$

gives a somewhat neater version of equation 2.12

$$P_{l,m,p}^{n+1} = (2 - 6\lambda^2)P_{l,m,p}^n + \lambda^2 S - P_{l,m,p}^{n-1} \tag{2.15}$$

More than simply being neat, this is now a representation of how to calculate new values for the three dimensional array from the previous ones using a simple stencil operation. This yields a new propagation relation.

2.1.3 Boundary Conditions

All of the physical properties of the room are encoded in the boundary conditions imposed on the differential equation. Modelling different boundary materials such as room walls is the subject of significant research at present. However, the main purpose of this report is to look at the computational aspects of applying stencil operations. As such a relatively concise treatment of the boundary conditions is given here.

For the purposes of a Von Neumann stencil most grid points that are inside the room will have 6 neighbours. However, on the edge of the room, there will be points that have fewer. The amount of these neighbours is stored in a new K grid, with one value for every point. This allows for the first time rooms of irregular shape to be considered, as those points outside the room are encoded with a zero K value.

The boundary scheme used modifies equation 2.15 by replacing the 6 value with a $K_{l,m,p}$ value. All values of $P^n_{l,m,p}$ where $K_{l,m,p} = 0$ are also assumed to be zero.

$$P^{n+1}_{l,m,p} = (2 - \lambda^2 K_{l,m,p})P^n_{l,m,p} + \lambda^2 S - P^{n-1}_{l,m,p} \tag{2.16}$$

In practice a slightly more sophisticated set of boundary conditions are used which model the reflectivity of the walls. The physics of this go beyond the scope of this project and report which were largely focussed on improving the memory capacity performance. The full stencil is shown in both the kernels in appendix C. This technique is taken from [2]. It is slightly more complex than the scheme used in [1], which should is noted when comparing performance (see section 5.3).

2.2 Stencilling Operations

A stencilling operation is one that updates array elements by combining its neighbours in some way. What is defined as a neighbour depends on the stencil being used. A Von Neumann stencil, as shown in figure 2.1, uses only the six elements that are directly next to a point (not diagonals). The S variable defined in equation 2.14 is an example of a Von Neumann stencil.

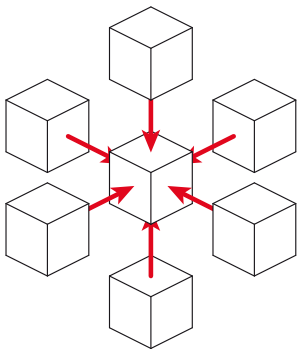

Figure 2.1: Diagram showing 3 dimensional Von Neumann stencil. Credit: Wikimedia Commons.

Stencilling operations are, by their nature, memory bound problems. This is because they involve a lot of data accesses per, relatively simple, calculation. This means that GPUs, with their high bandwidth memory, have been shown to be ideally suited to perform these operations [13]. However, while GPUs have the best performance, they also have limited amounts of memory.

2.3 Graphics Processing Units (GPUs)

GPUs were originally designed for 3D game rendering [4]. For this purpose they are equipped with the ability to process large arrays of floating point numbers very efficiently, both in terms of operations per second, and per Watt. It so happens that these qualities make these devices very well suited to the field of High Performance Computing (HPC). This coincidence is a happy one indeed, as it allows the very high research and development costs of bringing a new processor to market to be spread across the much larger gaming industry. The CEO of NVIDIA, which is the largest supplier of discrete graphics cards, described the budget of several billion US dollars for the design of the latest Pascal GPU architecture as so large it would probably be enough to send someone to Mars [5].

GPUs have a number of technical features which make them particularly suited to running FDTD simulations. GPUs were shown in [1] to perform up to 2 orders of magnitude better than CPUs when running room acoustics simulations.

2.3.1 GPUs and parallelism

GPUs primarily achieve their massive performance increases over CPUs by operating in a massively parallel fashion. Two principles of parallel computing are used to make this happen. These are the *single instruction, multiple threads* (SIMT) and *simultaneous multi-threading* (SMT) approaches.

Single Instruction, Multiple Thread

SIMT computer instructions execute on multiple objects in memory at once. In doing so they reduce the amount of silicone that is required to perform calculations. This is because multiple operations can share so called *control units* (CUs). These determine which instructions should occur, and when branches should be taken. GPUs are organised into *streaming multiprocessors* (SMXs). These contain multiple cores and therefore *floating point units* (FPUs), but share control circuitry. This means a much higher proportion of the silicone is taken up with performing floating point operations than with a conventional CPU, where there is typically one CU per processing core.

Figure 2.2: GPU and CPU architectures [6]

Figure 2.2 goes some way to showing the make-up of GPUs in relation to CPUs, without showing a detailed view of each multiprocessor.

More detail on the SMX is given in figure 2.3. This shows each SMX contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST) [7]. Instructions are scheduled and issued in groups of 32 parallel threads called *warps*. Each SMX can run a maximum of 4 warps concurrently. This depends on there not being a *stuctural hazard*. This is when a part of the processor's hardware is needed by two or more instructions at the same time. For example, the fact that there are only 64 double precision FPUs per SMX means that not all $4 \cdot 32$ threads from the four warps can conduct floating point calculations simultaneously.

Figure 2.3: The SMX architecture as in the K20 GPU [7]

Simultaneous Multi-Threading

SMT refers to the process of running a higher number of logical threads than the number of cores that exist in hardware. This means that processes can avoid using up resources while they are idle. In GPUs this happens at the warp level. When a SMX changes which warp is active this is known as *context switching*. GPUs have special hardware adaptations that make them very efficient at doing this. They have a very large amount of registers compared with conventional CPUs. This means that data from all *active* warps - meaning those that have been allocated to an SMX - are kept in registers. This differs from a conventional CPU where this data may have to be written out to at least L1 cache memory. The speed with which GPUs can context switch means that SMT techniques become appropriate to hide shorter latencies such as memory accesses, unlike CPUs, which are more likely to make use of SMT for threads that are idle while waiting for disk or network latencies.

As the efficiency of SMT on the GPU is dependent on large registers, performance can be limited when warps are required to complete tasks that require large amounts of registers. For ideal performance, GPUs need to be carefully programmed to ensure that they make maximal use of SMT. This reduces the amount of time hardware spends idling, and therefore increases throughput.

2.3.2 GPU Memory Hierarchies

Memory Type	Scope	Latency (approx. clock cycles)
Register File	CUDA Thread	0
Shared Memory/L1 Cache	CUDA Block / N/A	48
L2 Cache	N/A	120
Global Memory	CUDA Grid	440

Table 2.1: Memory hierarchy on NVIDIA GPU.

GPUs are tuned for throughput, rather than latency. This colours their approach to data storage as well as processing.

The previous section discusses the importance of registers for SMT. These form the top tier of the GPUs memory hierarchy. Modern NVIDIA GPUs, such as the K20 used in this project, also have two additional layers of cache memory above the global memory store. The top layer of this cache is also referred to as *shared* memory. This is because data can be explicitly loaded here and will then be shared among a group of threads defined by the programmer as a *block* (see section 2.3.3 for details). L2 cache is hardware controlled and functions similarly to CPU caches. Below this there is *global* memory. This is the largest and highest latency form of data storage on the GPU. There is also *texture memory*. This is only useful with two dimensional arrays so is not relevant to this project.

The K20 used in experiments for this report uses GDDR5 high bandwidth graphics memory for global memory. This differs from DDR3/4 used in conventional computing.

GDDR5 is tuned for maximal bandwidth. It does sacrifice some latency performance in exchange for this. This is achieved by loading in larger lines of data at a time. There is much excitement in GPGPU circles about the recent arrival of *high bandwidth memory 2* (HBM2). This has even higher bandwidth than GDDR5 and has even larger access widths.

Figure 2.4 shows the advance of memory bandwidth through time on both CPUs and GPUs. The NVIDIA K20 has a peak theoretical bandwidth of $208\,\mathrm{GB\,s^{-1}}$, which is less than a third of that of some of the latest HBM2 cards. AMD - who are NVIDIA's main competitor - have just released a card with HBM2 memory that claims to have a $1\,\mathrm{TB\,s^{-1}}$ memory bandwidth [8]. It is hoped the techniques described later in this report will be well suited to these advancements.

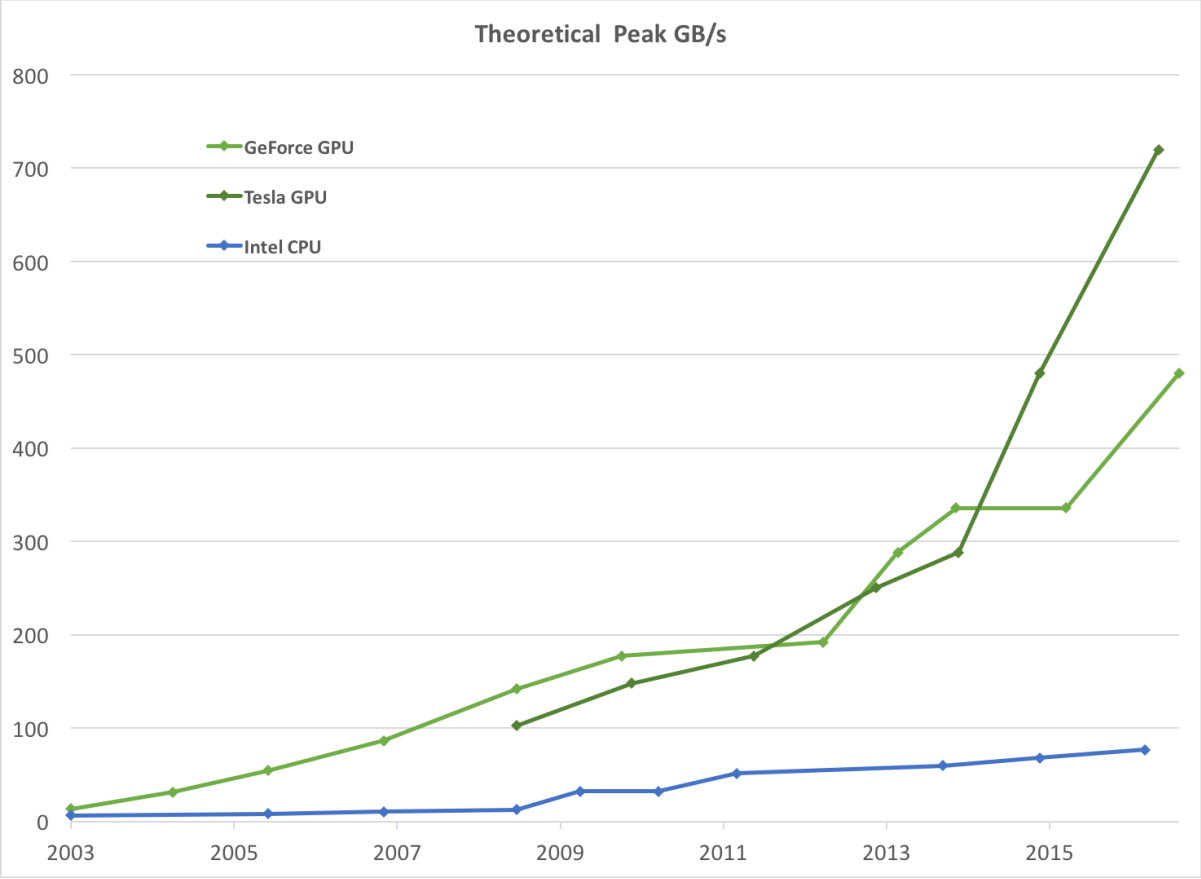

Figure 2.4: A graph from NVIDIA showing memory bandwidth of GPUs and CPUs through time [6].

2.3.3 The CUDA Paradigm

The nature of GPUs means that they need to be programmed carefully to extract performance. Computation done on the GPU is separated into kernels. In this case they were written as special CUDA C functions. These are launched in blocks and each block can contain multiple threads. When a kernel is called the amount of blocks and threads per block is specified.

The amount of threads per block can be specified using a `dim3` data type. This allows blocks of threads to be three dimensional. Inside a kernel each thread has an x y and z index. These indices can be used in a way so that each thread accesses a different data item without using control logic. For example, listing 2.1 shows a standard three dimensional array read in CUDA C - a version of CUDA that is an extension to ANSI C.

Listing 2.1: CUDA memory accesses

```
myData = data[threadIdx.z][threadIdx.y][threadIdx.x]
```

Such an access is considered idiomatic CUDA code and is well optimised in terms of memory read speed, particularly when the thread in the x direction corresponds to the fastest moving array index in memory. In C this is the last index specifier. Accessing data where the threads and data accesses follow the same order is known as *data coalescing*. Acessing data that is close together in memory is also important and is referred to as *data locality*.

These thread blocks are assigned to an SMX when resources become available. While they are waiting for global memory access latencies, SMT will be used to make a new warp active. As warps contain 32 threads and thread blocks can contain up to 1024, SMT can also be used within these thread blocks when multiple warps are required. The fact that threads are controlled in warps of 32 has a few ramifications for the programmer. Firstly it means that blocks that are smaller than 32 threads will waste compute capacity, as the remaining threads will execute on null data. It also means that branching logic is likely to introduce inefficiencies where it might require branching within a warp. This will result in all accessed branches being executed on all threads with those not activated operating on null data.

Shared memory has thread block scope. Once a thread is executed this memory is no longer available. Shared memory can be allocated statically or dynamically from within a kernel in CUDA C.

CUDA is one of many programming paradigms that are used for GPGPU computing. Others include OpenMP, OpenCL and C++ AMP. Unlike CUDA, these allow many brands of GPU to be programmed.

2.4 The State-of-the-art in FDTD Acoustical Modelling

Combining the ideas as written about in sections 2.1 and 2.3, it is possible to describe the most successful techniques thus far in running FDTD room acoustics models.

2.4.1 The Bounding Box Method

Current techniques are optimised for cuboid shaped rooms. It is easy, with such a room, to derive the $K_{l,m,p}$ value by adding a conditional statement to check whether the point is in the boundary of the cuboid. A sketch of this method is shown in algorithm 1. This was the method

used in [1]. In this sketch the update refers to the stencil scheme outlined in section 2.1. When the stencil is performed, the room is divided into blocks. These are each assigned to a CUDA kernel instance. A kernel can derive which points in memory to operate on from its block identities.

To perform the stencil two 3D arrays are allocated in GPU memory. One becomes $P^n_{l,m,p}$ and the other $P^{n-1}_{l,m,p}$. From these the next time step can be calculated as in equation 2.16. This value overwrites $P^{n-1}_{l,m,p}$. A pointer swap is then used so that this becomes $P^n_{l,m,p}$ and $P^n_{l,m,p}$ becomes $P^{n-1}_{l,m,p}$.

Data: Three dimensional array of $P^n_{l,m,p}$ and $P^{n-1}_{l,m,p}$
Result: Updated array of $P^{n+1}_{l,m,p}$
Get the 3D indices of the current thread from the block and thread IDs;
if *not at the boundaries* **then**
    Compute the linear address from the 3D indices;
    Overwrite node $P^{n-1}_{l,m,p}$ with $P^{n+1}_{l,m,p}$;
end
Swap pointers.

Algorithm 1: CUDA kernel for update on the basic 3D scheme.

To support arbitrarily shaped rooms as laid out in [2] it is necessary to encode the shape of the room in an array of $K_{l,m,p}$ values. In this case all blocks in the bounding box will be processed, even those that do not contain any points that are inside the room.

Data: Three dimensional arrays of $P^n_{l,m,p}$, $P^{n-1}_{l,m,p}$ and $K_{l,m,p}$
Result: Updated array of $P^{n+1}_{l,m,p}$
Get the 3D indices of the current thread from the block and thread IDs;
if $K_{l,m,p}$ *is not zero* **then**
    Compute the linear address from the 3D indices;
    Overwrite node $P^{n-1}_{l,m,p}$ with $P^{n+1}_{l,m,p}$;
end
Swap pointers.

Algorithm 2: CUDA kernel for update on an irregularly shaped room.

The update to $P^{n+1}_{l,m,p}$ is now a function of $K_{l,m,p}$. This results in an extra global memory read for every stencil point.

Block decomposition and caching.

The work up until here assumes that the grid is decomposed into three dimensional blocks. In [1] other decompositions were tried in addition to this.

The decomposition refers to how the data is shared between kernels. In addition to the 3D block decomposition, a 2D slicing decomposition was tried. Experiments were also done where data was chached into shared memory before the stencil was applied.

The 3D block decomposition without shared memory was in fact found to be the second fastest approach. A minor improvement of 6% to stencil running time performance was found when 2d slicing with shared memory caching was used.

Given the added complexity for a relatively small gain, the 2D slicing/shared memory approach was not replicated here. Instead the 3D block decomposition was used to provide a benchmark for the new method as outlined in chapter 3.

2.4.2 Memory Capacity

The performance of FDTD models can be measured using two factors: the computation time and the amount of memory used. Because of their high bandwidth memory and fast floating point performance, GPUs routinely perform very well in the computation time category. However, the graphics memory that they operate on is much more limited in its supply than standard DRAM. By way of example, the high end HPC targeted NVIDIA K20 device primarily used in this project has 5 GiB. This is significantly less than many modern laptops will have as main memory.

Going back to equation 2.12, it is possible to derive a stability relation for the value of the Courant number, or λ. The approximate solution to the differential equation is said to be stable if results do not grow exponentially. A full derivation is given in [11] using Von Neumann spectral analysis. This yields the important result that for stability it is required that $\lambda \leq \frac{1}{\sqrt{3}}$.

This gives a concrete relation between the sample rates in time ($\frac{1}{T}$) and space ($\frac{1}{H}$).

$$H \leq \sqrt{3}cT \tag{2.17}$$

High sample rates in time are required to avoid temporal aliasing. This is an effect where the ear lacks enough information to determine whether a signal is one of two possible values within the audible range ($\approx 20\,\mathrm{Hz}$-$20\,\mathrm{kHz}$). The Nyquist limit requires that the sample rate be twice as high as the highest audible frequency to avoid this aliasing effect [12]. This requires a very large number of points for even a moderate sized room. Taking a sample rate of $44.1\,\mathrm{kHz}$ (as is used in audio CDs), and the speed of sound in air as $344\,\mathrm{m\,s^{-1}}$ the Courant limit becomes $0.0135\,\mathrm{m}$. This means that to model a room of even $1\,\mathrm{m}^3$ a matrix of 75 by 75 by 75 floating point numbers is required. This equates to a total array size of $421\,875$. The means there is a requirement for $3.22\,\mathrm{MiB}$ or $1.61\,\mathrm{MiB}$ of memory for double or single precision respectively per m^3.

As an example, The Royal Albert hall is popularly claimed to have a volume of approximately $80\,000\,\mathrm{m}^3$. This would require up to $270\,\mathrm{GiB}$ just to meet the Courant limit at a sample rate of $44.1\,\mathrm{kHz}$.

In addition, the Courant limit simply provides a minimum number of points that will provide numerical stability. In order to get good results the number of points required is significantly higher. In the field of electro-magnetism - where FDTD techniques are most commonly used -

it has been shown that many tens of samples per wavelength are required to accurately model the propagation of a wave [14]. This makes sense at a more intuitive level that the result of the Courant calculation and yields an even more taxing requirement on memory quantities.

2.4.3 Memory Bandwidth

In addition to requirements on memory capacity, the performance of FDTD simulations on given hardware is largely determined its memory bandwidth. Two factors that affect the memory bandwidth performance of FDTD simulations are the load/store efficiencies and the compute to memory access ratio (CMA).

Stencil operations by their nature have a low ratio of floating point operations (FLOPS) to memory access. For ideal performance, data will be loaded from global memory to a register or to cache and then be operated on multiple times. For methods such as the stencil operation outlined in section 2.2, this CMA value approaches 1.0. This is because for each time step the stencil operation must be completed for all points in the grid followed by a process of synchronisation. This synchronisation overhead proves to be very expensive, as it requires that all points in utilised memory be read and written to for a relatively trivial amount of computation. Under a simple three dimensional Von Neumann stencil, as shown in figure 2.1, each point is used in 6 FLOPS. This yields a theoretical maximum CMA of 6.0. However, storing these values in local memory as described in [1] results in very marginal gains, in one case (see section 2.4.1) and significant losses in all others, all at the expense of more complex code.

This is because of a factor called *occupancy*, which is a measure of how many register files and how much L1 cache a compute kernel uses. High occupancy reduces the efficiency of SMT as outlined in section 2.3.1. Instead, any performance boost from data reuse comes from hardware managed L2 cache. Cache hit rates as found in the literature are low, meaning there is a very low effective CMA for this method [1].

GPU DRAM differs from CPU DRAM primarily in that it is optimised for bandwidth rather than latency. As a result each memory load and write can operate on a relatively large chunk of memory at a time. Global memory accesses using current GDDR5 technology are optimised for $128\,\mathrm{B}$ vectorised accesses. These are by default cached in L1 and L2 which also have $128\,\mathrm{B}$ sized cache lines. These large reads can be wasted if memory accesses are scattered. This is the case when large three dimensional arrays are stored. In such an array only neighbouring points in one dimension will be neighbouring in memory but in the other dimensions there is some considerable offset. This results in memory reads being wasted, cache lines being *overfetched*, or storing unnecessary data, as well as less efficient non-vectorised memory accesses being used. The first of these problems can be partially fixed by using a compilation flag to disable L1 cache. This technique is recommended in [6] for scattered memory accesses and was shown in [1] to yield marginal improvements in performance for the FDTD problem.

2.5 Other Methods

The methods outlined in this section are described in a deliberately terse way. They do not form the body of the work done for this dissertation. They are discussed here to offer some justification as to why not, as well as to give the FDTD approach some context within its field.

2.5.1 Ray Tracing

Ray tracing methods treat sound waves as rays that travel in straight lines. Sound waves are launched in multiple directions from a source position and those that cross a given output location are recorded. This has the combined effect of allowing some filtering to occur at the room boundary as well as providing a digital filtering effect from the differing path lengths of all the possible reflective paths that could lead the sound to the output location. This method yields credible results with much less computational expense than FDTD techniques and has therefore been used in the computer gaming and virtual reality industries [15]. It is, however, fundamentally limited by the fact it cannot model more complex wave effects like diffraction [2].

2.5.2 Spectral Methods

Stencil effects such as the FDTD approach can be shown to be equivalent to a convolution. It is well known that a convolution in the space domain is equivalent to a pointwise vector multiplication in the frequency domain:

$$h(x) = f \otimes g = \int_{-\infty}^{\infty} f(x-u)g(u)du = \mathcal{F}^{-1}\Big(\sqrt{2\pi}\mathcal{F}[f] \cdot \mathcal{F}[g]\Big) \tag{2.18}$$

Such an approach would remove the synchronisation overhead in the stencil operation and allow very high CMA ratios to be achieved by performing this pointwise multiplication of the Fourier transformed stencil and pressure field matrices to be performed multiple times on each memory location in turn, without having to read and write through all memory every time step.

While these approaches have been applied to stencil techniques in image processing [16] they are not applicable here. This is because the boundary conditions are the only feature that varies between different rooms and a technique for encoding these well in the frequency space is not known. In addition to model any input signal larger than one sample long it would be necessary to transform the arrays back from the frequency domain after every time step, removing any advantage from applying this method and adding a considerable overhead in the performance of two FFTs for every time step.

Chapter 3

The Proposed Data Structure and Method

"Nothing surpasses the beauty and elegance of a bad idea."

Craig Bruce

The bulk of the work in this project went into the design and implementation of a new algorithm for the computation of FDTD simulations. This section details the design of this algorithm and some of its theoretical performance benefits.

The purpose of this technique improve memory usage for irregularly shaped rooms and memory locality in large 3D FDTD acoustics simulations. The proposed method is to replace the large 3D arrays currently used with an array of structs. Each struct will store a smaller sub-array containing points in the simulation, as well as information on the location of neighbouring points to those points of the edge of each sub-array. This will allow blocks where no points are inside the room being simulated not to be stored in expensive GPU memory. This technique also improves data locality. The smaller 3D sub-arrays will result in neighbouring points being scattered less far apart in memory.

There is some analogue that can be found here with the field of sparse matrices. Indeed, the stencil operation can be thought of as a sparse matrix multiplication. When matrices are sparse, so-called *unstructured* approaches are used to store the data. With a standard matrix, a particular matrix index's location in memory can be derived easily. However, such an approach can require the storage of a lot of unnecessary data. With unstructured approaches, only non zero matrix elements need be stored. However, this means it is no longer possible to easily derive the memory location of an element in memory - meaning that this data must also be stored. This means that matrices must typically be very sparse indeed before unstructured approaches become practical.

If a bounding box representation can be thought of as a structured approach, and typical sparse representations as unstructured, here a technique is presented which can be thought of, and from here will be described as, a *semi-structured* approach. This is where smaller blocks of structured data are stored only if they contain at least one point that is non-zero - or inside the room. These blocks are sized so that they are large enough to dwarf any additional locational data requirements, but small enough so that not too much empty space need be stored.

The nature of the CUDA paradigm can also be taken advantage of. The blocks are designed so that they correspond to a CUDA block of threads. This means that one kernel can operate on one block in a parallel fashion. This relation means that each kernel can load one contiguous block of data from global memory. This makes this approach highly suited to devices that are optimised for memory bandwidth, and therefore have larger memory transaction sizes.

3.1 The Data Structure

Listing 3.1: The block Data Struct

```
typedef real bl_array[Bz][By][Bx];
struct block {
    int up;
    int down;
    int left;
    int right;
    int fore;
    int aft;
    bl_array u;
    bl_array u1;
    char k[Bz][By][Bx];
};
```

Listing 3.1 shows the block struct. The K array is a store of the amount of internal neighbours a point has. It is set to zero if the point in question is external. A block should therefore only be stored in the array of structs if it has at least one point with a K value above zero. This means rooms whose bounding box would contain large amounts of empty space can be stored much more efficiently.

The fact that it is a stencil operation means that each kernel need not know the location of the block it is operating on - only the location of the block's neighbours. These are therefore stored in the struct as six integers which give the block index of a neighbouring block in each direction.

These six numbers are an overhead of this method that could lead to it being less efficient in memory than the bounding box technique. However, the size of these in memory compared to the rest of the data is small.

Taking, as an example, the dimensions of the sub-arrays to be 8 by 8 by 8, then the total size requirement of a struct would be $1.002\,757$ times that of the representation without these neighbour integers stored when the user defined datatype, `real`, is set to be a double precision floating point number.

The worst case memory performance can be calculated for single precision as:

$$\frac{S_{ss}}{S_s} = 1 + \frac{24}{9 \cdot \mathrm{Bx} \cdot \mathrm{By} \cdot \mathrm{Bz}} \tag{3.1}$$

and for double precision as:

$$\frac{S_{ss}}{S_s} = 1 + \frac{24}{17 \cdot \mathrm{Bx} \cdot \mathrm{By} \cdot \mathrm{Bz}} \tag{3.2}$$

Where S_{ss} and S_s are the sizes of the semi-structured and structured representations in memory respectively.

Some discussion has been given of the analogue between the semi-structured approach and sparse matrix representations. This data structure also has some similarities to a *linked list*. Such a structure is a data object where each element stores the location of the next point in the list. The struct in listing 3.1 also has some similarities to this. However, instead of storing the next in the list, it stores an index of its neighbours in all six directions.

3.2 The Algorithm

3.2.1 Forming the Data Structure

The new method requires a new algorithm for converting a room's representation to a semi-structured one, as well as for performing the stencil on the new data structure.

Data: Three dimensional array of $K_{l,m,p}$ for a given room
Result: Array of blocks (AOB), block locations, number of internal blocks (T_i)
Divide $K_{l,m,p}$ into Bx by By by Bz sized blocks;
while *not at end of blocks* **do**
    do nothing;
    if *block contains non-zero* $K_{l,m,p}$ *value* **then**
        increment T_i;
        store block's location;
    else
        store zero as block location
    end
    go to next block;
end
allocate memory for T_i block structs;
while *not at end of block locations* **do**
    if *block location has non-zero value* **then**
        copy $K_{l,m,p}$ values to correct block in AOB;
    end
    go to next block location;
end

Algorithm 3: Convert bounding box to array of blocks

A sketch of this is given in algorithm 3. It requires two passes over the data as it necessary to know the amount of blocks that have internal points before memory can be allocated and the K sub-arrays can be copied to the new representation.

3.2.2 Stencilling Algorithm

The new data structure also requires a new stencilling algorithm. Within each block the algorithm can remain the same as with the bounding box method, but it is necessary to read in the edges of the neighbouring blocks to calculate the stencils for those points at the edge of a block. The 6 integers that are stored in the data structure make this easy to do. Unlike with typical sparse methods, no search or look up is required to find a specific block. Shared memory was utilised to cache each block and collect the neighbouring elements to form a *halo* region around the data in the block.

Data: Array of blocks (AOB).
Result: Updated AOB
Get the current block from CUDA block ID;
Allocate array in shared memory that is 2 points larger in each dimension than the block;
Copy data from block to the centre of the shared array;
Copy neighbouring blocks data to the edges of the shared array;
if $K_{l,m,p}$ *is not zero* **then**
  Overwrite node $P^{n-1}_{l,m,p}$ with $P^{n+1}_{l,m,p}$;
end

Algorithm 4: CUDA kernel for update on an array of blocks.

A sketch of this algorithm is shown in algorithm 4. More detail can be seen in the kernel itself which is included as an appendix.

It should be noted that unlike the algorithms outlined in section 2.2, it is not possible to do a pointer swap within the struct between the u and u1 arrays that represent the $P^{n-1}_{l,m,p}$ and $P^{n}_{l,m,p}$ grids. This is because they are statically allocated arrays. It is not possible to allocate arrays dynamically within a struct in ANSI C. Therefore, two versions of the kernel were written, one which takes u to be $P^{n-1}_{l,m,p}$ and u1 to be $P^{n}_{l,m,p}$, and the other vice versa.

3.2.3 Input/Output

For the models to be useful they need to give some sort of output. They also need to model how a specific signal will be modified by being played in a given room. This is because a room's acoustical performance is a function of the frequency of the input signal.

A special I/O kernel was written that takes a 1D array of the input signal and outputs a 1D array of the output from the room. This is also given the I/O locations in the room as an argument. This is because how the rooms sounds depends on where the listener and the sound source are. This kernel is run once after every stencil iteration.

To find the I/O grid locations it is necessary to use the *block locations* array. From this it is possible to derive the location of every point in the AOB. A function was written that converts a structured grid location to its point in the AOB.

3.3 Theoretical Benefits

The proposed semi-structured method's benefits are most obvious on rooms that would occupy a small part of their bounding box. Such rooms can be thought of a *sparsely* populated - to borrow a term from the field of linear algebra.

A typical example is shown in figure 3.1. Churches and cathedrals are often used for musical performances, as well as services, so their acoustic performance is important. They are also ill suited to ray-tracing methods because they have pronounced diffraction effects around their sharp corners.

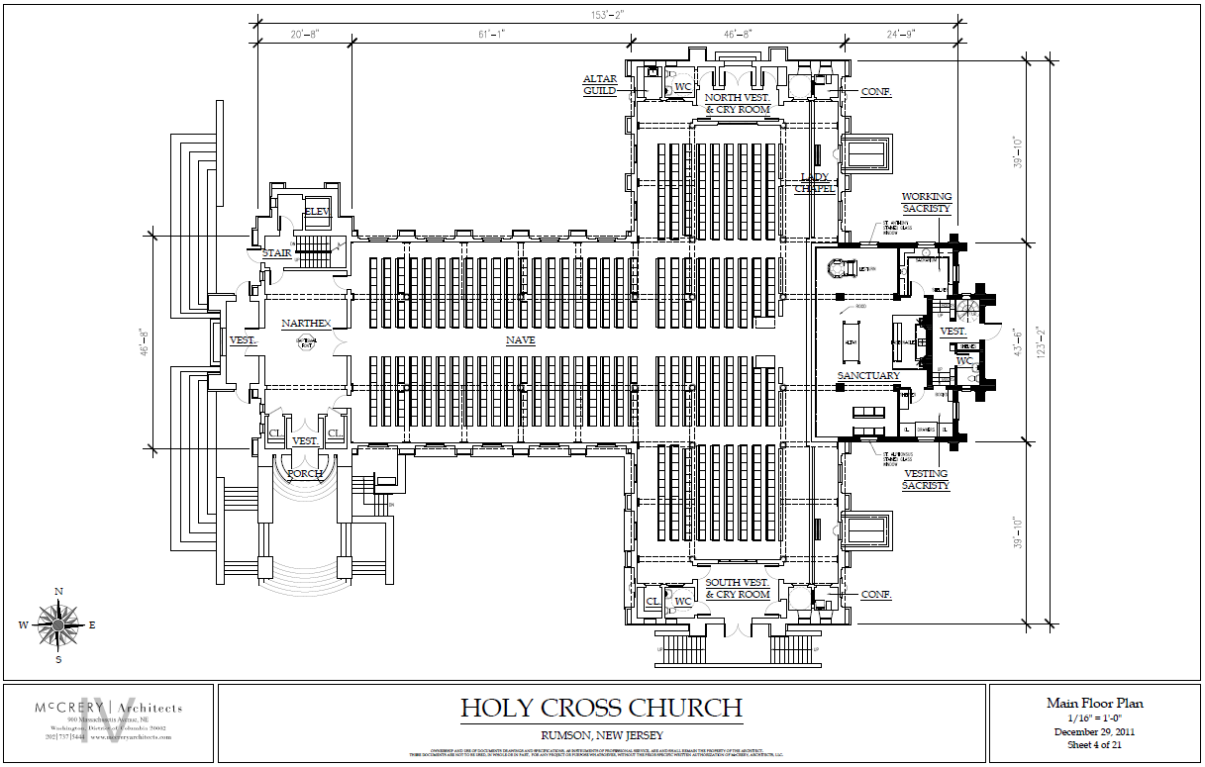

Figure 3.1: A diagram showing a schematic of a church building. Credit: McCrery Architects.

The proposed method will avoid storing all the empty space that would be in the bounding box. This saves memory. It also saves the computational expense of performing a stencil on these areas.

In addition, the proposed method also has better data locality. In a conventional 3D array, neighbouring points in space can be far away in memory. This means that to fetch a grid point's

six neighbouring elements, it is necessary to perform separate data fetches. The proposed method caches a relatively small contiguous block of around 512 points. This one large read can make use of high bandwidth memory's long fetch lengths. This is particularly important as there is a trend in GPUs towards higher bandwidth graphics memory.

Chapter 4

Design and Implementation

> *"The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time."*
>
> Tom Cargill

The software was written in CUDA C. This is a proprietary language developed by NVIDIA and is an extension to ANSI C that allows for GPGPU programming. The CUDA C language was selected because the University of Edinburgh Acoustics and Audio Group (AAG) has an NVIDIA K20 GPU server. CUDA is a well supported way to program these devices, although the resulting programs will only run on NVIDIA hardware. Had OpenCL been used, the code would have also been able to run on graphics cards produced by AMD, who are NVIDIA's main competitor [17]. This paradigm is generally considered more complex than CUDA C, and, as the AAG does not own any AMD hardware, was not pursued. Some discussion of how future work might involve OpenCL is given in chapter 6.

The work stems from a Matlab script that implements the scheme using a bounding box as described in 2.4. There were also CUDA kernels from [1] that served as inspiration, but these did not model cases where non-cuboid rooms were used. This meant that they were of limited use. As no code exists that implements the semi-structured approach proposed here, the project was therefore largely one of software development from scratch. Altogether the source base extends to over 1500 lines of code. With hindsight the project was probably too ambitious in terms of software development, meaning that not enough time was left for experimentation and analysis.

4.1 Design Overview

The code was designed to be as extensible and reusable as possible. As such each experiment is divided into a separate directory.

A directory tree (the output of the GNU `tree` utility) of all the new source files is given in listing D.1.

The `lib` directory contains source files with code that is necessary for the implementation of the semi-structured method. They contain functions for forming the data structure as described in chapter 3, as well as generating room shapes, and performing the FDTD stencil.

There are source directories for the memory capacity experiment and the stencilling experiment. These each contain a CUDA C source file that performs the required experiment as well as a makefile that compiles it and the necessary code from the `lib` dir. These then call functions from the library that do all the tasks associated with the proposed method.

There is then a `test` directory. This contains functions that test the library functions for correctness.

As well as these source directories there is also a directory for gathering and storing results. This contains a script that measures the memory capacity performance with rooms as described in 5.2. These results are automatically written to 3 CSV files, which can be read by LaTeX to form the three tables in appendix A. This automation means that this experiment is easily repeatable.

4.2 Software Development Process

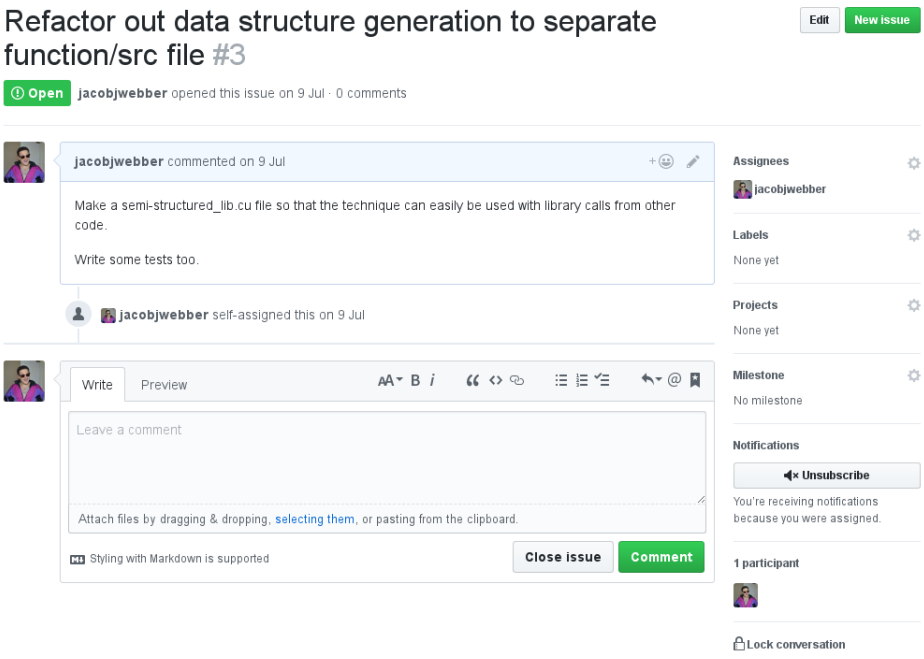

Figure 4.1: A screenshot of a GitHub Issue.

At the root of all development was GitHub and the git tool. These were used to store code and data, and distribute it between the machines that were used for development. Accurately tracking changes that were made to this document and to code across multiple computers was made trivial. GitHub also has *Issues*. These can be used to record work that needs to be done and can be marked as complete, as well as linked to specific commits. A screenshot is included in figure 4.1.

The software development was largely done from scratch by one developer. Although this removed the absolute necessity of a rigorous development process, this does not mean that a more thorough process would not have been beneficial. GitHub issues were used occasionally, but development was largely organised on an ad hoc basis. This could be improved upon.

4.3 Debugging and Profiling Tools

NVIDIA includes developer tools with its CUDA software. As well as the compiler, NVCC, there is a debugger, an IDE and some profiling tools.

For debugging, the `cuda-gdb` program functions like gdb, but is extended to work on GPU code. The `cuda-memcheck` tool checks for memory errors such as segmentation faults. It will provide the line of code where the memory error is when the the binary is compiled using debugging flags.

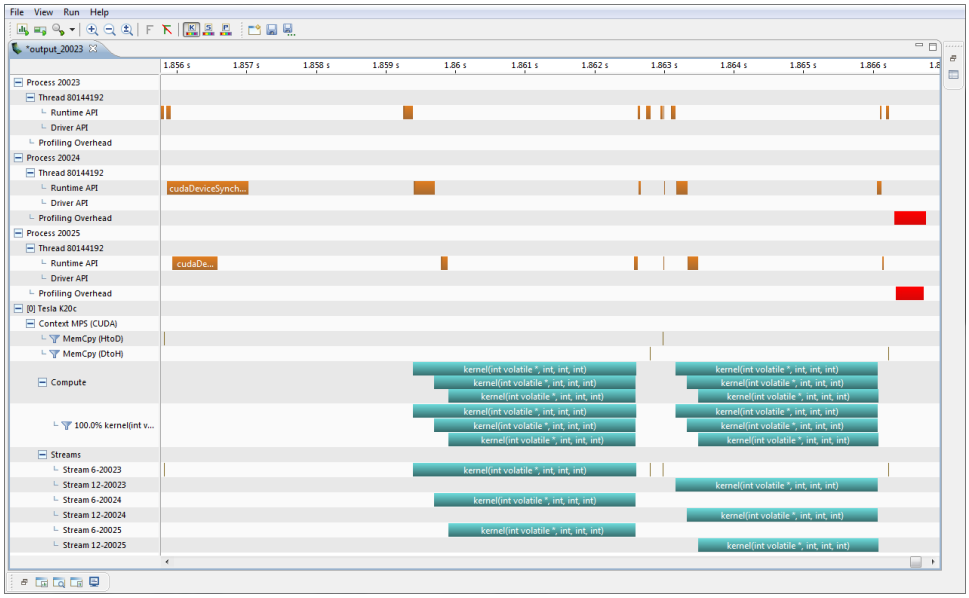

Figure 4.2: A screenshot of the visual profiler tool from NVIDIA's own documentation.

For profiling there is the `nvprof` tool for the command line. There is also the NVIDIA *Visual Profiler*, which provides a GUI. This can provide details of the amount of time spent on each part of the computation, the time taken for each kernel, as well as other details such as the

theoretical and achieved occupancy (see section 2.3.1). Figure 4.2 provides a screenshot of this tool in use. The tool provides a time line of kernel executions as well as data on the individual kernel performance such as runtime, theoretical and achieved occupancy, and global read efficiency.

The `nsight` IDE, based on eclipse, provides access to the visual profiler as well as a GUI based debugger and all the typical features of a modern IDE.

It was unfortunately not possible to use the GUI based tools when working on the K20 GPU server that was used to gather results. This is because the server is not configured to run graphical programs. It was possible to run these tools on a home desktop with an NVIDIA GTX 970 graphics card installed. This was useful for debugging as this device is capable of running CUDA code. However, it was of limited use for profiling, as the card achieved very different performance to the K20. The GTX 970 is a commodity card that lacks the double precision floating point units of the K20. It also has fewer register files and less L1 cache.

Somewhat ironically, the debugger built into the `nsight` tool was itself riddled with bugs - causing it to crash every time it reached the end of a program. More luck was had with the `cuda-gdb` command line utility that NVIDIA also provides. It was also possible to use this on the K20 server through SSH - meaning that the same machine that was used for gathering results could also be used for development, debugging and profiling.

Chapter 5

Results and Analysis

5.1 Correctness

The correctness was largely tested by a collection of functions in the `test.cu` source file. One of these checks the translation from a bounded box representation to a semi-structured one. It does this by looping through every point in the grid, checking where it is translated to in an AOB representation, then checking the $K_{l,m,p}$ is correct. As this checks every point the fact that it passes shows the method is correct and therefore that the memory usage results are valid.

Confirming the stencil was more complex. For this the Matlab script that was used as a starting point was run for a very simple example. The results from this were compared to the output from the examples run here. This showed that the new method gives correct results.

Another possible way to check the correctness would be to find analytical solutions to the wave equation inside of a simple cube example. This could then be compared with the output. Unfortunately there was not time to implement this so it is left as further work. A thorough analysis on the accuracy of FDTD approaches to acoustics generally and the results achieved here in particular would be very worthwhile.

5.2 Memory Usage

Very high memory usage is currently the biggest inhibiting factor to the performance of FDTD simulations.

5.2.1 Sphere

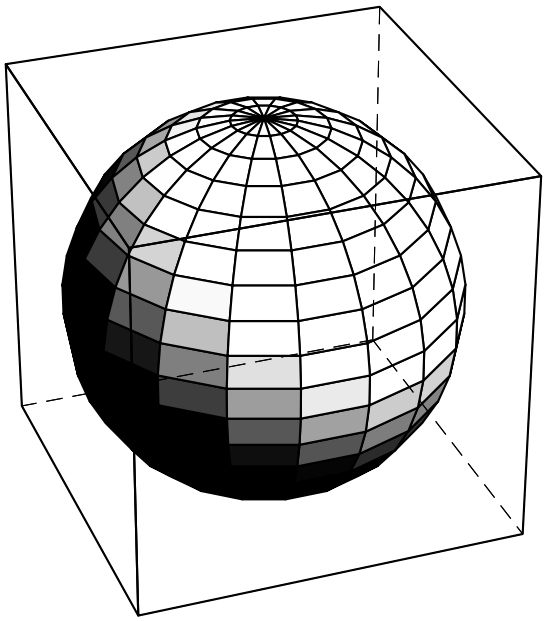

Figure 5.1: Computer graphic showing a sphere of unit diameter within a cube of unit side.

To determine if the efficiencies as theorised in section 3.3 were realised, the example of a spherically shaped room was used, as shown in figure 5.1. While the existence of such a shaped room seems far-fetched, the example illustrates some of the benefits of the new data structure even in somewhat unfavourable conditions. This configuration tests the performance of the new struct based method when a curved room shape is chosen. This curvature will necessarily result in partially empty block arrays being stored within structs at the boundary. The arrangement is also not particularly sparse. The volume of the sphere V_s is given by:

$$V_s = \frac{\pi d^3}{6} \tag{5.1}$$

meaning that that the fraction of points in the cube that are also inside the room is:

$$\frac{\pi}{6} \approx 0.5236... \tag{5.2}$$

This means that under the existing bounding box technique for modelling non-cuboid rooms, 47.6 % of points stored in the grid will be stored unnecessary. The amount by which this is reduced depends on ratio between the dimensions of the sub-array within each struct (shown as Bx, By and Bz in listing 3.1) and the dimensions of the bounding box array, X, Y and Z. A simple experiment was done with the sphere example to measure this effect. The values of X, Y and Z are set as equal and can be thought of as the diameter of the sphere measured in grid

points. A sub-array size of 8 by 8 by 8 was selected for ease of arithmetic. Spherical rooms were then generated with varying diameter sizes and converted to a semi-structured array of structs representation. The size of this array in memory was then compared with that yielded from the bounding box method. The result is shown in figure 5.2. Full results are included in appendix A.

When the size of the sub-array is equal to that of the bounding array, the semi-structured and structured approaches can be seen as equivalent. However, as shown in listing 3.1 there is some overhead of storing the array indices of the neighbouring structs. This requires the storage of 6 integers for every struct, meaning that for a room of size 8 by 8 by 8 and with double precision floating point numbers, the semi-structured approach uses $1.002\,757$ times the memory of the structured approach, in line with the worst case scenario presented in section 3.1. This overhead is marginally higher when single precision numbers are used. The storage of these integers is still a very small overhead.

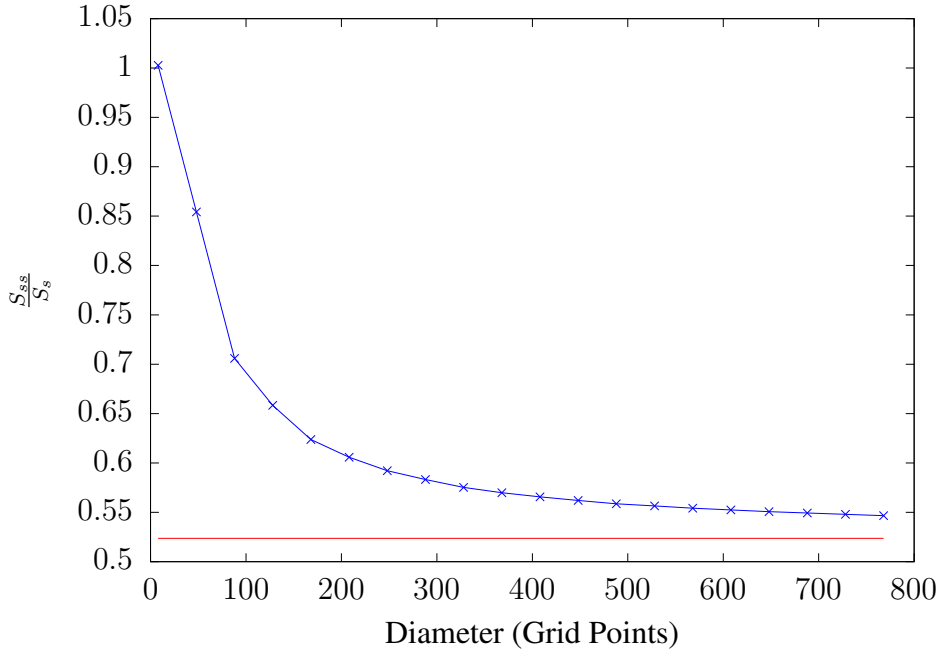

Figure 5.2: The fraction of memory used by the semi-structured approach with respect to the structured approach (double precision). The ideal fraction of $\frac{\pi}{6}$ is shown as a dashed line

The results shown in figure 5.2 show that as the diameter becomes larger, the storage efficiency of the semi-structured method increases. It is easy to see that should the diameter, d, become very large with respect to Bx, By and Bz, such that

$$\mathrm{Bx}, \mathrm{By}, \mathrm{Bz} \ll d \tag{5.3}$$

then the case is equivalent to the infinite sum of infinitesimals making up a spherical volume integral in Cartesian coordinates. This means that

$$\lim_{d\to\infty} \frac{S_{ss}}{S_s} = \frac{\pi}{6} \cdot 1.002757 \tag{5.4}$$

Practical gains come long before d approaches infinity. With a diameter of 768 points, the semi-structured approach means that $4014\,\mathrm{MiB}$ of global memory is required, which falls within the 5GiB limit of the K20. The bounding box method for a room of this size would require $7344\,\mathrm{MiB}$ - meaning such a simulation could not be run on on a K20 graphics card.

5.2.2 Cube

The results from the cube shaped room experiment were very simple, as far as memory capacity goes. In all cases $\frac{S_{ss}}{S_s}$ was $1.002\,757$. This is because there is no empty space stored with the bounding box method. This represents the worst case performance for the semi-structured method. As such an increase in memory usage of around a quarter of one percent is a fairly tolerable result. Full results are included in appendix A.

5.2.3 Cross

A very simple cross shape as shown in figure 5.3 was modelled in various sizes. Like the cube example all sizes of room yielded the same performance relative to the bounding box method.

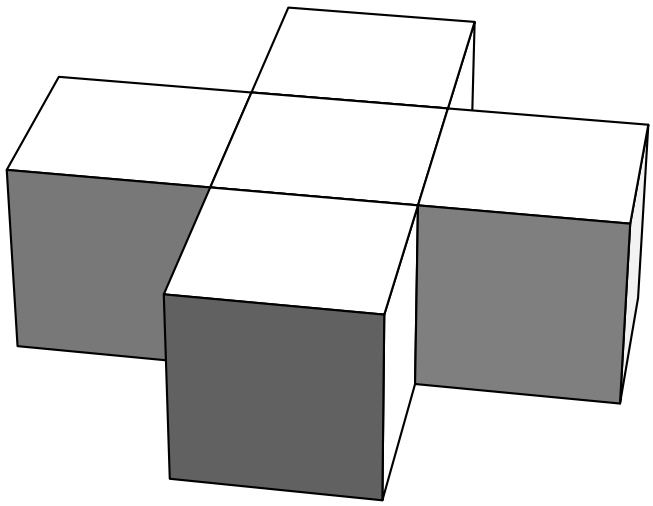

Figure 5.3: A figure showing simple cross shape. The cross is made up of 5 equal cubes.

The diagram in figure 5.3 shows that the cross was made of five cubes. To store this in a bounding box the volume of nine cubes would be needed. This means that an ideal representation would have a ratio:

$$\frac{S_{ss}}{S_s} = \frac{5}{9} \tag{5.5}$$

However, the semi-structured approach necessitates the storage of neighbouring blocks. Using equation 3.2 it can be shown that the expected storage space required on this shape would be

$$\frac{S_{ss}}{S_s} = \frac{5}{9} \cdot \left(1 + \frac{24}{17 \cdot \mathrm{Bx} \cdot \mathrm{By} \cdot \mathrm{Bz}}\right) \tag{5.6}$$

As with the other memory capacity experiments, the Bx, By and Bz values were all set to 8. This means that the estimated performance would be:

$$\frac{5}{9} \cdot \left(1 + \frac{24}{17 \cdot 8 \cdot 8 \cdot 8}\right) = 0.5570874183 \tag{5.7}$$

This is exactly the performance that was achieved in all cases when the experiment was run on a variety of sizes of cross shaped rooms. Full results for this experiment are given in appendix A.

5.2.4 Quantifying Improvements

To quantify the improvement, ideal performance is taken to be when no points that are outside the room are stored. This is found by multiplying the number of points that are inside the room by the size of two double precision floating point numbers and one char (the char is used to store a K value).

With the 768 point diameter sphere example this would mean that ideal performance would be $3845\,\mathrm{MiB}$ of memory usage. This means that $168.7\,\mathrm{MiB}$ more data than ideal performance is used with the semi-structured approach and $3499\,\mathrm{MiB}$ for the structured. This shows a reduction in unnecessary data storage by a factor of 20.7.

Room Shape	Diam	Ideal (MiB)	Achieved minus Ideal (MiB)		Improvement
			Structured	Semi-structured	
Cube	768	7344	0	20.25	N/A
Sphere	768	3845	3499	168.7	20.7X
Cross	488	9221	7736	226.4	34.2X

Table 5.1: Quantifying the memory capacity improvements by comparing unecessary data stored between both methods.

The cross example shows even greater reductions as its shape is well suited to cubic decompositions. Taking example made from 488 point diameter cubes, ideal performance was theorised as taking $9221\,\mathrm{MiB}$. This means $7736\,\mathrm{MiB}$ of extra data was stored with the structured approach compared with $226.4\,\mathrm{MiB}$ for the semi-structured. This is a reduction in additional data storage by a factor of 34.2.

These performances would be even more improved with rooms that populated a bounding box more sparsely - for example a cross with longer limbs. Such a shape was not trialled for fear of being unrealistic and therefore over claiming this method's effectiveness.

Table 5.1 shows the full results of surplus data storage. Note that with the structured method no excess data is stored in the cube example.

5.3 Stencil Performance

Reducing the memory consumption of these models is pleasing, but of little use if stencil performance does not stay at a reasonable level. If performance were significantly worse then it would not be worth using a GPU at all, and CPU based memory is cheap and plentiful enough that these optimizations are unlikely to be worthwhile.

Initial experiments were run to find the ideal block dimensions (shown as Bx, By and Bz in listing 3.1). Best performance was achieved when all were set to 8.

The stencil was run on the three rooms described in section 5.2. Results in table 5.2 show the time taken to run 1000 time steps. The *Diam* figure is the width in grid points for the cube and sphere and the width of each of the five cubes in the cross.

Room Shape	Diam	Memory Use (MiB)		Time (s)		Speed-up
		Structured	Semi-structured	Structured	Semi-structured	
Cube	608	3643.84	3653.90	79.49	81.25	0.98X
Sphere	608	3643.84	2012.92	79.50	44.78	1.78X
Cross	256	2448.00	1363.76	53.27	30.34	1.76X

Table 5.2: Performance results for the stencilling operation.

The results were obtained by running the experiments multiple times and taking a mean of the time taken for each experiment. The timing results were consistent - with a standard deviation value of less than one percent in all cases.

The performance figures were very similar between the two methods in the cube example. Performances scaled relative to memory savings for the other two examples. To illustrate this point, table 5.3 is included which shows in $\mathrm{GiB\,s^{-1}}$ the rate at which data is processed. To get this result the memory usage from table 5.2 was multiplied by the number of time steps (1000) and then divided by the time taken. These results show that the rate at which data is processed is relatively constant within each method. The structured approach was, on average, 2% faster than the semi-structured on a per GiB basis. However, this is easily overshadowed by the data savings that are possible if the room is even slightly irregularly shaped.

Room Shape	Data Rate ($\mathrm{GiB\,s^{-1}}$)	
	Structured	Semi-structured
Cube	44.77	43.92
Sphere	44.76	43.90
Cross	44.88	43.90
Mean	44.80	43.91

Table 5.3: Data processing rates.

The data processing rates are well below the $208\,\mathrm{GB\,s^{-1}}$ maximum memory bandwidth of the K20 GPU. This suggests there is still significant room for improvement - particularly as the maximum theoretical processing rate could be significantly higher than this if cache is well used.

5.4 Comparison with Results Achieved in [1]

Results for the structured approach should match those as achieved in [1] for a cube shaped room. However, because the method allows irregularly shaped rooms, it is necessary to store a K array of the rooms shape. This results in an extra global memory read for each point processed. In addition, slightly more complex boundary conditions are imposed in the work described in this report. In the literature, *voxels* are often used when measuring performance. A voxel is one point in a 3D grid and $\mathrm{voxs^{-1}}$ is the number of points processed per second. With the 3D tiling method that was replicated here, [1] achieved 4484 $\mathrm{Mvoxs^{-1}}$. In contrast a rate of 2827 $\mathrm{Mvoxs^{-1}}$ was achieved for the cube example here. This is roughly in line with what would be expected, given that there are 50% more global memory reads per point and significantly more floating point operations.

Chapter 6

Conclusions and Further Work

"It is not incumbent upon you to finish the task. Yet, you are not free to desist from it."

Tarphon

To conclude, the current state-of-the-art method was implemented on a GPU with a slightly more sophisticated set of boundary conditions than had previously been done in [1]. In addition, a new technique that represents a not insignificant step forwards in its field was both devised and implemented.

In terms of memory capacity, the results from experiment match the theory exactly and give significant improvement. This was the case even though, to avoid over-claiming, particular effort was made to avoid choosing rooms that were overly suited to the new method. In addition, the stencilling time performance was shown to improve almost as much as the memory capacity. It was also suggested that the proposed technique might be better suited to more modern graphics cards than the device that was used to generate the results in this report. Unfortunately, such cards were not available for these experiments.

The project was largely one of devising and implementing a complex piece of new software. As such, less time was available than would have been ideal for analysing and optimising the resulting code. However, this means there may well be easy optimisations that unlock some of the theoretical advantages of the novel approach.

In terms of further work, of particular interest and necessity would be verifying the results in a more thorough fashion. While this is not straightforward, there are ways to do so. One way would be to solve the wave equation analytically for a simple enough case and then compare the answer with the results of the FDTD method used here. Another way would be to compare the output of a real room with that of one simulated.

Another piece of further work would be to conduct more accurate profiling of the code - collecting data such as theoretical and achieved occupancy, and global read efficiency. This would be useful in further optimising the code. If the work were ported to OpenCL then it would allow the code to run on AMD devices. These currently have the highest bandwidth memory.

This could be used to verify the unproven claim that this new method will outperform the old to a larger extent on this hardware.

A more acoustics-orientated further project might describe the derivation of the new boundary conditions and some of the finer points of the theoretical acoustics background. However, the subject matter would be that of cutting edge physics. This was deemed to fall outside the scope of this report - which is meant to focus on high performance computing.

Bibliography

[1] Webb, C.J. (2009) *Parallel computation techniques for virtual acoustics and physical modelling synthesis.* PhD thesis. University of Edinburgh. Available at: http://www2.ph.ed.ac.uk/~cwebb2/Site/Docs/CJWebb_thesis_FINAL.pdf (Accessed: 5 April 2017)

[2] Hamilton, B. (2016) *Finite Difference and Finite Volume Methods for Wave-based Modelling of Room Acoustics.* PhD thesis. University of Edinburgh. Available at: http://www2.ph.ed.ac.uk/~s1164563/thesis/hamilton2016phdthesis.pdf (Accessed: 5 April 2017)

[3] Richard Feynman, 1969, *Lectures in Physics, Volume 1,* Addison Publishing Company, Addison

[4] NVIDIA Corporation *What's the Difference Between a CPU and a GPU?* https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ retrieved August 18, 2017.

[5] NVIDIA Corporation *NVIDIA Special Event: The GeForce GTX 1080 and Pascal (Part 4)* https://www.youtube.com/watch?v=0xnzwjPyx8A retrieved August 18, 2017.

[6] NVIDIA Corporation *CUDA C Programming Guide* http://docs.nvidia.com/cuda/cuda-c-programming-guide retrieved August 18, 2017.

[7] NVIDIA Corporation *NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110/210* http://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf retrieved August 18, 2017.

[8] AMD *FirePro S9300 x2 Server GPU* http://www.amd.com/en-us/products/graphics/server/s9300-x2 retrieved August 18, 2017.

[9] P. Micikevicius, *3D finite difference computation on GPUs using CUDA*. Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2, New York, New York, USA, 2009, pp. 79-84, ACM Press.

[10] K. Hou, Y. Zhao, J. Huang, and L. Zhang, *Performance evaluation of the Three-Dimensional Finite-Difference Time-Domain(FDTD) method on Fermi architecture*

GPUs in Algorithms and Architectures for Parallel Processing vol. 7016 of Lecture Notes in Computer Science, pp. 460-469. Springer Berlin / Heidelberg, 2011.

[11] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, Wadsworth and Brooks/Cole Advanced Books and Software, Pacific Grove, California, 1989.

[12] Harris, Frederic j. (Aug 2006). *Multirate Signal Processing for Communication Systems*. Upper Saddle River, NJ: Prentice Hall PTR. ISBN 0-13-146511-2

[13] Andreas Schäfer, Dietmar Fey, *High Performance Stencil Code Algorithms for GPGPUs*, Procedia Computer Science, Volume 4, 2011, Pages 2027-2036, ISSN 1877-0509, `http://dx.doi.org/10.1016/j.procs.2011.04.221.` retrieved August 18, 2017

[14] D. W. Zingy , H. Lomax , and H. Jurgens , *High-accuracy finite-difference schemes for linear wave propagation*, SIAM J. Sci. Comput. 17, 328346, 1996.

[15] L. Savioja and U. P. Svensson, *Overview of geometrical room acoustic modeling techniques*, Journal of the Acoustical Society of America, vol. 138, no. 2, pp. 708730, 2015.

[16] Shapiro, L. G. and Stockman, G. C: *Computer Vision*, page 137, 150. Prentice Hall, 2001

[17] Tech Power Up *Graphics Card Market Up Sequentially in Q3, NVIDIA Gains as AMD Slips* `https://www.techpowerup.com/194979/graphics-card-market-up-sequentially-in-q3-nvidia/-gains-as-amd-slips` retrieved August 18, 2017.

Appendix A

Memory Capacity Experiment Results

The results are for double precision numbers.

Diam	Internal Blocks	Size semi-Structured (MiB)	Size - Structured (MiB)	Ratio
8	1	0.008324	0.008301	1.002757
48	184	1.531555	1.792969	0.854201
88	937	7.799278	11.048340	0.705923
128	2689	22.382347	34.000000	0.658304
168	5761	47.952660	76.873535	0.623786
208	10618	88.380722	145.894531	0.605785
248	17590	146.413345	247.288574	0.592075
288	27133	225.846130	387.281250	0.583158
328	39541	329.126221	572.098145	0.575297
368	55309	460.373840	807.964844	0.569794
408	74823	622.801941	1101.106934	0.565614
448	98430	819.298767	1457.750000	0.562030
488	126459	1052.602905	1884.119629	0.558671
528	159528	1327.858398	2386.441406	0.556418
568	197805	1646.463379	2970.940918	0.554189
608	241830	2012.912964	3643.843750	0.552415
648	291849	2429.254639	4411.375488	0.550680
688	348446	2900.349365	5279.761719	0.549333
728	411749	3427.262451	6255.228027	0.547904
768	482264	4014.206055	7344.000000	0.546597

Table A.1: Sphere memory capacity results

Diam	Internal Blocks	Size semi-Structured (MiB)	Size - Structured (MiB)	Ratio
8	1	0.008324	0.008301	1.002757
48	216	1.797913	1.792969	1.002757
88	1331	11.078804	11.048340	1.002757
128	4096	34.093750	34.000000	1.002757
168	9261	77.085503	76.873535	1.002757
208	17576	146.296814	145.894531	1.002757
248	29791	247.970428	247.288574	1.002757
288	46656	388.349121	387.281250	1.002757
328	68921	573.675598	572.098145	1.002757
368	97336	810.192688	807.964844	1.002757
408	132651	1104.143066	1101.106934	1.002757
448	175616	1461.769531	1457.750000	1.002757
488	226981	1889.314819	1884.119629	1.002757
528	287496	2393.021729	2386.441406	1.002757
568	357911	2979.132812	2970.940918	1.002757
608	438976	3653.891113	3643.843750	1.002757
648	531441	4423.539062	4411.375488	1.002757
688	636056	5294.319824	5279.761719	1.002757
728	753571	6272.476074	6255.228027	1.002757
768	884736	7364.250000	7344.000000	1.002757

Table A.2: Cube memory capacity results

Diam	Internal Blocks	Size semi-Structured (MiB)	Size - Structured (MiB)	Ratio
8	5	0.041618	0.074707	0.557087
48	1080	8.989563	16.136719	0.557087
88	6655	55.394020	99.435059	0.557087
128	20480	170.468750	306.000000	0.557087
168	46305	385.427521	691.861816	0.557087
208	87880	731.484070	1313.050781	0.557087
248	148955	1239.852173	2225.597168	0.557087
288	233280	1941.745605	3485.531250	0.557087
328	344605	2868.378174	5148.883301	0.557087
368	486680	4050.963379	7271.683594	0.557087
408	663255	5520.715332	9909.962891	0.557087
448	878080	7308.847656	13119.750000	0.557087
488	1134905	9446.574219	16957.076172	0.557087

Table A.3: Cross memory capacity results. Diam here gives the diameter of the five constituent cubes of the cross.

Appendix B

NVIDIA K20 Device Details

Listing B.1: Output from deviceQuery program

```
CUDA Driver Version / Runtime Version          8.0 / 7.0
CUDA Capability Major/Minor version number:    3.5
Total amount of global memory:                 5062 MBytes (5307367424
    bytes)
(13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
GPU Max Clock rate:                            706 MHz (0.71 GHz)
Memory Clock rate:                             2600 Mhz
Memory Bus Width:                              320-bit
L2 Cache Size:                                 1310720 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,
    65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048
    layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
```

Appendix C

Kernels

Listing C.1: The semi-structured stencil CUDA kernel.

```
__global__ void perform_stencil(struct block *aos, real l2, real l, real g)
    {
    //Launch with blocksize threads in each dimension.
    //l is courant number. l2 is this squared. g is wall permittivity.

    int x = threadIdx.x;
    int y = threadIdx.y;
    int z = threadIdx.z;

    int bl = blockIdx.x + 1;

    char k = aos[bl].k[z][y][x];

    int left, right, fore, aft, up, down;

    left = aos[bl].left;
    right = aos[bl].right;
    up = aos[bl].up;
    down = aos[bl].down;
    fore = aos[bl].fore;
    aft = aos[bl].aft;

    __shared__ real u1_s[Bz+2][By+2][Bx+2]; //u1 in shared memory.

   __syncthreads();

    //Read in working block
    u1_s[z+1][y+1][x+1] = aos[bl].u1[z][y][x];

    //Read in neighbours.
    if ( x == 0 ) {
        u1_s[z+1][y+1][0] = aos[left].u1[z][y][Bx-1];
    } if( x == Bx-1 ) {
        u1_s[z+1][y+1][Bx+1] = aos[right].u1[z][y][0];
    }
```

```
    if ( y == 0 ) {
        u1_s[z+1][0][x+1] = aos[aft].u1[z][By-1][x];
    } if( y == By-1 ) {
        u1_s[z+1][By+1][x+1] = aos[fore].u1[z][0][x];
    }

    if ( z == 0 ) {
        u1_s[0][y+1][x+1] = aos[down].u1[Bz-1][y][x];
    } if( z == Bz-1 ) {
        u1_s[Bz+1][y+1][x+1] = aos[up].u1[0][y][x];
    }

    __syncthreads();

    x++;y++;z++;

    //Main stencil
    aos[bl].u[z-1][y-1][x-1] = ((2.0 - 12 * (real) k) * u1_s[z][y][x] +
                                                  12 * ( u1_s[z][y][x+1] +
                                                         u1_s[z][y][x-1] +
                                                         u1_s[z][y+1][x] +
                                                         u1_s[z][y-1][x] +
                                                         u1_s[z+1][y][x] +
                                                         u1_s[z-1][y][x])+
                                (0.5 * l * g * (6.0 - (real) k) - 1.0) *
                aos[bl].u[z-1][y-1][x-1])/(1 + 0.5 * l * g * (6 - k));

    __syncthreads();
    if (k == 0) {
        aos[bl].u[z-1][y-1][x-1] = 0.0;
    }

}
```

Listing C.2: The structured stencil CUDA kernel.

```
__global__ void perform_stencil_structured(real* u, real* u1,
                                           char* k_d, real l2, real l,
                                          real g, int X, int Y, int Z) {

    // get x,y,z from thread and block Ids
    int x = blockIdx.x * Bz + threadIdx.x;
    int y = blockIdx.y * By + threadIdx.y;
    int z = blockIdx.z * Bx + threadIdx.z;

    // Test that not at boundary
    if( (x>0) && (x<(X-1))
            && (y>0) && (y<(Y-1))
            && (z>0) && (z<(Z-1)))
    {
        // get linear position
        int cp = z*X*Y+y*X+x;
        char k = k_d[cp];
        u[cp] = ((2 - l2 * k) * u1[cp] +
            l2*(u1[cp-1]+u1[cp+1]+u1[cp-X]+u1[cp+X]+u1[cp-Y*X]+u1[cp+Y*X])+
            (0.5*l*g*(6-k)-1)*u[cp])/(1+0.5*l*g*(6-k));
    }
}
```

Appendix D

Source Directory Structure

Listing D.1: Output from the GNU tree program

```
|-- lib
|   |-- block_dims.h
|   |-- make_rooms.cu
|   |-- make_rooms.h
|   |-- semi-structured_lib.cu
|   |-- semi-structured_lib.h
|   |-- stencils.cu
|   `-- stencils.h
|-- memory-capacity
|   |-- Makefile
|   |-- memory-capacity
|   |-- memory-capacity.cu
|   `-- obj-
|-- results
|   |-- data
|   |   |-- memory-cap_cube.csv
|   |   |-- memory-cap_graph
|   |   |-- memory-cap_raw
|   |   `-- memory-cap_sphere.csv
|   `-- scripts
|       |-- make_mem-cap_graph.sh
|       `-- mem-cap_graph.gp
|-- sphere
|   |-- Makefile
|   |-- obj-
|   |-- sphere
|   `-- sphere.cu
|-- stencil-performance
|   |-- Makefile
|   |-- obj-
|   |-- stencil-performance
|   `-- stencil-performance.cu
`-- test
    |-- Makefile
    |-- obj-
    |-- test
    `-- test.cu
```