The application of parallel processing techniques to computationally intensive biomedical imaging studies

Blake Andrew McLuckie

A dissertation submitted to the Faculty of Engineering and the Built Environment,
University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, 2013

Declaration

I declare that this dissertation is my own, unaided work, other than where specifically acknowledged. It is being submitted for the degree of Master of Science in Engineering at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.

Signed this _____ day of_____ 2013

_____

Blake Andrew McLuckie

Abstract

The landscape of modern computing is changing. While Moore's law is currently holding and the number of transistors that can be produced on a given area of a chip is still growing exponentially, the practice of improving performance by increasing the clock frequency of a single processor is reaching its limit. Instead, the focus has shifted to applying multiple processors to solving a single problem, a methodology known as parallel processing. Parallel processing has the potential to overcome many of the shortcomings of linear processing, but also presents a number of unique challenges. This dissertation explores the potential benefits of parallel processing by examining the application of a near-field coded aperture simulator on a parallel cluster and contrasting its implementation and performance with previously written simulators for serial processors. The platform used is a cluster of Sony PlayStation 3's; featuring the IBM developed Cell Broadband Engine Architecture. It was found that the PS3's were capable of producing performance gains of around forty times an equivalently priced conventional processor, with the capability of easily scaling the system by adding or removing nodes as required. However, this comes at the cost of a much increased burden on the developer. Apart from the core application, a great deal of code must be written to handle communication and synchronization between nodes, a task which can at times be very complex. In addition, a number of tools available for serial processors, such as highly efficient compilers, advanced development environments and many standardized libraries cannot be applied in a parallel environment. The main conclusions drawn from this research are that while the potential gains of parallel processing are enormous, allowing attainable solutions to problems that were previously too costly, the costs of development are prohibitive. Still, parallel processing is the natural next step in modern computing, and it is only a matter of time before its idiosyncrasies are solved.

Acknowledgments

This work was carried out in the Biomedical Engineering Research Group, within the School of Electrical and Information Engineering at the University of the Witwatersrand, Johannesburg. Funding was provided partially by the School, and partially by the National Research Foundation of South Africa.

The author would like to thank:

Professor David Rubin as research supervisor, for the initial conception of the work and for his patience, guidance, and insight.

Professor Tshilidzi Marwala as research supervisor and for the co-ordination of funds.

Dr. Ken Nixon, for answering so many questions.

David Starfield, for helping with the details.

Ian and Bev McLuckie for their ongoing emotional, intellectual, and, yes, financial support.

Kerry McLuckie for listening, and for the pizza.

Professor Ian Jandrell, Gill van der Heever and the rest of the staff in the School of Electrical and Information Engineering, for making it a place that genuinely cared about its students.

Table of Contents

List of Figures

List of Tables

Table Of Acronyms

BEI	Broadband Engine Interface
CBE	Cell Broadband Engine
CBEA	Cell Broadband Engine Architecture
DMA	Direct Memory Access
EA	Effective Address
EIB	Element Interconnect Bus
ENIAC	Electronic Numeric Integrator and Computer
ES	Effective Storage
FLOPS	Floating Operations Per Second
GUI	Graphical User Interface
IOIF	Input/Output Interface
LA	Local Address
LS	Local Storage
MFC	Memory Flow Controller
MIC	Memory Interface Controller
MIMD	Multiple Instructions Multiple Data
MISD	Multiple Instructions Single Data
MPI	Message Passing Interface
MPP	Massively Parallel Computing
OS	Operating System
PPE	Power Processing Element
PPU	Power Processing Unit
SDK	Software Development Kit
SETI	Search for Extra Terrestrial Intelligence
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SPE	Synergistic Processing Element
SPU	Synergistic Processing Unit
STI	Sony Computer Entertainment, the Toshiba corporation and IBM.

2 Introduction

There can be no doubt that computers have become an invaluable tool in nearly every application in existence; education, media, warfare, architecture, communication; no facet of life is unaffected and bio-medical engineering is no exception. From the complex calculations required in MRI's to the design of prosthetic limbs, computers are an essential asset to the field.

2.1 The Problem

The physical world is often a great deal more complex than simple observation would imply, and simulating it can also be similarly complex. The problem is Consider the following: There is an estimated 1.37 billion km^3 of water in the oceans of the world [1]. This translates to $1.37^9 \times 10^6 = 1.37 \times 10^{15}$ mL of water. Now, a standard teaspoon has a volume of 5 ml. Dividing the volume of the oceans by the volume of the teaspoon:

$$\frac{1.37 \times 10^{15}}{5} = 2.74 \times 10^{14}$$

Similarly, given the molar mass of water and Avogadro's constant [2], it can be shown that there are approximately 1.668 x 10^{22} molecules of water in a single tea spoon.

Dividing this number by the number of teaspoons of water in the ocean gives

$$\frac{1.668 \times 10^{22}}{2.74 \times 10^{14}} = 6.087 \times 10^{7}$$

There are more than 60 million times more molecules of water in a teaspoon than there are teaspoons of water in all the oceans in the world. Performing a complete simulation of all the interactions of all the molecules in a teaspoon – or even a single drop - of water is a completely infeasible task.

And yet, a number of methods of accurately simulating the behaviour of a fluid exist, including Smoothed Particle Hydrodynamics methods(SPH) [3], vorticity based methods [4], and Lattice Boltzmann methods [5]. All have which have been adopted by graphical developers to produce entire rivers and waterfalls in modern movies and video games. This is accomplished not by modelling individual particles, but rather the *emergent* behaviour of the sum of those particles. Of course, there is a trade-off, each of the listed methods is an approximation of fluid behaviour, and none them are going to be as accurate as an effective particle by particle simulation. Another example of this is Newtonian physics. Theories on relativity and quantum states provide a more exact description of motion and are necessary for systems that require a great deal of precision, such as the Global Positioning System [6], but their mathematical complexity is

prohibitive. Newton's Laws are a great deal simpler and are accurate enough for general use. Even in standard GPS navigation systems, the error that would be caused by ignoring relativistic effects would amount to less than a centimetre [6].

The question then becomes finding the balance; a method which produces results accurate enough to be meaningful, but cheap enough computationally to be completed in a reasonable amount of time and effort.

In a number of areas, this balance has already been found. After all, computer generated rivers and oceans can be very convincing indeed. However, the landscape of computational methods is changing and the age of serial processing is coming to an end; we have simply arrived at the physical limits of this form of computing [7].

The relatively new field of parallel processing may shift this balance. There is effectively no limit to the amount of processing power that can be applied to a given problem, as long as that problem can be effectively parallelised [8]. Systems which were previously too costly to simulate may now be viable. Image processing methods stand to benefit enormously, as images can easily be broken into sections and simultaneously processed on different nodes in a cluster.

There is also another area where simulation times may be improved, within the code itself. Most high level languages have been designed to be as user friendly as possible, such as that in MatLab or Visual Basic. While this does make the design and implementation of non-time-critical processes a great deal easier, and despite the continued evolution of economic compilers, the use of these languages often results in massively inefficient code. As an example, converting a previous version of a near-field coded aperture simulator from MatLab to C++ and tweaking the algorithms used yielded a 100 times performance gain.

2.2 The Question

The question asked within this Thesis is simply; given the power parallel processing represents and with well-designed software written at an appropriately low level, can problems that are currently incredibly computationally expensive now be solved within a reasonable amount of time?

2.3 The Application

Even though a 100 times performance gain has already been attained in the near-field coded aperture simulator, individual simulations can still take over twenty hours to run. Eventually, three dimensional simulations will be required, which will be achieved by individually processing at least 50 distinct layers, running into thousands of hours of machine-hours (as opposed to man-hours). Clearly this is an unacceptable time frame.

As a case study, the near-field coded aperture simulation will be re-examined for three reasons: The problem is easily parallelised simply by breaking up the source images into smaller sections and delegating them to nodes in a cluster, and collecting the results at a central machine after the simulation is complete. Secondly, since the problem is familiar, the key question – that is, how viable is application of parallel processing within the field of bio-medical engineering – may be focused on. Results can easily be verified by comparing them with previous versions of the simulator, which in turn have been verified by physical trials.

All these points make a near-field coded-aperture simulator the ideal application for the basis of this thesis.

2.4 The Platform

To test this premise, a cluster of six Sony PlayStation 3's connected by a gigabit Ethernet switch will be used. The PlayStation has been chosen as, at the beginning of this project, it held the record for the lowest cost to performance ratio, at just $0.2 per billion floating operations per second(GFLOP) due to its processor [9], IBM's innovative Cell Broad Engine(CBE). The operating system used is Linux Fedora 7, as it is one of only two supported by the CBE software development kit [10].

2.5 Dissertation Layout

This dissertation will begin in chapter 2, which gives a brief history of computing and examines the move away from serial computing to parallel computing. Chapter 3 explores some of the work done in similar areas, including a brief overview of Ray Tracing and computing on Graphical Processing Units. Chapter 4 gives an overview of the hardware setup, the details of the network setup, the communication software and the particulars of the development environment. Chapter 5 examines the Cell Broadband Architecture(CBEA) in detail and provides a simple example of an easily parallelisable problem and how the features of the processor are used to solve it. There are three layers of processors involved in the in simulator, namely the central PC node, the two layers of processors on the PS3, and chapter 5 gives a brief description of each level's tasks and responsibilities, as it easier to understand the process as a whole first before trying to connect each of the pieces, before giving a more detailed explanation of the implementation of each strata.

Chapter 6 gives a brief introduction into the field of coded apertures, explaining their use, and why the simulations are so computationally expensive.

Chapter 7 details three aspects of the algorithm used to perform all the necessary calculations of the simulation, and how the capabilities of the Cell Broadband Engine Architecture are used to optimise the process.

To conclude, chapter 8 examines the performance of the simulator and provides a summary of the findings, including recommendations on future work.

Two appendices are given on CD. Appendix A supplies a Doxygen generated set of HTML documents demonstrating the layout of classes and files within the code, and Appendix B supplies the source code itself, divided into PC and PS3 code.

3 A Short History of Computing

Although modern computing has become a prominent force in everyday life, there is no specific time or place where it can be said to have been *invented* in its entirety. Instead, like many things, it is the culmination of decades of invention and innovation.

War has often acted as an accelerant for technology [11], and so it is no surprise that the Second World War is perhaps the first period in history that electronic devices began to play a major role. Technologies such as sonar and radar were developed, and the need to calculate ballistic tables and quickly and accurately resulted in the now famous ENIAC (Electronic Numerical Integrator and Computer) [11]. A machine running on over 18000 vacuum tubes, the ENIAC was the size of a small building. Although its original function was to calculate ballistic tables, it was used as tool in cryptography, and was also used to solve early problems on the Manhattan Project [11]. However, for all its accomplishments, the ENIAC is still a far cry from the tools available to modern computing. A list of major breakthroughs and dates is given below in Table 1.

Table 1: Important breakthroughs in Computing

Devise	Year of Invention	Inventor(s)
The Transistor	1947/48 [12]	John Bardeen, Walter Brattain & Wiliam Shockley
The Integrated Circuit	1958 [13]	Jack Kilby & Robert Noyce
The Mouse	1964	Douglas Engelbart
ARPANET(The original internet)	1969 [14]	
Intel 4004, First Microprocessor	1971 [15]	Faggin, Hoff & Mazor
The Floppy Disk	1971	Alan Shugart
MS-DOS	1981	Tim Paterson, Bill Gates
Microsoft Windows	1985	

In the last six decades, computing has made its way into every facet of our lives. Digital microchips can be found in everything from cellular phones to refrigerators. The uses are limitless. While the humble ENIAC could produce over 350 multiplication operations per second, the Japanese Earth Simulator centre is capable of producing over 35 trillion floating point operations per second [16].However, while computers may perform operations many thousands of orders of magnitude faster than a human can, there are still applications where this is not enough. This is especially true in the field of scientific computing.

As in chapter 0, Consider that there are 5 mL of water in a standard teaspoon, which translates to 5g. The molar mass of water is around 18g, which means there are around 0.22 moles of water in a single teaspoon. This translates into $0.277 \times 6.022 \times 10^{23} = 1.668 \times 10^{22}$ molecules. Even with 35 trillion operations per second, attempting to

simulate the interaction of the molecules in a simple teaspoon of water is a laughable exercise. Thus, the drive to develop ever faster computational devices continues.

Until now, the primary method of increasing computational power has been to increase the clock frequency of the microchip, known as frequency scaling [17]. The higher the frequency, the more clock cycles occur in a second and more work can be done, and while Moore's Law appears to be holding true for the present, this technique of increasing performance has begun to reach its limit.

The power consumption of a chip is given by

$$P = f \times V^2 \times C$$

Where f is the frequency, V is the voltage change, and C is the capacitance of the chip [18]. Thus we see a linear relationship between power and frequency scaling. On a large scale, this could consume a great deal of energy. Other factors contribute as well, such as memory latency (The time it takes to retrieve information from ram).

Thus a move has begun away from serial processing, that is, instructions which are completed on a single computational core in a specific order, to parallel processing, in which a problem is broken down and computed on two or more cores simultaneously, using a divide and conquer strategy. The greater the number of cores, the more work can be done simultaneously, and the problems with frequency scaling can be avoided.

3.1 Parallel Processing

3.1.1 Understanding the Problem

While processors with two or four cores have become common, manufacturers are promising to continue to increase that number in a fashion similar to frequency scaling [8]. A good example is the Cell Broadband Engine Architecture (CBEA) used in this project, which has one complete processing core, and eight "Synergistic Processing Units". Parallel Processing, however, is far more complex than ordinary, sequential processing for a number of reasons, a few of which will be discussed below.

3.1.2 Amdahls Law

Like frequency scaling, we might expect a linear relationship between an increased number of processors and performance, however this not necessarily the case. The limiting factor of performance increase in parallelism is the linear dependencies in a sequence of instructions. Imagine there is a series of commands that must be completed in a particular order, and no command can be started before its predecessor is completed. A good example of this is in old adventure computer games. A character must collect an apple to exchange for a screw-driver so he may repair a car to drive to town and so on. If this is the case, it does not matter how many people are playing the

game, it will still take the same amount of time. Thus the benefit an application can receive from applying multiple processors is dependent on how much of the application is inherently sequential. This is known as Amdahl's Law and is given by the following equation.

$$S = \frac{1}{1 - P}$$

Where S is the possible performance gain in percentage form, and P is the percentage of the application that can be effectively parallelised [19].

3.1.3 Additional Technicalities

Apart from the intrinsic limitations imposed on parallel processing by Amdahl's Law, there are a number of additional complexities associated with coding parallel applications. A common example is a race condition [20]: two processors are performing tasks on the same variable, the output dependant on which processor finishes first. This is obviously undesirable and because of this, there must be a way to lock variables such that only one processor has access to it at once. However, this may result in a deadlock condition [20]. Processor 1 has locked variable A and is waiting for variable B to be released, meanwhile processor 2 has locked variable B and is waiting for variable A. The result of which is a deadlock and the system crashes. One can see that there must be a degree of communication and synchronisation between processes in order to avoid these conditions.

Parallel problems are thus cast into one of three very broad and largely undefined classes [20, 21]: Fine Grained, Coarse Grained and Embarrassingly Parallel. Fine grained require a great deal of communication and synchronisations and communication, Coarse grained less so and Embarrassingly parallel problems require little or no intercommunication. In especially Fine Grained problems, by adding more processors than required, the additional overhead of data transmission and synchronisation may actually have a detrimental effect on the running time. [20]

Additionally, there is a problem of consistency. A sequential processor will always produce the same output given a particular input. Take, for example, Conway's game of Life [22]. The game is "played" on two dimensional grid with each square being classified as "alive" or "dead". Each cell interacts with its eight surrounding cells in accordance with the following four rules.

1. Any live cell with fewer than two live neighbours dies.
2. Any Live cell with two or three neighbours lives to the next iteration.
3. Any Live cell with more than three live neighbours dies.
4. Any dead cell with exactly three live neighbours comes to life.

The game is well known for its ability to demonstrate chaos theory and emergence of very complex and very different patterns from initial conditions that vary by even a single cell in the right place.

Given a grid with an initial state, a sequential processor will always produce exactly the same pattern every time it is run, since it will always process the cells in a particular order.

Run on a parallel system, however, it makes sense to divide the grid into smaller sections and allocate them to each core. Now, depending on subtle timing differences, each time the simulation is run, a particular cell on the edge of a division may have a different state, which would produce a new pattern nearly every time the game is run. A parallel process may thus produce unstable or unreliable results, depending on the application and how it is implemented.

3.1.4 Compilers

Sequential compilers have been around for decades now, and they have become very efficient at optimising even poorly written code. High level languages like Visual Basic have allowed people to very quickly produce reliable and user friendly applications, delegating the task of optimisation to the compiler.

In the new era of parallel computing, however, this is no longer true [20]. Complications, such as deciding how a problem should be split up, when and where synchronisation needs to occur and how to handle the delegation of memory are, for the time being, simply too sophisticated for a computer to handle efficiently, and must be managed by the programmer. So, curiously, the advent of this new era in computing has resulted in the necessity of coding practices being reverted back twenty or thirty years.

3.1.5 Conclusion

While Parallel Processing possesses the capacity to overcome the limitations of sequential computing, there are some problems that cannot benefit from this method of processing. In addition, it presents a number of challenges and pitfalls that must be thoroughly understood before the benefits can be achieved.

3.2 Introduction to Parallel Computing

Parallel computing is a complicated subject in terms of both hardware and software. As such, an introduction to the already well defined terms and patterns is given below.

For a computer to perform an operation, it needs both an instruction and a piece of data to be acted on. As an analogy to help understand each concept; consider the kitchen in a busy restaurant. Each chef representing a processor, each dish being prepared a task,

each action such stirring or chopping an instruction and each dish or ingredient a piece of data.

3.2.1 Flynn's Taxonomy

In the broadest sense, there are only four ways to process data and their classification is known as Flynn's Taxonomy [20]. First we have Single Instruction Single Data (SISD). This is essentially sequential computing, and is analogous to a single chef preparing a single dish, adding a single ingredient at a time. Then there is Single instruction Multiple Data(SIMD), this time the chef is preparing a number of dishes simultaneously, following the same instructions and performing the same actions for each but using different ingredients, or different quantities of the same. This form of processing is rarely used. Next is Multiple instruction Single Data(MISD). This is equivalent to multiple chefs preparing the same dish but each performing a different task, one chopping carrots, another adding salt and so forth. This form of processing is almost never seen. Lastly we have the most common variety of parallel computing, and the form you would expect to see in the philosophical kitchen; Multiple Instructions Multiple Data. Here each chef is preparing his or her own dishes, more or less independently of the others.

3.2.2 Hardware

There are a number of important decisions to make when designing a parallel architecture, a few of which will be discussed here.

3.2.2.1 Memory

Memory in computer is either distributed or shared [20, 21]. In the case of distributed memory, each processor has its own local address space. This type of memory is often physically distributed on the chip as well as simply being allocated to a specific processor in software. With shared memory, each core is connected to a single, unified address space via a high speed bus. Both distributed and shared memory types have their advantages and disadvantages. The CELL processor used in this project uses a combination of both; each core has its own 256kB cache of local memory but is also connected to an effective address space, which may vary in size, but is usually several orders of magnitude greater than 256kB [23]. In the case of the PlayStation 3, the shared address space is typically 256MB.

3.2.2.2 Levels of Parallelism

There are a number of levels of parallelism, each stacked on the level below it. A brief description is given below [20, 21].

1. At the lowest level is bit level parallelisation. This refers to the size, in bits, that a processor can operate on. If an 8 bit processor must add two 16 bit numbers, it must first add the two lowest 8 bit words, then add the two higher words,

possibly with a carry bit from the first operation, and recombine the results. Increasing the number of bits the processor can operate on to 16 would reduce this process to a single addition instruction. This is bit level parallelism.

2. Vectorisation and instruction level parallelism. These refer to the SIMD and MISD parallelisation discussed above. Some processors can perform the same operation on multiple data elements (vectors of data), or it may have multiple instruction pipelines, allowing it to perform more than one instruction in a clock cycle.
3. Multi-Core processors. Modern chips are being designed with multiple, independent cores, allowing them to perform a number of separate tasks at once.
4. Symmetric Multiprocessing. A number of identical processors are connected together via a high speed bus.
5. Cluster Computing and Massive Parallel Processing (MPP). Cluster computing is effectively a number of individual, stand-alone commercial machines connected together via network. This configuration is typically known as a Beowulf cluster. MPPs are similar to cluster computing, except that they are usually connected by specially designed network hardware, and are typically much larger than a Beowulf cluster, often reaching several hundred individual units. Many of the world's super-computers are MPPs.
6. The highest level of parallelism is Grid computing. Grid Computing links any number of individual machines over the internet, usually making use of machines that would otherwise be idle. Because of the unpredictability of bandwidth and latency, only embarrassingly parallel problems are feasible through grid computing. A well-known example is the SETI@HOME [24] projects, which utilises consenting computers to sift through the massive amount of data collected by satellites and telescopes for signs of life in the universe.

3.2.3 Software

There are also a great many decisions that must be made for an effectively parallelised problem. A few of these will be discussed below.

3.2.3.3 Partitioning and Delegation

The first step in writing the application is deciding what parts of the problem can be parallelised and how to delegate these segments to processors. Each core can perform an identical task on different data sets, or the processors can be set up like a production line, each processor performing part of the whole operation, with data being passed from one core to the next [21]. Each has its advantages and disadvantages. The production line model will allow functions to be streamlined and optimised as well as changed easily, but will increase the need for inter-communication. Having each processor perform the same task will save on communication, but will take up extra

space in local storage. Of course, there are often many ways to parallelise a problem, all of which need to considered.

3.2.3.4 Synchronisation and communication

Once it has been decided as to how required tasks are to be delegated, the amount of synchronisation between processors must be determined. Fined grained problems will require constant attention, while embarrassingly parallel problems may require none at all. There are two types of communication [23]; synchronous and asynchronous. Synchronous communication will stall a sending processor until a received signal has been generated by the recipient. Asynchronous communication will not. Which method is chosen depends on how closely the two processors must work. It is undesirable to have one processor running idle waiting for another as performance is lost, but synchronisation must also be maintained.

3.2.3.5 Application control and load balancing

Load balancing is an important factor in parallelisation [20]. Allocate too much work to a processor and you run the risk others running idle waiting for its results. Too little work increases the communication overhead, also resulting in a loss of performance. A good practice is the use of double buffering; that is, storing the next set of data to be processed while the current set is running. Once complete, the results can be returned and work on the second set of data can begin immediately, while a third set of data is loaded into storage concurrently. While this practise can effectively eliminate the transfer time of large sets of data, it also requires more space in local memory and must be carefully managed; it can be easy to overwrite data that has not yet been processed.

3.3 Conclusion

A parallel application must be structured very differently to a sequential application performing the same task. From choosing the hardware it will run on to the general form of the application, as well as similar problems at instruction level, designing a good, efficient parallel solution can be an extremely complex task. Once designed, implementing the solution while avoiding the issues discussed in chapter 3.2.3 can be similarly difficult.

Keeping this in mind, Chapter 4 discusses the chosen set up; including hardware, operating systems and software used, as well as the methods for establishing a secure channel between nodes and the communication protocol for inter-nodal communication.

4 Literature Review

The field of high performance computing is expanding rapidly, and a great deal of research is currently exploring the efficiency of both new hardware design and software paradigms. Much of the current work involves the potential use of Graphical Processor Units (GPU's), rather than the traditional CPU.

4.1 The GPU

A GPU is a processing unit created specifically to deal with the complexities of rendering a complex image to a screen, a common demand in media applications such as video games. This is accomplished by breaking this process down into a few well defined steps, known as the Graphics Pipeline, as shown in Figure 1 [25].

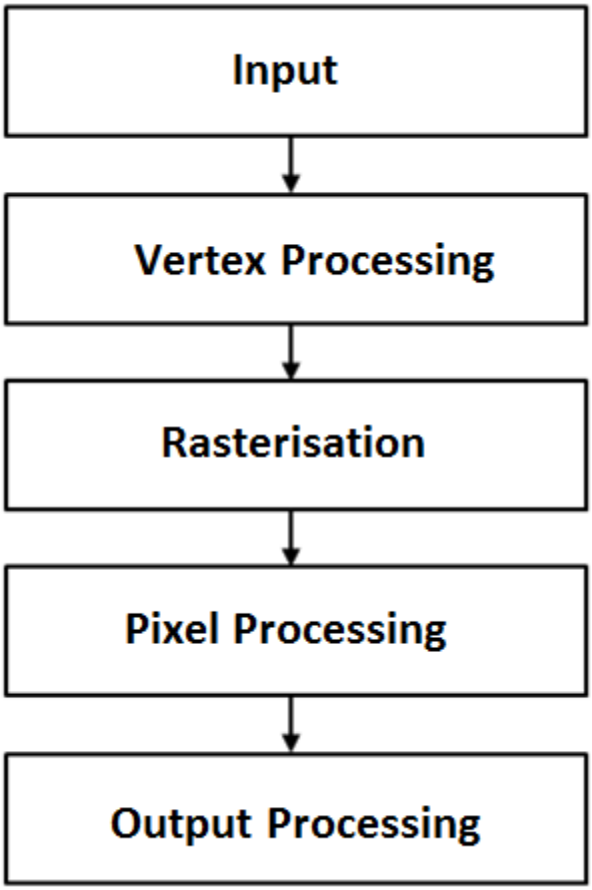

Figure 1: Simplified Graphics Pipeline

An input is first passed to the GPU. This is usually a collection of vertices created by the CPU. A vertex contains information about a particular object – it's position in space, colour, lighting information and so forth. At the second stage, this data is processed, which could involve smoothing edges, rotating, scaling or otherwise manipulating the geometry of the scene. The vertices are then Rasterised. Given the position of the

camera or viewer, this is simply the process of mapping an object to a collection of pixels on the screen, defining what objects are going to be displayed where. Once this has been done, the pixels are processed, and colours are determined based on texture, lighting, shadows and any other visual effects. The output of this stream is the final displayed image. Programs that instruct the GPU on how to act on pixels or vertices are known as shaders. This is a simplified view of the graphics pipeline, and each stage can be broken down into several sub stages.

4.1.1 GPU Architecture

Both vertices and pixels have a set of common, well known properties: All vertices are going to have a position in two or three dimensional space, all pixels are going to have colour and brightness and so forth. Because of this, the types of operations that are going to be performed on them are also generally predictable [25].

This, taken with the intrinsically parallel nature of displaying an image, means that GPU's can have a large number of highly specialised cores, as opposed to CPU's, which have a low number of very general cores. The ATI X1900 GPU, for instance, has 36 pixel shader processors and 8 vertex shader processors, whereas most CPU's currently have at most 4 general purpose cores.

As such, GPU's typically have much greater performance than a comparable CPU, but at the cost of being extremely limited in both the types of operations that can be performed and the format of the data that can be operated on.

GPU's are becoming increasingly flexible as their value to high performance computing projects become more obvious [26]. This has become known as GPGPU (General Purpose computing on the GPU). A few projects employing GPU's are discussed below in section 4.2.1. There are still a number of drawbacks to employing a GPGPU, notably that there is no good way to deal with conditional logic, and the number of instructions that can be queued up on a single core is quite small [26].

4.2 Ray Tracing

While the Graphics Pipeline discussed in section 4.1 is the most common method of rendering an image, it does suffer from a number of drawbacks. Because objects are handled individually and in parallel, it becomes very difficult to calculate effects where the objects influence the appearance of each other, such as in effects like shadows, reflection and refraction. An alternative approach with the potential to create much more realistic images is that of Ray Tracing [25].

Generally speaking, most rendering algorithms function by processing the geometry and lighting of a scene, and then projecting the result onto an image plain in front of a view point, an eye or a camera. Broadly speaking, ray tracing follows the reverse approach,

projecting a ray from the eye, through a section of the image plane, onto the scene [27]. This is shown below in Figure 2. The path of the ray intersects the objects that will be displayed on, or at least influence, the pixels on that section of the image plane. The below image also demonstrates that advanced graphical effects such as reflection and refraction, which are very difficult to achieve with the traditional graphics pipeline, may effectively be built into the algorithm [27]. This may be thought of as a physical simulation of actual light.

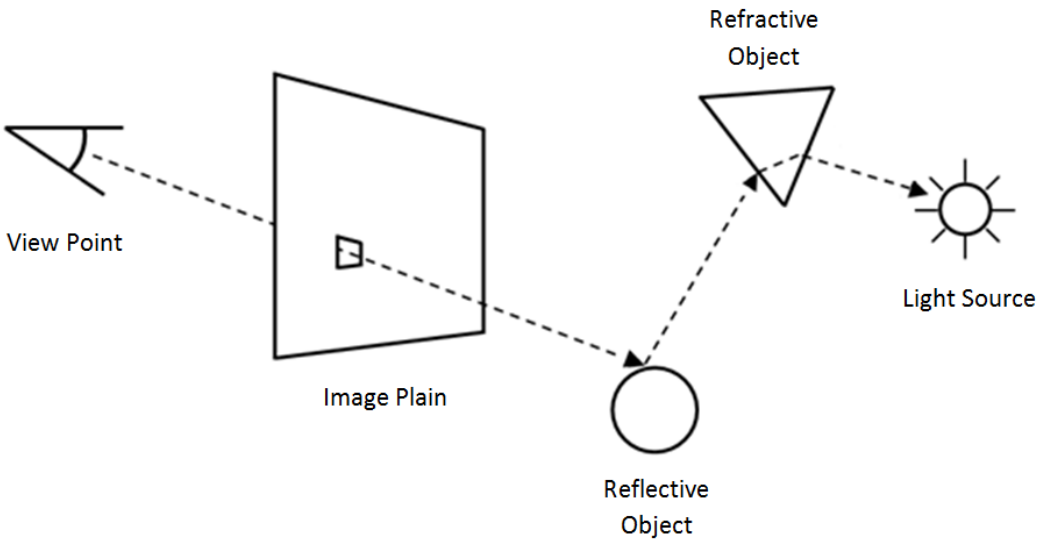

Figure 2: Illustration of Ray Tracing

Ray tracing is of interest in the context of this project because is similar to the process of tracing a ray through a coded aperture. It is also a highly computationally expensive process, and much research has gone into the problem of effectively parallelizing it so complex images may be rendered in real-time.

4.2.1 Ray Tracing on the GPU

Ray Tracing is an Embarrassingly Parallel problem. In principle, as many cores may be utilized as there are primary rays. For this reason the GPGPU, with its high number of processing units, seems like a natural platform on which to implement a successful, real time application, and indeed there have been several attempts to do just that [25].

It has been found that up to 95% of the processing time in a ray tracing Algorithm comes from intersection calculations [28]. That is, the process of following the path of the ray and determining which objects in a scene the ray passes through or comes into contact with. Keeping this in mind the first approach by Carr et al [29] utilized the GPU just as a means of calculating these intersections. The results were impressive, with 114 million ray/triangle intersection tests per second on an ATI Radeon 8500. This compared to an

estimated 20 million on a comparable CPU implementation at the time. The main drawbacks of this approach were that a significant portion of the computation, including ray generation and shading, were left on the CPU. In addition, the relatively low precision of the Floating Point numbers representation on the GPU (only 24 bit) resulted in image artifacts [29].

Purcell et al used a slightly different approach [30], implementing the entire algorithm solely on a GPU. This approach was calculated to achieve 56 million ray/triangle intersection tests per second on comparable hardware. The conclusion of the work was that real-time ray tracing may be possible with GPU architecture, but it would require the hardware to evolve to a more generally programmable pipeline.

Martin Christen implemented an approach similar to Pascall, and compared it directly with a similar approach on a CPU [31]. The conclusion was that while GPU's theoretically possess sufficient processing power, actual performance was only slightly better than on a CPU. Several drawbacks of the GPU were highlighted, which if improved could improve performance significantly. Predictions were made that advances in GPU architecture would make real-time ray tracing feasible.

The general consensus across these works is that while ray tracing is in theory a well suited application to the architecture, and while GPU's possess enough processing power to make the application far more viable than on regular CPU's, the hardware is still too specialised to be effectively put use, and do not presently provide significant performance gains. Predictions are made that the architecture will converge on a more flexible, programmable pipeline, and that this will dramatically improve the viability of the GPGPU [26].

4.2.2 Ray Tracing and Coded Aperture Simulation

Simulating a coded aperture bears a great deal of resemblance to running a Ray Tracing algorithm to render a scene, in so far as the primary aspect of both is tracing the path of a ray and determining points of intersection. However, the focus of many of these papers is creating efficient data structures to represent objects. This is not necessary in a coded aperture simulation, as it is known that the object the ray travels through can easily be represented by a grid. This allows for a much greater degree of specialization, and should allow greater efficiency. Also, the criterion for a successful Ray-Tracing implementation is real-time rendering, and this must be achieved for the methods to really make an impact. This is not the case with a coded-aperture simulation, and any performance gain could be considered a success. These factors may make coded aperture simulation on a GPU more viable.

4.3 The Cell Processor

An alternative to the GPU may be seen in the IBM Cell processor. This is discussed in detail in section 5. Put somewhat simplistically, the Cell may be viewed as half way between a typical CPU and a GPU. It has one standard, complete Power Processing Unit (PPU), and eight Synergistic Processing Units (SPU). The SPU's resemble the more specialized cores of a GPU, but are optimized for performing general Floating Point mathematical procedures [32]. The Cell processor is comparatively new architecture, but has already seen use in a number of high profile endeavors. Including the Folding at Home project, utilizing idle Sony Playstation 3's in houses all over the world to perform calculations involving the formation of complex proteins.

Another example is the University of Massachusetts "Gravity Grid". This is a cluster of sixteen PlayStation 3's. This cluster has been used for a variety of projects, including Binary Black Hole Coalescence using Perturbation Theory, and Kerr Black Hole Radiative "Tails" [33].

As a more concrete example Buttari et al benchmarked the Cell processor with a few common algorithms, such as an implementation of a Fast Fourier Transform (FFT), and procedures to solve both sparsely and densely populated system of linear equations [10]. The findings were varied. The sparsely populated linear equations achieved a performance of 155 billion floating point operations per second (Gflops), which is around 75% of the theoretical maximum performance of the chip. The FFT achieved around 90 Gflops, and the sparse linear algebra fell to 12.8 Gflops, only 6% of the maximum processor efficiency.

However, for comparison, a 3GHz pentuim 4 processor has a maximum theoretical limit of 12 Gflops. It estimated that traditional architecture can yield about 30% efficiency for sparse linear equations, for a total of 3.6 Gflops.

Buttari et al conclude that while the Cell processor has an enormous amount of raw processing power, there are several drawbacks, especially as part of the Playstation 3. Of concern were limitations on memory size and access speed, networking access speed and a lack of optimization for double precision floating point operations [10]. The general difficulty of the programming paradigm was also brought up as well, summarized in the following statement: "Writing efficient and fast code for the CELL processor is, in fact, a difficult task since it requires a deep knowledge of the processor architecture, of the development environment and some experience. In many cases, high performance can only be achieved if very low level code is produced that is on the border line between high level languages and assembly."

Despite this, the Cell has achieved some remarkable results, often within one to two orders of magnitude of performance gains over traditional processors. The variety of different projects it has been successfully involved in also speaks well of its capabilities.

5 The Platform

The cost of computing, measured in unit money/GFLOPS has been falling at an exponential rate over the last decade or so. Below Table 2 demonstrates the fall in the cost of computing.

Table 2: Cost per Gigaflop through history

Year	Computer	$/GFLOP
1961	17 million IBM 1620 units [34]	1.1 trillion
1997	Two Pentium-Pro-processors in a Beowulf Cluster [35]	30,000
2003	KASYO, University of Kentucky [36]	82
2006	ATI X1900 Graphics card [37]	1
2007	Sony PS3 [10, 37]	0.2

Thus, at the start of this project, the Sony PlayStation 3 held the record for the lowest cost to performance ratio. This figure is slightly misleading, as it takes into account the graphics card, which is inaccessible to open source developers. The actual figure is closer to $2/GFLOP [10, 37] which is slightly higher than the ATI X1900 GPU. Overall though, the theoretical performance of the Cell Processor and a GPU are comparable, and the Cell is expected to be far more versatile, and able to retain a higher performance ratio over a wider range of applications, and is therefore the platform of choice for this project.

5.1 The Sony PlayStation 3

The PS3 is driven by the Cell Broadband Engine Architecture (CBEA, or simply the Cell) processor discussed in detail in chapter 5. It also comes standard with an RSX GPU running at 550 MHz, and 256 MB of XDR main ram running at 3.2 GHz. Two USB 2.0 ports as well as a Blu-Ray disk reader allow the transfer of data. There is an option to load a UNIX based operating system (OS) on top of the simpler game OS [10]. There are a number of excellent tutorials easily available on the internet as to how to do this, though unfortunately the later versions of PS3 firmware have removed this option. A Linux release known as Yellow Dog was developed specifically for the PS3, however the freeware Software Development Kit (SDK) for the Cell processor recommends either Red Hat 5.1 or Fedora 7 [10]. Though once installed, a kernel modified to take advantage of the Cell processor must be used to replace the original.

5.2 The Cluster

It was decided that the initial configuration for the project would be six PS3s in a Beowulf cluster, but with a standard Linux PC box running as a central node, all connected via a

gigabit switch. Fedora 7 was chosen as the OS for all nodes in the cluster, as it is known to be compatible with the software development kit available for CBEA and is not hampered by the relatively small amount of RAM. An important feature of the cluster would be the capability to add more nodes as required without any changes in the software, if six PS3's proved to be insufficient.

In a Linux based system, there are a number of run-levels. A run level dictates how many features of the OS are active, with run level 1 being the most basic, and run level 5 being the highest, with all options enabled by default, though this can be customised [38]. Typically only run-levels 3 and 5 are used [38]. Run-Level 3 is the standard consoled based version of Linux, with a few essential daemons active. A daemon is a Linux term for a small program running in the background. Run level 3 typically only has a few essential daemons active, such as those responsible of networking and communication. Run level 5 adds a GUI (Graphical user interface) similar to Microsoft Windows on top of this, as well as a number of superfluous or unnecessary daemons which can be nice to have, but also consume performance. It is worth noting that even run-level 3 is still running on top of the system firmware. This is not unlike Virtual Machine software, such as VirtualBox [39], that allows a copy of Microsoft Windows to be simulated on a computer running the Macintosh operating system (MacOS). The PS3 is effectively running two operating systems simultaneously. This, coupled with what is currently regarded as a relatively small amount of RAM may stress the system and produce bottle necks with even very simple tasks, such as running basic C++ development environment software. Thus, run level 5 may run a little slowly to those used a normal PC. Since high performance is the goal of the project, only the central PC node will run in run-level 5. The PS3's will run in run-level 3 to avoid the extra load of a GUI and other non-essential daemons, except during development, where a good GUI is invaluable.

5.3 Inter-Communication and Message Passing Interface

There must be communication between nodes if a cluster is to function and although it is not recognized by any major body of standards, the Message Passing Interface (MPI) has nevertheless become the de facto standard in high performance intercommunication in clusters [40]. There are a number of MPI implementations, such as LAM-MPI [41] and OPEN-MPI [42]; however they all largely consist of a set of routines callable from development languages such a C, C++, FORTRAN [40]. The goal of the development of MPI was to establish a portable, efficient and flexible means for communication between closely linked clusters. It supports both synchronous and asynchronous communication as well as point-to-point (one to one) and collective (one to many or many one) communications. In order to ensure standardised data types, MPI functions require the pre-declaration of variable types, to ensure they are passed correctly between non-homogenous systems [40]. It is worth noting that to ensure the support of heterogeneous systems, as a variable is passed from one node to another, it is converted

to an MPI-defined data type. It is safe to say this has not been tested on Cell based systems, and may be problematic.

5.4 Establishing a secure channel and SSH

Before MPI can begin passing data between nodes, a secure channel of communication must first be established. Since the process must be automated, a system that requires the user to enter a password every time a connection between nodes is established is infeasible, yet basic security should always be maintained. SSH (Secure Shell) provides an answer to both these problems [43]. SSH uses public key cryptography to authenticate a remote machine and for the remote machine to authenticate the user if necessary [44]. A machine using public key cryptography generates two keys (a very long sequence of numbers), a public key and a private key. A message is encrypted with the public key, and can only be decrypted with the matching private key. While the two keys are mathematically related, it is infeasible to calculate the private key from the public one [44]. Thus each node has two files; one containing its private key and another containing a list of public keys. It can then encrypt a message with the appropriate public key and send it to the node with the corresponding private key, without the need for any kind of password. To add another node, one simply adds the new nodes public key to the other nodes public key lists.

5.5 Software development on the PS3

It is safe to say that software development on the PS3 was not a priority when Sony designed it, as its main function is as a media platform and game console. This, coupled with the fact that a non-standard kernel of Linux is being installed non-standard hardware, as well as the flood of compilation options Linux presents mean that even seemingly trivial tasks in the set-up have the possibility of being a stalling point in the development process. A few such examples are discussed below.

5.5.1 Installing The Software Development Kit (SDK)

The software development kit (SDK) for the CELL processor is freely available for download from the Brazilian Centre for Supercomputing. The tutorial that was followed recommended manually downloading the SDK, writing it to a CD and installing from there [10]. Following their recommendations, there was no difficulty downloading or writing the SDK to CD. Linux has dedicated software installation daemon known as YUM, and it is this that must be used to install the SDK. Despite changing the installation path in the appropriate folder from the internet site for the Brazilian Centre for Supercomputing to the CD-ROM, YUM insisted on downloading from the internet. It was by pure chance that the folder containing the path file was open while an installation was attempted, it

became apparent the file placed there was re-named before a second file with the original settings was created. After this, the system would attempt to install from the internet and if it failed, the new file was deleted and the original renamed, leaving no trace that anything had ever happened. There seems to be no way to change this setting. Eventually, after several attempts which failed due to a poor connection, the SDK installed successfully. This little technicality is ultimately trivial, but it illustrates nicely some of the more peculiar quirks involving development on the PS3.

5.5.2 Development Environment

While there is a version of the *Opera* development environment that supports the cell SDK [23], it ran so slowly that it was more hindrance than help. Instead, the text editor GEDIT was used to produce code, which was compiled, linked and run via the console. Debugging was accomplished with feedback from the compiler and the tried and trusted method of writing variable values to the screen, a task made significantly more complex by running several tasks in parallel. For this reason, most development was done with only the primary and one secondary core active.

With the platform and cluster setup discussed, chapter 5 examines the CBEA in detail, and explains how its many features add up to produce performance gains of up to forty times that of other comparatively priced modern processors, as well as giving a practical example of how these features may be put to use.

6 The Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture Engineis the result of an attempt made by Sony Computer Entertainment, the Toshiba Corporation and IBM, collectively known as STI, to address some of the issues presented by more traditional single core processors [45].

6.1 Architecture

The architecture of the Cell Broadband Engine is laid out in Figure 3 below [23, 32].

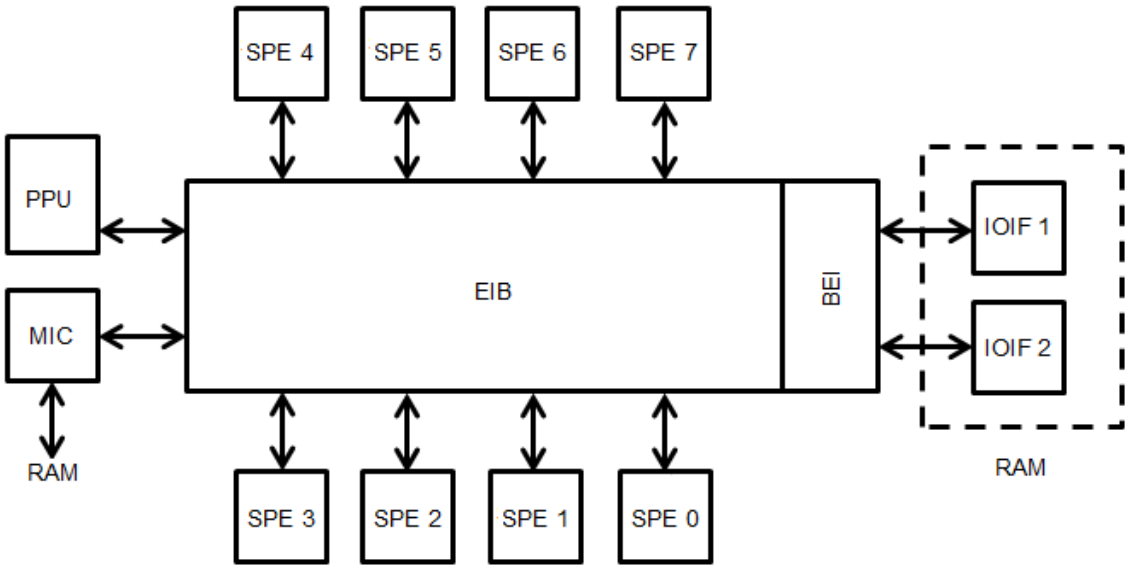

Figure 3: Cell Broadband Engine Arcitecture

More detailed diagrams of the PPU and an SPU are shown below in Figure 4 and Figure 5 [23, 32].

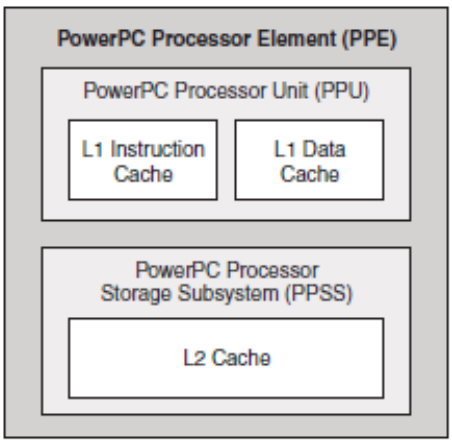

Figure 4: PowerPC Element Layout

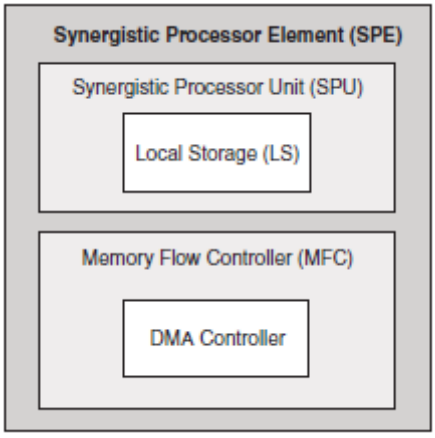

Figure 5: Synergistic Processing Element Layout

The Cell is composed of several components. There is one 64 bit PowerPC, known as the Power Processing Element (PPE) as well as eight "Synergistic Processing Elements" (SPEs). Each core is linked to the others, as well as external RAM and any other external sources by a high speed "Element Interconnect Bus" (EIB). The EIB can transfer up to 96 bytes of data per clock cycle. The EIB interfaces with external devised such as ram through the Cell Broadband Element Interface (BEI) and one of two Input/output Interfaces (IOIF). RAM may also be accessed through the Memory Interface Control unit (MIC). The PPE has separate level 1 caches for instructions and data, as well as a level 2 data cache. Each SPE has its own 256 kB cache of memory, as well as its own "Memory Flow Controller" (MFC). Henceforth, the cache associated with an SPU is referred to as "Local Storage" (LS), and the rest of system memory as "Effective Storage"(ES). Likewise, the memory locations will be referred to as "Local Address "(LA) and "Effective Address" (EA) [32].

The major difference between the Cell and other multi-core processors is specialization. Most current multiple core CPU's have two to four identical, complete cores. However, with the cell, the PPU and SPUs are distinctly different entities. The PPU is a full-fledged PowerPC core, with an extended instruction set for single instruction multiple data (SIMD) operations. Its purpose is to run the operating system as well as serving as a central, controlling hub for the eight SPUs [45, 32].

The SPUs, in turn have been stripped down; they would not be capable of the sophisticated instructions required to run an operating system. In trade, they have been optimized to run single precision floating point operations, with an extensive SIMD instruction set [32]. As an example, to further simplify and optimise the SPU's many data types have been cut entirely and only the most common floating point representations are supported. De-normalised numbers are automatically set to zero [45]. *Referring back to section 5.3, this feature might become problematic as MPI first converts all variables to its own format before being transmitted, and then converted back to the original format on the other side.* Additionally, there is only one rounding mode. As a result applications requiring a great deal of floating point operations, such as is common in media applications enjoy a performance boost in the range of an order of magnitude over more traditional architectures [46]. This is especially true in the field of scientific computing, such as performing complex matrix operations [23].

In addition to the performance benefits gained by the specialization of the SPUs, is a marked drop in the power requirements, making the Cell one of the most power efficient processors available in today's market [23].

Possibly the most innovative feature of the Cell is the Memory Flow Controller (MFC) attached to each SPU. An MFC is responsible for moving memory between local storage space and effective storage space. Because it is effectively a separate entity from the SPU, memory transfers may be managed independently and asynchronously from the SPU. By working on one block of memory while another is loaded to or from effective

storage, a technique known as “double buffering”, the effects of memory latency may be drastically reduced or nullified completely. Because of this, certain applications may gain up to two orders magnitude in performance, rather than the single order generally predicted [23]. Of course, this feature is something of a double-edged sword. Unless great care is taken on the side of the programmer, memory transfers may affect memory blocks currently being processed. The consequences of which could be anything from erroneous results to a total system crash.

6.2 Communication

In order to initiate a transfer of elements from effective storage to local storage, or vice versa, an SPU must write a request to the associated registers of its MFC. This is known as a Direct Memory Access (DMA) request. A DMA request must provide a local address, an effective address, the number of elements to be transferred as well as a 5 bit tag to identify the transfer [23]. This request may also be written by the PPU, or any of the other SPU’s. While being able to initiate a transfer from the PPU itself is certainly a useful feature, it reinforces the danger of memory transfers mentioned above, as the PPU has no means of determining where an SPU is in its instruction cycle unless specific synchronization points are coded in.

The MFC’s provide a variety of useful features, including scatter/gather instructions as demonstrated in Figure 6

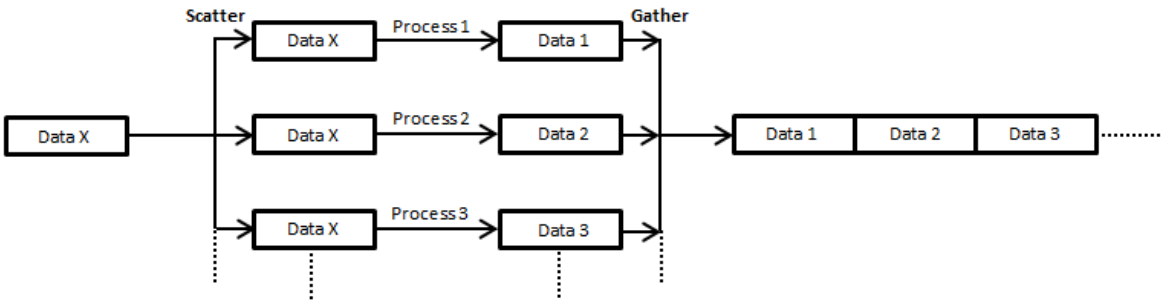

Figure 6: Illustration of Scatter/Gather Operations

A scatter command sends an identical block of data to each SPU, while a gather is the opposite; the PPU gathers a block of data from each SPU and combines it. The tags associated with a transfer play a major role in managing the flow of memory [23]. An MFC can queue multiple transfers, potentially with different tag, and when beneficial, the MFC may process these commands out of order. An SPU can also await the completion of all DMA requests, or only a specific selection of tags. This is accomplished by writing to a 32 bit wide mask register. 32 bits are used because the tag id is 5 bits wide, corresponding to 32 possible values. A ‘1’ is positive, and a ‘0’ is ignored [32]. It is worth noting that this register is, counter-intuitively, defaulted to all 0’s. In other words, the SPU will not wait for any transfers to complete before acting. The register must be

manually set. This simple feature makes the complicated task of parallelization a great deal easier, by allowing the programmer to easily distinguish different threads within the system and to act accordingly. As a simple example, deadlocks, where two processors await information from the other the before continuing can easily be avoided simply by assigning different tag groups.

A DMA command is not the only means by which an SPU can communicate with the PPU: There are also “Mailboxes” and “Signal Notification Channels”, or simply “Signals” [23]. The differences between the two are somewhat subtle and are described below. Both Mailboxes and Signals are unidirectional, that is, depending on the direction, they may be used to communicate with the PPU from the SPU, or vice-versa, but one channel may not do both.

6.2.1 Mailboxes

There are two single entry, 32 bit wide outgoing mailboxes (from SPU to PPU), one of which may be configured to trigger an interrupt. There is also one four entry inbound mailbox, meaning up to four 32 bit wide messages may be stored. Associated with this mailbox is a 2 bit wide register containing the number of awaiting messages. It is imperative to query this register before attempting a read, as reading an empty mailbox will stall the SPU until a message arrives. Reading from the mailbox will consume one entry and decrement the counter register [23].

6.2.2 Signal Notification channels

There are two, 32 bit, signal notification channels, both of which of are inbound. The second channel may be configured to trigger an interrupt, but it is the first channel that presents the most interesting features. It may be configured to operate in two modes; overwrite mode and logical OR mode. In overwrite mode, the PPU simply replaces any data that was previously in the signal box. Needless to say, this is somewhat dangerous as there is no guarantee the previous message was processed.

In “Logical Or”, as the name suggests, any incoming message undergoes a bitwise OR operation with any signal still pending in the signal box [23]. This effectively allows the programmer to line up to thirty two discreet commands for the SPU to perform without the possibility of overwriting a previous message. A read from either Signal channel resets all bits to 0.

A common example of the uses of signal and mail boxes is in the initiation of a DMA transfer. An SPU may alert the PPU that it requires another block of memory to process by writing a pre-defined number to an outgoing mailbox. The PPU will pack the required data into a single contiguous block before writing the address of the first element and the number of elements in the block to the four entry inbound mailbox before writing a “complete” signal to the inbound signal channel. Meanwhile, the SPU has stalled, polling

the signal channel a “complete” signal, before reading the address and number of the data elements. Using this information, it will then initiate a DMA transfer and assign a tag to the transfer, polling the DMA register until the transfer associated with the tag is complete before finally writing a “Transfer complete” message to an outgoing mailbox, at which time the PPU can release the reserved memory. Of course, the SPU need not be stalled polling for completion signals, it may instead process any remaining data assigned to it between polls, reducing performance losses.

This process might seem overly complex for a simple transfer of memory, but it leaves both the PPU and the SPU free for other tasks while the transfer is taking place.

6.3 The SPU

One of the major advantages the Cell has over other CPU or GPU architectures is the ability to load each SPU with different ‘programs’. That is, each SPU may perform functions unrelated to any of the others. This is known as context switching [23], and allows the SPUs to be configured to run the same instruction set, or to be chained together in a similar fashion to a production line. This also frees up additional memory for use in the program itself; a blessing considering each SPU has only 256 kB of local storage.

The true power of the SPU lies in the extensive SIMD instruction set. There are 128 4-word long registers (on the SPU, a word is 32 bits) that support these operations [32]. These are known as vectors. This effectively quadruples the processing power of a core that has already been stripped down and optimised for single precision floating point operations. However, this can present the programmer with some interesting challenges as to how to arrange data, as not all problems split neatly into four block units, adding to the already complex task of optimising a problem.

A simple example of programming for the Cell is presented below in an iterative approach to calculating PI.

6.3.1 Calculating PI

The method discussed is a common, iterative method of calculating Pi, known as a Monte Carlo Method [47]. Consider a square whose sides are given as 2 units long, then the area of that square is four units squared. Now consider a circle in the centre of the square with a radius of one a unit as is shown in Figure 7, then the area of the circle is πr^2, which equals pi because r is 1.

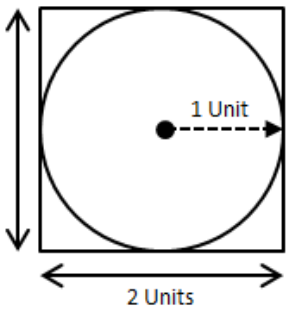

Figure 7: Monte Carlo Pi Estimation

The ratio of the area of the circle to the area of the square is

$$\frac{\pi r^2}{2^2} = \frac{\pi}{4}$$

Now random points are generated within the square, and the number of points that are also within the circle are counted, using the equation $\sqrt{x^2 + y^2} < 1$. With enough points the ratio of the circle to the square - **π -** may be calculated [47].

The most obvious approach to solve this problem with an ordinary processor is shown below.

1. Generate a random pair of points
2. Square the points.
3. Add points together
4. Root result.
5. Compare the results to 1, if less than, add to a total.
6. Repeat 1 – 5 as often as necessary. More repetitions will yield greater accuracy [47].
7. Divide the total number of points within the circle the total number of points generated and multiply by 4.

Now, from an optimisation point of view, there are several shortcuts that can be made immediately. Since the root of 1 is 1, the costly root operation may be done away with entirely. Secondly, while generating two random numbers between 0 and 1, might not take much time, there is a more efficient way to accomplish it. Instead two arrays with ten thousand random numbers each can be created, the pairs of points may be cycled to create a sample set of 10000^2 = 100 million discrete points, which is more than sufficient for the purpose.

Now, four X coordinates and four Y coordinates may be copied into vectors. Multiply and add the results, then compare it with 1, and if it is less, 1 is added for every appropriate result to a running total. Once complete, cycle the coordinates and repeat the process.

However, there is still one way to further improve performance. "If" statements can dramatically slow down a process, as the SPU must first make the comparison before fetching the next instruction to execute. One of the many SIMD instructions available to the SPU is a simple less than/greater comparison. A positive result will result in a word comprising all 1's, while a negative result yields all zeros. If this result then undergoes a bitwise AND function with a vector containing the number 1, the result may then be added directly to the running total without any branches in the code at all.

Of course, each SPU on the CELL may perform this entire process without any communication between each other, and the end results may be passed to the PPU via Mailboxes.

The result of this process is 50 times speed improvement over a 2.4 GHz Intel Core Duo processor running the process initially described.

Chapter five discusses the background of the application, including a description of nuclear imaging and coded apertures. Chapter six briefly discusses the purpose and responsibilities of the three layers of processors; namely the central PC node and the PPUs and SPUs on the PS3's before giving a more detailed explanation of the implementation of the software of each, with a particular focus given to the communication between layers.

7 Nuclear Imaging

7.1 Background

Nuclear imaging is the practice of introducing small quantities of a radioactive substance, known as a radiopharmaceutical, into the body [48]. There are many such radiopharmaceuticals, each intended to interact with different tissue types such as lung, heart, kidney, thyroid, etc. They may be taken up readily by some tissues and not by others thus introducing activity differences which can be visualised with a gamma camera [48].

Unfortunately, high energy electro-magnetic rays are difficult to focus with a simple optical lens. For instance, X-rays may only be focused by grazing incident reflection at energies lower than 10 KeV [49, 50].

For this reason, the usual method of producing a usable image in nuclear imaging is with the use of a collimator [51]. A parallel-hole collimator consists of septa, or cells, and is designed to allow only rays perpendicular to the source and camera surface pass. This essentially produces a one-to-one image on the camera, demonstrated below in Figure 8 [51].

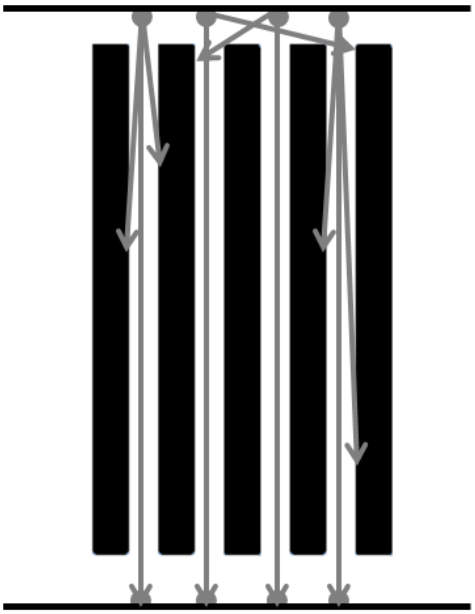

Figure 8: Illustration of a Parallel-Hole Collimator

However, a parallel-hole collimator is a very large, heavy apparatus that is difficult to maneuver into position, and may restrict the resolution of the final image. An alternative to a collimator is the simple pin-hole camera. The resolution of a pinhole camera is inversely proportional to the size of the hole; a very small hole would produce a very high resolution, while a large hole would yield a poor resolution [51].

7.2 Coded Apertures

The major advantage of a pin-hole camera is that it works equally well electro-magnetic rays of all energies. Regrettably, a single pinhole camera is not quite up to the demands of nuclear imaging. In order to yield a suitably high resolution image, a relatively small hole would have to be used, restricting the number of rays that would pass through. Due to current limitations in the sensitivity of the gamma cameras used, a patient would either be forced to remain still for an inordinately long time, or be exposed to dangerously high levels of radiation [51].

The solution to the problem is to use multiple pin-hole cameras on a single plate, all placed in mathematically significant positions. The greater the number of holes, the greater amount of radiation passed to the camera. This is known as a coded aperture [51, 50]. The result is a number of overlapping images, which may appear to be a random blur to the naked eye, but may be decoded into a single image, given the geometry of the aperture plate. The advantage of this method is that as many holes as necessary may be used without compromising the resolution of the final image. Also, a coded aperture would be much smaller and weigh significantly less than a collimator [51].

Coded apertures have been used in astronomy for a number of years, but have only recently been applied to the field of nuclear imaging [51]. However, their use in astronomy is viable because the source of the radiation is so far away that the EMF rays from a single source are effectively all parallel to each other and perpendicular to the surface of the aperture plate and gamma camera. This is known as far-field imaging.

In contrast, in nuclear imaging the aperture plate and gamma camera are placed as close the patient as possible in order increase the ray count on the camera. As one might expect, this is called near-field imaging. Due to the close proximity of the source, the aperture plate and the camera, rays are scattered at all angles, as shown below in Figure 9 [51].

Because of this, when the image is reconstructed, artifacts and ghost images are introduced. There are a number of methods that may be used to reduce or eliminate these images, but it is clear the precise design and dimensions of the aperture plate are important in the application of this technique.

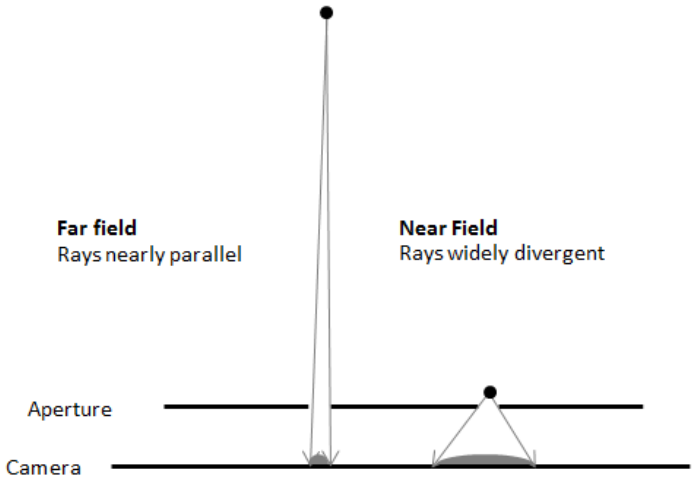

Figure 9:Far and Near Field Coded Apertures

7.3 Design and Testing

While the design and efficiency of a coded aperture may be done purely mathematically, its effectiveness must still be established in real-life situations.

It is unreasonable to physically construct every variation of a design, and deliberately expose a patient to even small amounts of radiation simply to determine the effectiveness of a particular pattern. Apart from the danger to the patient, it would be prohibitively expensive in terms of both money and time.

The next best thing then, is a computer model of the process. Even modern serial processors have been found to be inadequate for this process, due to sheer computational power required. Literally hundreds of billions of rays may be traced in a single simulation.

The University of the Witwatersrand has thus far produced two versions of such a simulator. The first, took up to two weeks running on six machines to run a single simulation [51]. The second, written in C++ with a slightly different algorithm was at least 100 times faster, taking just 20 hours to run on a single machine. However, all simulations run thus far involved a strictly two dimensional source. The ultimate goal is to run three dimensional simulations by simulating two dimensional 'slices', one at a time. Unfortunately, this may involve fifty or more such slices, bringing the running time of an individual simulation back into the realm of the unacceptable.

The requirement, then, is for a massive increase in the processing power available to run these simulations. The next chapter briefly examines the history of computing, as well as the direction future processors may take.

8 Software Architecture Overview

As was discussed in section 2.4, there are three distinct levels of hardware: The central PC node, the PPUs on the PS3s and their corresponding SPUs. As one would expect, there are three correlated layers of software, all written and compiled separately. This chapter first presents a brief description of the design philosophies of each layer of software, before giving a more detailed description of the implementation.

8.1 Task Level Paradigms

Since it is known that the slowest component of current parallel networks is communication, this should be kept to a minimum. This can be achieved relatively easily in a Coded Aperture simulation by breaking the sources up into sections, performing all the costly calculations separately and only recombining the results at the very end. However, care must be taken to strike a balance between communication latency and load balancing. In the PC and PPU layers, preference should be given to allocating work over performing their own calculations. Also, since the PPU has access to over a thousand times the memory available to an SPU, and the PC potentially even more so, it makes intuitive sense to store data for later computation, giving priority to processing requests from a lower level.

8.1.1 The PC

The PC represents both the input and output sources of the simulation. On the input side, there are several tasks that the PC should perform before passing the information on. It must convert the source and aperture bitmaps into useful data structures; in this case, two dimensional arrays. This could be done on the PPU's but it make little sense to perform a task six times when it can be done once, and also guarantees uniformity. In the case of multiple queued simulations, it should ensure that only one instance of each source and aperture array is stored with a unique key, saving space and once again guaranteeing uniformity. It should pack the rest of the simulation information into a single data structure that is easily transferable. It should also perform basic functionality tests on the given information; ensuring rays do not pass around the aperture and any other factors that might cause the simulations to fail.

With the input taken care of, the PC can now transfer each source, aperture and information data structures to the PPUs. With 256Mb of memory, they can easily store all the information without having to request and replace data from the PC.

The PC can now delegate rows of sources from a particular simulation to each PPU, and wait to recombine the results.

Once all simulations have been completed, the PC can decode each image, and the process will be complete.

8.1.2 The PPU

Because it must also run an operating system in the background, the PPU can easily become a bottleneck. For this reason, the PPU should be treated purely as administrative layer: assigning work to and responding to requests from the SPUs while relaying the results back to the PC.

8.2 Instruction Level Paradigms

The above section spoke about how the program as a whole should be split up and managed. This is the point where all the actual calculations and computations that make up the simulator are done. As such, the difficulties that must be overcome and the choices that must be made are of a very different nature.

8.2.1 The SPU

There are a number of complex calculations the SPU must perform, and as mentioned in section 6.3 the true power of the SPU lies in its extensive SIMD instruction set, which allows it to operate on a vector of four 32 bit words in a single instruction cycle. The challenge is then to organise the program structure and data such that full capability of feature is utilised. A task that, at times, is more easily said than done. As an example, despite the fact that the SIMD vectors contain four entries, only two multiplications may be done concurrently. In digital logic, the product of two 32 bit words is a single 64 bit number. Since a vector is only 128 bits wide, it has space to store only two answers. This is true even if it is known that the product can always be contained in the original 32 bits, as is the case when the number is only ever going to be multiplied by a 1 or a 0. This is an operation that is relatively common, and so a more efficient method should be found.

Another concern is that the SPUs have not been designed to handle branches in the code well [23], so “if” statements and “while” loops should be avoided if possible. The SPUs should instead perform exactly the same sequence of instructions regardless of the circumstances. As will be seen in chapter 9 these restrictions often result in the use of bitwise instructions, and in many places the code more closely resemble assembly level languages than standard C++. In turn, this requires anyone attempting to understand or alter the code to have an intrinsic comprehension of microprocessors at a more primitive level.

The final challenge is that the SPU has only 256 kB of memory of local memory. It is self-evident that at some point in the ray tracing process information about the aperture must be required, however these files (or arrays used to represent them) may be very large, and so a method of selecting and storing an appropriate section of the aperture must be supported. Additionally, memory transfers and many calculations are dependent on the data being aligned in a particular way in local memory. Once again,

this more closely resembles developing embedded applications on small microprocessors rather than programs written in higher languages to run on computers.

8.3 The PC

The PC layer of software can be broken down into three specific tasks; initialization which includes loading and copying simulation details to all nodes, load delegation which ensures no processors are left idle for want of work and result recombination and processing.

8.3.1 Input

The input to the simulation software is simply a text file containing the following information: The names of the source and aperture image bitmaps as well the location of the aperture and camera on the z axis – the source is assumed to be at zero – and the thickness of the aperture. The nomenclature of axes used is illustrated in Figure 10 below. It must show the length of the sides of the source, camera and aperture as well as the number of block per side (the resolution) of the camera, which is assumed to be square. The resolution of the source and aperture images is read directly from the image itself. Lastly, the input file must contain the attenuation constant α, which is dependent on the material used in the aperture as well as the energy of the radiation. Units are usually given in millimetres, though any unit of length may be used as long as it is used consistently throughout all the variables in the simulation.

Each unique source and aperture image is assigned a unique key, to avoid storing multiple copies of the same image, and to ensure uniformity.

In order to transform the source and aperture bitmaps into a two dimensional character array, an open source C++ library, easyBMP, was used [52]. A bitmap pixel consists of three colours – red, green, and blue. In a grey scale image, the values for all three are the same, and so only the blue layer was extracted. In the source image, light colours represent areas of high radiation and dark areas low radiation. A black pixel yields a value of 0 and a white pixel gives 255. In the aperture image, white represents empty space while black represents solid material. In this case white is assigned a value of zero and, for reasons explained in section 9.2, black is given the value -1, which currently looks to the PC like 255, but more importantly, yields a register filled only with 1s.

It is assumed that source, aperture and camera are aligned, and so the centres of all three are set to be at 0,0 on the x,y plane. If for some reason, the user wishes the source or the aperture to be out of alignment, he may pad the image with a black margin.

If one viewed the setup as shown below in Figure 10, with the source at the front and the camera at the back, the left and down represent negative values, while right and up represent positive values. Camera cells are numbered, starting with 0 at the bottom left

and increasing to the right until the border is reached, then the numbering continues from the left one row up. Although no structure or variable actually contains theses values, they are used when recombining results.

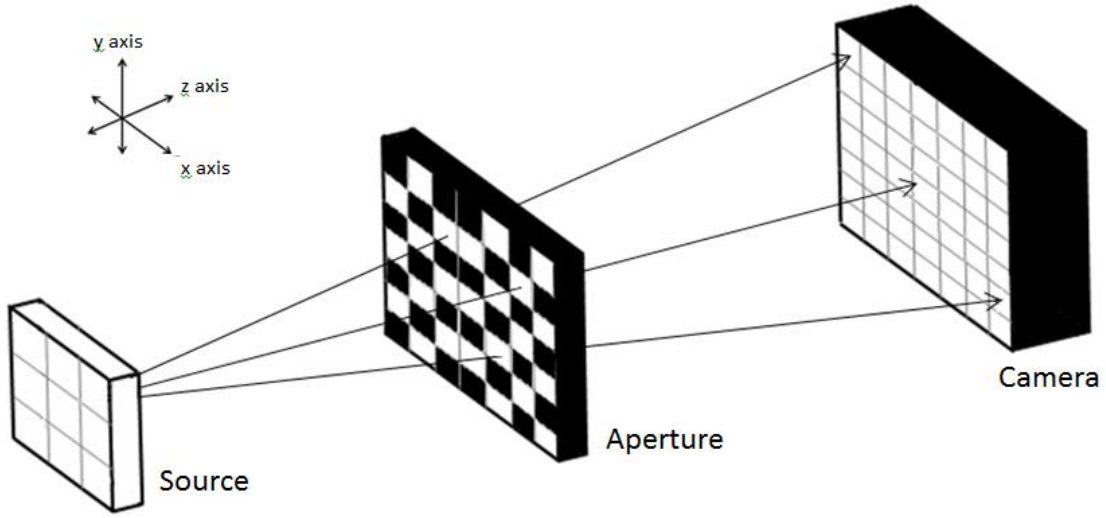

Figure 10: Coded Aperture Simulation Setup

From the dimensions and resolutions of the source, aperture, camera, the PC calculates the distance between the centres of two adjacent source blocks ($\Delta x, \Delta y$) and the distance between two adjacent cell boundaries of the aperture and camera.

The PC then packs all the simulation information into easily transferable data **structs**. (A **struct**, short for structure, is a C++ term for a user definable data type which may be composed of any number of other data types, a simple example is a Student **struct**, which would contain **strings** for name and surname, and **unsigned integers** for student number, date of birth, ID number and so forth) With this this accomplished, it then performs a few basic checks. It ensures source, aperture and camera are all roughly the same size, it checks to see that the aperture does indeed come before the camera and that the aperture and the camera do not overlap because of the thickness of the aperture. It also checks that rays traced from the bottom left of the source to the bottom left of the camera pass through the aperture, and the same for the top right, ensuring all rays pass through the aperture. If a problem is found, it alerts the user and asks if he wishes to continue, if necessary black margins may be added to the aperture.

Finally, source and aperture arrays, as well a simulation data structs are copied to each PPU and the PC transmits a “Begin” signal, once received, each PPU replies with a request for work, and we move into the run time phase of the simulations.

8.3.2 Run Time

Having each PPU request work instead of directly assigning tasks may seem somewhat unintuitive, but it makes load sharing a good deal easier. Instead of polling each PPU in turn, the PC need only check if a request has been sent. In answer of a request, the PC will assign a PPU with a simulation key and a number of rows of the source to process.

The PC may also receive a "complete signal", indicating that there are results ready to be transferred. The results will come in the form of an array, the first number indicating the simulation key, the second number is the quantity of results sent in that batch and after that are the results themselves in blocks of 10. The first unit in a block contains the number of the central cell on the camera, where the results are to be added, and the remaining nine are the results for the central cell and its eight neighbours. The necessity for the values in the 8 neighbouring cells is explained in section 9.3.

A "Complete" signal does not necessarily mean the PPU has finished processing its assigned load, merely that an array on the PPU whose purpose is to temporarily hold results is full. As mentioned, DMA transfers require memory to be 16 bit aligned, or 64 bit aligned for maximum transfer speed and so memory management on them can become tiresome. Also, the PS3 has only 256 MB of RAM, while the PC may have up to 4 GBs, thus it makes sense to move results from the PS3 to the PC as quickly as possible. The results are not processed on the PPU for the reasons mentioned in section 8.1.2.

The PC then has three responsibilities during runtime; in order of importance they are work delegation, result transmission and result processing.

8.3.3 Completion

Once all simulations have been completed, the PC must only finish processing any results it still has stored and write the completed simulation results to a file for de-coding.

8.4 The PPU

As mentioned in section 8.1.2 the PPU has the potential to act as a bottleneck with even relatively simple calculations. The PPU layer of software, therefore, has been designed to act purely as an administrative layer, sending information forward to the SPUs and sending results back to the PC.

Initially, the PPU waits for the PC to copy forward the source and aperture image arrays, first receiving the dimensions so it can call **malloc** (A C++ command used to requisition a contiguous space in memory) to allocate data space for each image. With the sources and apertures received, it creates an array containing the memory addresses of the start of each row source and aperture, as well a as an array containing the x-coordinates of the centre of each cell in each source, using the position of the bottom and left most source

cell and the Δx Δy calculated previously. The reasons for these calculations will become apparent in section 9.2.

With all relevant information received or calculated, as with the PC, the PPU forwards a signal to the SPU's that they begin.

The SPUs may send back a number of signals, most of which are requests, with a few 'task complete' indicators. The SPU's main request will be for source cells to process. The PPU has two 32 entry arrays that will be sent to the SPU: One floating point array and on character array. The first entry of the character array is the simulation key. If the key is different from the one currently being processed by the SPU, the SPU will send a request for new simulation information. The remaining 31 entries are the intensities of the source points. The first entry of the floating point array is the y coordinate of the row being sent, the remaining 31 entries are x coordinates. If an entry in source row is zero, the PPU will skip it and move on to the next entry, which is why it must also supply the x-coordinates of each source point. If the end of a row is reached, a 0 will be set in the intensity array and the SPU will process up to that point before requesting more data.

The SPU may also request the memory address of the starting point of a particular row on the aperture image. Since memory is severely limited, the SPU may not be able to store the entire aperture, thus it may be forced to store only a small part relevant to the space the currently simulated rays pass through. This is why the memory addresses of the start of each aperture row are calculated once initially and then stored. It was decided that the SPUs could only reasonably store 5120 cells at a time; only ten or so rows of the larger apertures simulated.

The SPUs may also request the memory address of an array into which to move results. Keeping in mind that these arrays must be 64 byte aligned, each PPU has 36 such arrays, each ten times the size of an SPU result batch. Once again, memory restrictions prevent the SPU's from storing too many results at once. By making the storage arrays on the PPU ten times the size of their SPU counterparts, communication times may be kept to a minimum. Since the PPU arrays are contiguous, the SPU may calculate the next address to transfer results to. An SPU will never mix results from two different simulations, they will instead request a new address. In this way, results for different simulations are kept separate. The PPU has two arrays monitoring the result storage arrays. The first keeps the simulation key associated with the results, the second contains one of only three values: 0 = Available for allocation, 1 = In Use, 2 = Complete.

Once a "complete" value is found, the entire array is sent back to the PC, along with the appropriate simulation key, and reset to 0 for re-allocation.

The PPU will continue this loop until it's delegated work load is completed, at which point it will request more work from the PC. If the PC sends a 'no work remaining' signal, the PPU will wait until all SPUs return a 'complete' signal before ensuring that all threads are successfully terminated before sending the remaining results back to the PC along with its own 'complete signal.

8.4.1 Communication conflict

While integer numbers, both signed and unsigned, could be passed freely between the PC and PPUs, any floating point numbers sent between them were always automatically set to zero in the process. The best guess as to the reason behind this behaviour is the somewhat expected clash between MPI's requirement to convert floating point numbers to a native format before transmission to ensure compatibility in non-homogenous systems, and the Cell's reduced number of supported data types. Several implementations of the MPI protocol were used, including LAM MPI and OPENMPI, two of the most common, with identical results. Creating a new integer variable before typesetting and copying the floating point variable into the new integer variable, one bit at a time, then transmitting the integer and reversing the process on the other side met with some success, but proved too unreliable to be a valid solution to the problem. While MPI has become the most frequently used communication protocol in modern clusters, an alternative may be to revert to the older PVM (Parallel Virtual Machine) protocol.

Another alternative is to convert every unit in a floating point number into an integer, recombining them into a single integer value while storing the position of decimal point, transmitting these to the PPU and then re-converting these to back to a floating point. However, assuming each number is ten digits long, converting a floating point to an integer, or vice versa would require ten multiplications (to get one unit at a time to the left of the decimal point), ten rounding procedures (to eliminate the numbers to the right of the decimal point), ten subtractions (to remove the now processed number), ten further multiplications (to place the number in the correct place in the integer) and then ten additions to reconstruct the integer number, before transmitting. The procedure would then have to be reversed to yield the original floating point number on the other side. It was decided that this procedure consumes too many resources for what is ultimately a relatively unimportant procedure.

8.5 The SPU

As one might expect, the PPUs layer of software is the most complex, as it must deal with both the administration of receiving details, sending results back to the PPU and, of course, performing all calculations pertinent to the simulations. As was discussed in section 6.3, coding on the SPU is a far more complex task than on either the PC or the PPU due to the fact that the data must be aligned and arranged to take full advantage of

the attached MFC module and SIMD instruction set, as well as manually managing the limited amount of memory while avoiding any branching code, such as “IF Statements” and “While Loops”, where possible [23].

An understanding of the computational complexity of low level commands, such as the number of clock cycles required to perform an addition calculation versus multiplication is also useful when attempting to optimise performance. Three examples of overcoming such issues are given in chapter 9 . This chapter will instead focus on the structure of the code and detail the responsibilities of the SPU.

There are three basic layers of abstraction in the SPU layer, while each layer is stored in separate **.cpp** and **.h** files, only the lower two are represented by classes. The first, caSim_spu is simply responsible for initiating the SPU and ensuring its successful termination. The second, henceforth referred to as the spu_manager, is the administrative layer and is responsible for all communication between the SPU its associated PPU, sending results and requests and handling inbound commands. The lowest layer, called base_spu, is left to co-ordinate all calculations involved in the actual simulation. The complete code may be found in appendix B.

8.5.1 caSim_spu

In clarification of the name, ca refers to **c**oded **a**perture, Sim refers to simulation, and spu is simply following a naming convention, and refers to the fact that the **.h** or **.cpp** file is associated with the SPU rather than the PPU. This first layer is as simple as described. Its only purposes are the successful initialisation of the SPU, as well as transferring some data related to natural exponential function discussed in section 9.4 and the successful termination of the application. After initialisation, caSim_spu creates and calls a single instance of the spu_manger class.

8.5.2 SPU_Manager

The Spu_Manager class is more complex, handling a number of instances of a-synchronous memory transfers and double buffering, ensuring no calculations are being performed on a section of memory currently involved in a memory transfer.

As mentioned in section 8.3, a particular structure (or struct) is used to transport source information. A 32 entry floating point array stores the x-coordinates of the corresponding source points, with the first entry containing the y-coordinate of the contained row. Another 32 entry character array holds the related intensities of the sources, with the first entry containing the simulation key. The SPU_Manager contains two such structs, one active, the other held in backup. When one source is finished, the backup source is made active and a request for a new set of source information is sent. The reply is handled by the MFC and loaded a-synchronously in the background, thus no time is wasted waiting for a reply from the PPU when a source has been finished. This is a text-

book example of double buffering, and is used extensively by the SPU_Manager. Of course it must keep track of which source is currently active, which block in the source is currently in use, and most importantly, whether the transfer of new data has been completed before switching active source structs.

If the simulation key of a source struct is different to the one currently being processed, the SPU_Manager is responsible for sending a request for the new simulation details, initiating the transfer once an address has been returned, and halting the rest of the processes until this transfer is complete.

As with the source structs, there are two result **structs**, each one tenth the length of the same on the PPU. The SPU_Manager stores the address of the start of one such struct on the PPU. When one result struct is filled on the SPU, the SPU sends the results, switches the active storage container, and calculates the address of the next location on the PPU to send to. This is done nine times before "result full" signal and a "request new address" signal are sent. Though, as with most other communications, two Addresses are stored, an active address and a backup. Once the active address if full, the backup is made active and the replied address is used to replace the backup, another example of double buffering.

The SPU_Manager must also keep track of the first row of the aperture stored. Due to the lack of memory available to the SPU, a single 5120 entry character array is used to store the state of each aperture cell. This may or may not be enough to contain the entire aperture. If not, only the portion required for the current state in the simulation is stored. If this changes, it is the responsibility of the SPU_Manager to request the required section. Since it is unlikely that more than ten cells are crossed in the y axis (calculations on known simulations averaged out at three to four) , and since the largest resolution aperture so far simulated was fewer than 500 cells wide, only entire rows are stored. If only partial rows were stored, it would greatly increase the time and complexity of transferring aperture data. Thus full rows are stored, but only the relevant section of columns are used.

The PPU may also issue a "Halt" command to each SPU in order to process a possible backlog of results; naturally there is an associated "Start" command. The SPU_Manager is responsible for holding the SPU state on a "Halt" command and resume on a "Start" command.

8.5.3 Base_Spu

The final layer of abstraction is known as Base_Spu, and it is this layer that actually handles all calculations in the simulation. While all calculations are performed with the four word vector SIMD instruction set, this section will give a brief overview of the

simulation process, and is only discussed in a general sense. More detail as to how the calculations are carried out are given in chapter 9

The first step in the process is to create a ray object. To calculate the necessary constants, two things are required; a starting point and an ending point. It is possible to do this with by selecting a start point and specifying a gradient, however, many calculations are greatly simplified by choosing the former approach.

The start point is provided by the source struct and the end point is calculated from the position of a cell on the camera. In terms of the camera, only the location of the bottom and left edges, the number of cells per side and the dimensions of each side are stored. The Base_Spu must therefore calculate an end point for the ray from the number of the currently active cell, and the given dimensions.

Due to the four entry vector SIMD nature of the SPUs, four rays are traced at once with 16 (4 X 4) rays traced per camera cell, one row of rays at a time. The complexity of the mathematics of a line in three-dimensional space is simplified by breaking it down into a combination of two, two-dimensional lines; one on the (x,z) plane, the other on the (y,z) plane. Thus calculations may be made with the well-known formulae for straight lines:

$$y = mx + c$$

Or more specifically in this case

$$z = mx + c$$

$$z = ny + d$$

With constants *m*,*n*,*c* and *d* determined for each of the four rays, the remainder of the simulation may be broken down into three specific parts, each presenting a unique problem and each is discussed in detail in chapter 9 . The first step is to determine the amount of solid material each ray passed through. The second is to multiply the source by an attenuation constant given by

$$N = Se^{-\propto r}$$

Where *S* is the intensity of the source, *r* is the amount of material passed through, calculated in the previous step, and α is an attenuation constant dependent on the type of material used and the energy of the radiation [51].

In reality, Gamma cameras are not perfect - a single gamma ray may trigger several neighbouring cells if it does not fall in the centre of a cell. To emulate this, the final value N is distributed among the central cell and its eight neighbours according to the formula [51].

$$v = N\frac{x - \Delta x}{x}\frac{y - \Delta y}{y}$$

Where *v* is the value to be added to a cell, *N* is the value calculated in the previous step, *x* and *y* are the width and height of a cell respectively and *Δx* and *Δy* are the distances to the point from the centre of each cell in the x and y axis respectively. As an example, see Figure 11 below.

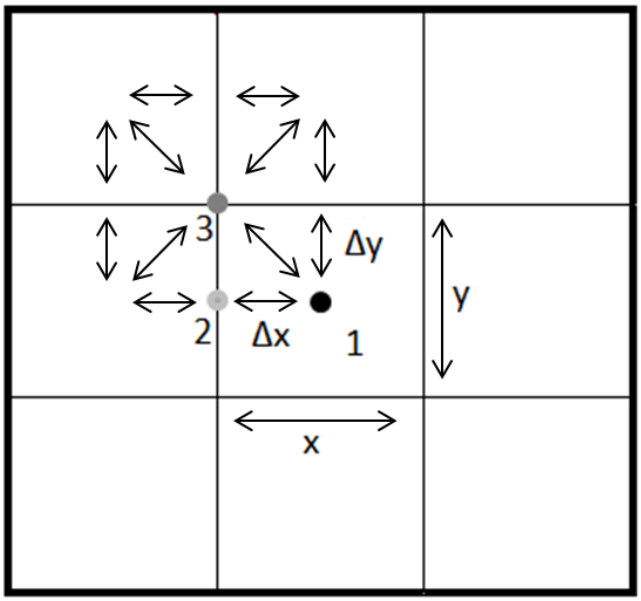

Figure 11: Illustration Of gamma camera ray distribution

A negative value indicates that the ray hit farther than one full cell length away, and would not, in reality, affect the cell, and is thus discarded if it occurs. However, the simulation is structured in such a way that these values are not calculated at all, as it would be a waste of resources.

The black dot is in the centre of the nine blocks, marked 1, has a Δx and Δy of 0, thus 100% of the value is added. For all other blocks Δx = x and Δy = y, which results in 0% being added. The light grey dot marked 2 has Δx =1/2x but Δy = 0 for both blocks. Thus 50% of the total value is added to each block. The dark grey dot marked 3 has Δx = 1/2x and Δy = 1/2y for all four block, thus 25% of the total value is added to all four blocks.

With all calculations performed, the results are added to the active results storage array, and the process is repeated on the three remaining rows on the camera cell, before again repeating the process on the next cell on the camera. Once all camera cells have been simulated, the next source block is selected and the procedure is repeated. Thus, it can be seen that the running time of a whole simulation increases exponentially with the resolution of the source and camera.

This concludes the basic structure of the SPU layer of software, for a more detailed description of the class structure, members and functions, refer to the associated doxygen documentation on the attached CD.

Having discussed the general structure of the software on the SPU, chapter 7 presents three key areas of simulation itself, and details their implementation on a SIMD architecture.

9 Coding on the SPU

Working on the SPU involves a movement from task level parallelism to instruction level, due to the SIMD nature of the processor.

The challenge is to take full advantage of the SIMD instruction set, while avoiding branching code, such as if statements and while loops. The SPU assumes sequential flow, correctly predicted branches execute in a single cycle, but an incorrectly predicted branch incurs an 18 to 19 cycle penalty [23], thus it is important to reduce the opportunities for incorrectly predicted braches. The uses and difficulties of programming in a SIMD environment will be demonstrated in two cases, before its application to near field coded-aperture simulation is examined.

9.1 SIMD Examples

9.1.1 Case 1

To demonstrate how SIMD may be used, consider the following code segment:

```
int x[4];
int y[4];
int z[4];
for (int i = 0; i < 4; i++ )
{
        z[i] =x[i] + y[i];
}
```

This is a simple piece of code, adding four corresponding values in two arrays. The code may be vectorised as demonstrated below in Figure 12. Each x value and each y value are loaded into on cell of a vector, before a single addition instruction is given.

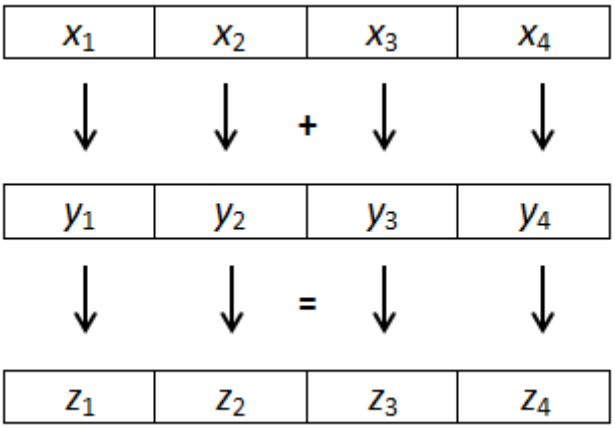

Figure 12: Vector addition

This demonstrates that four addition operations may be completed simultaneously, and will yield a four times performance gain.

However, this is a simple case, and most operations do not fit so easily into the SIMD architecture, and the programmer must often demonstrate a little creativity to solve the problem.

9.1.2 Case 2

Consider the code segment below

```
int x[4] = {6;1;9;2};
int y[4] = {9;2;7;5};
iInt z[4];
for(int i = 0; i < 4; i++)
{
        if (x[i] < y[i])
        {
                z[i] = x[i];
        }
        else
                z[i] = y[i];
}
```

Once again, this is a relatively trivial function, simply finding the lower value of two arrays in their respective places. However, this operation does not fit so neatly into vector form, as each pair must be compared separately and there is no guarantee that the results will be the same, yet the same operations must be carried out on all four values simultaneously *and* give a correct result.

One cannot simply say “if x < y”, since four values must be compared concurrently and the results of each set may differ, and running through a loop for each value in the arrays defeats the purpose of using a SIMD instruction set. This is why it is important to learn the entire instruction set, and what each instruction returns as seemingly unimportant commands can make a difficult task much easier.

The above problem can be solved with a comparison instruction and a few bitwise logic commands.

The comparison function compares each value in the x array with its counter-part in the y array. If the comparison is true, the result is a four byte word with each bit set to 1. Likewise, if the comparison is false, each bit is set to zero. This is shown below in Figure 13.

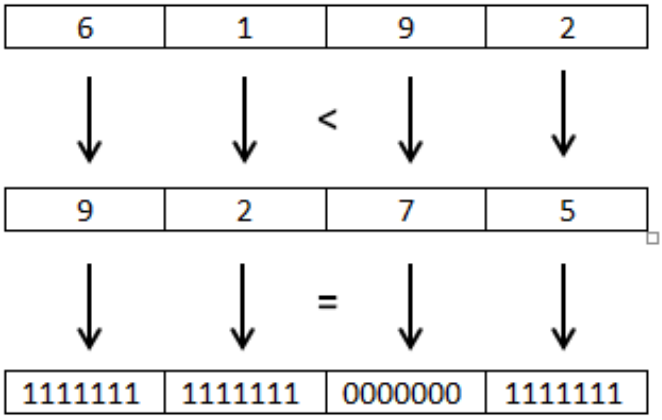

Figure 13: Vector Less Than Operation

This does not give the final answer, but it does create the tool to so, namely the binary array at the bottom of a figure. Since all the bits are set to one, a bitwise AND function will give out the same number, if all the bits are set to zero, a bitwise AND will always give zero as the answer. Thus by '**AND**ing' the x array with the binary array, as shown below in Figure 14, only the x values that were smaller than the y values are left.

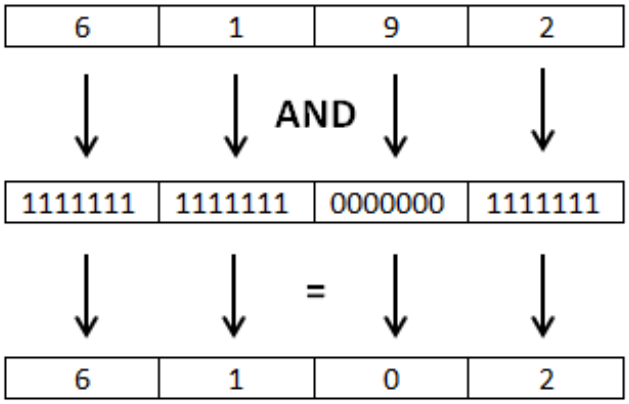

Figure 14: Vector 'AND' Operation

Now the values that are zero must come from the y array, so the binary array must be inverted with a NOT instruction to give Figure 15.

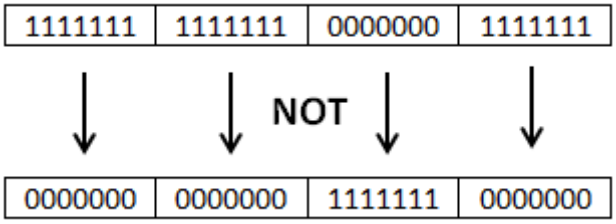

Figure 15: Vector 'NOT' Operation'

The same process that was used to on the x array is now repeated on the y array, as shown in Figure 16.

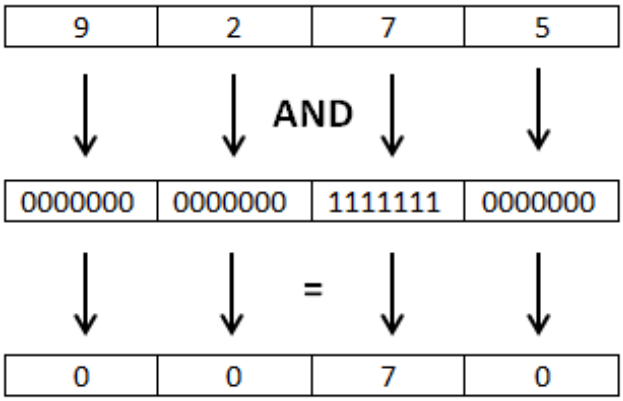

Figure 16: Second 'And' Operation

Adding the two results together, the correct answer is obtained, as shown in Figure 17.

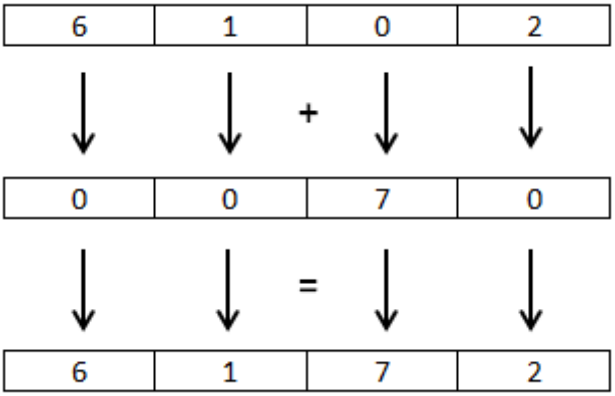

Figure 17: Result Recombination

This may seem like an overly complicated method, but bitwise operations are computationally inexpensive, and it fulfils the criteria of having no branches; exactly the same sequence of instructions is followed every time.

This procedure, or variations of it, is used extensively when calculating the amount of solid material a ray has passed through.

9.2 Distance Calculation

Probably the most important element of a near-field coded aperture simulator is the algorithm used to determine the amount of solid material a ray passes through.

The most obvious approach is to divide the aperture into n slices in the z plane, then calculate the x and y positions of the ray at each slice, then use the result to calculate

which cell of the aperture is in, and if the block is made of solid material, add Δz to a running total, where Δz is the distance between slices. This is illustrated in Figure 18.

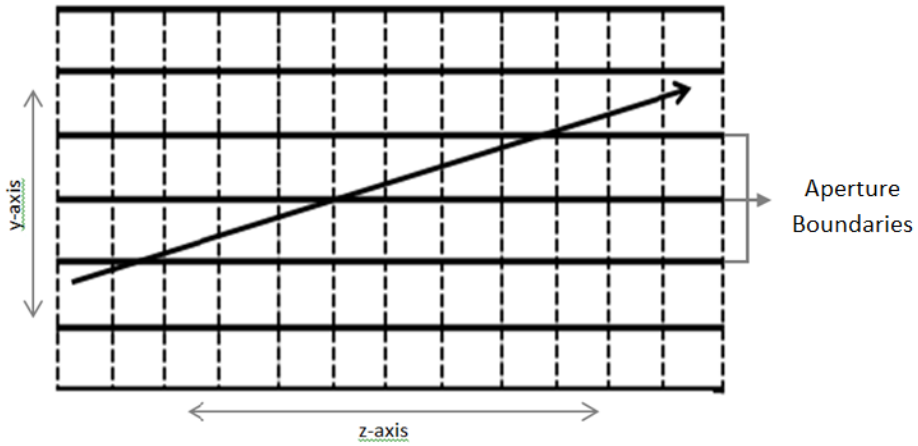

Figure 18: Cross-section of a ray passing through an aperture

The black horizontal lines represent cell borders, while the vertical, dotted lines represent the points at which a calculation is made.

Indeed, this is the method used in the first version of the simulator written in MatLab [51]. One can increase accuracy at the price of increased running time by increasing the number of slices the aperture is divided in to.

A more efficient method can be seen by viewing the aperture as a grid. A ray can only change states – into or out of solid material – at the lines of the grid, representing the sides of the cells. Thus accuracy can be increased and running time decreased by only making calculations when the ray crosses one of these lines.

$$z = mx + c$$

$$z = ny + d$$

The above equations are used to find the entry and exit points of cells. At each boundary, these positions were used to calculate which cell in the aperture the ray had moved into, still adding Δz as before, if the cell moved through was solid.

This was the algorithm used in the second, C++ version of the simulator.

The goal now is to further improve on this algorithm, and there are several areas that can easily be optimised.

The entry point and exit points to the aperture, as well as the first x and y boundary crossings of a ray can only be calculated using the linear equations above. However, once these initial calculations have been made, it can be see that because the gradients of the rays do not change, and the size of the cells do not vary, the Δz between two x or y boundaries must remain constant. Thus, Δz may be calculated once, and then added to a

running total every time. This reduces the number of operations from a multiplication and an addition to just a single addition. While this may not seem much, when repeated hundreds of billions of times, it can become quite meaningful.

Also, instead of calculating which block of the aperture enters, the actual physical process may be emulated. The aperture is stored in a single, contiguous 5120 entry character array. The signs on the gradients of ray do not change, that is, a ray with positive gradients will continue to move up and right through the aperture, a ray with negative gradients will do the opposite. If the initial point of entry is calculated, and a pointer is used to store the address, this aspect may be taken advantage of. If the ray crosses a y boundary, moving from one column to another, one may be added to or subtract from the pointer. If the ray crosses an x boundary, moving from one row to another, then the number of cells in a row is added or subtracted. This process directly simulates the ray moving through the aperture, and once again, a number of operations are reduced to just a single addition. Once the pointer has been operated on, it can simply be de-referenced to determine if the cell is made of solid material or not.

The only caveat to this method is that there must be a preliminary check to ensure that no rays exit through the side of the aperture, as this would produce erroneous results.

9.2.1 Implementation discussion

Now that a few ways have been found to further increase the efficiency of the algorithm, they must now be implemented, avoiding the use of branching code as much as possible. It is here the use of bitwise operations is invaluable. All bitwise operations, such as: less than, equal to, AND, NOR and so on are all included in the SIMD instruction set. If a condition is true, it returns a 4 byte word with all bit set to 1, if the condition is false, all bits are set to 0.

As an example, imagine calculating the Δz of a ray moving from one cell to another. If the cell is solid, then it can be multiplied by one, if not, multiplied by zero, the result then added to a total. However, in the world of digital logic, one type of variable must be multiplied by another of the same type, and the product of two 32 bit words is one 64 bit word, despite the fact that the answer can only be either the original 32 bit word, or zero. On an ordinary processor, this would not matter, but since on the SPU, all answers must fit into their original 32 bit containers to take advantage of the SIMD architecture.

The answer is simple one, instead of multiplying by one, the same effect may be achieved by performing a bitwise AND operation on a word with all bits set to 1, as in section 9.1.2. If the bits are set to 0, then this is the same as multiplying by zero.

9.2.2 Overview

The following is an overview of the steps taken to calculate the distance of solid material a ray passes through. Section 9.2.4 will present a much more detailed description of each step, however, it is a somewhat intricate process and a simpler explanation will be useful in understanding the procedure.

1. Calculate the initial aperture cell the ray enters for the pointer, the z locations of the first x and y boundary crossings and the Δz between two x and two y borders.
2. Compare the z locations of the first x boundary crossing, the first y boundary crossing, and the z location of the end of the aperture.
3. If the ray crosses an x boundary first, subtract the z location of its crossing from the z location of the entry point (ie the face of the array), de-reference the pointer and perform a bitwise AND on the two results (a de-referenced array will yield all 1's if solid). Add the result to a running total. Lastly, add Δz for the x boundaries to the position of the first x crossing. Do the same if the first crossing is a y boundary, obviously substituting the y equivalent. If it exits the back of the array, then the process is finished.
4. Update the pointer as discussed above, and repeat from step 2 until the process is finished.

9.2.3 SIMD

Now a method of determining how much solid material a ray passes through is known, but a few decisions must be made as how best to implement it within a SIMD framework. Firstly, the starting point of a ray is given, but an end point must be selected in order to calculate the necessary constants. This point will naturally fall on either the aperture or the camera. An arbitrary point could be chosen, but it makes little sense considering a well-chosen end point could simplify a great many calculations. By and large, there is little reason to consider the aperture as a valid option. It has already been demonstrated that once the initial entry block has been calculated, there are few calculation to be made, none of which are made easier by selecting the aperture as a target. In addition, more calculations would have to be run to see if a ray passing through the aperture would even hit the camera.

The camera is the more logical choice, as it allows the distribution constants to be pre-calculated and hard coded.

The next question is 'Are calculations going to be carried out on four rays at a time, or at four points on a single ray?' On a single ray, there is no way of knowing if it going to pass through a number of boundaries divisible by four before exiting the aperture, indeed, it is most likely this will not be the case, and so performance will be lost.

If four rays are simulated simultaneously, performance will likewise be wasted unless all four rays pass through the same number of boundaries. However, if 4 x 4 rays are simulated per cell in the camera, it is likely that all four rays will have nearly identical gradients, and are thus like to cross a similar number of boundaries in the aperture.

This, then, seems like the logical approach, and is the one chosen for implementation.

9.2.4 Implementation

Now that the general structure of the algorithm has been laid out, as well as how the SIMD architecture is applied, what follows is a point by point explanation of the actual implementation. Unfortunately a flowchart would be redundant, as one of the main goals of working on an SPU is to avoid branching code [23]

First, though, is a list of vector names used and their purpose. For reference, all names listed below are taken directly from the code itself.

- Z_{now} – The Current Z position of the simulation
- ΔZ – The total amount of solid material a ray has passed through in the Z axis
- Z_{next} – The Z position of the next boundary crossing
- Z_{Xnext} – The Z position of the next x boundary
- Z_{ynext} – The Z position of the next y boundary
- Z_{end} – The Z location of the end of the aperture.
- ΔZX – The distance in the z plane a ray will travel between two x bounds
- ΔZY – The distance in the z plane a ray will travel between two Y bounds
- Pointers – Keep track of the pointers to the array
- Ones – Vector filled with 1's, for bitwise operations, contains the value -1.
- Zeros – Vector Filled with 0's for bitwise operations.
- One – Used to manipulate Pointers, contains the value 1.
- Aperture_Columns - Used to manipulate Pointers

Some Cell jargon may also be useful. To "Splat" a number means to copy a single value into all four words in a vector, and to "Gather" means to add each of the four values to provide a single result. The following are the preliminary steps required for a simulation to run. Any names underlined refer to the vectors defined above.

1. Splat Ones, One, Zeros and Aperture_Columns with the appropriate numbers.
2. Splat Z_{now} with the Z position of the front position.
3. Splat Z_{end} with the Z position of the back of the aperture.
4. Splat ΔZ with 0, since the ray has crossed no material yet.
5. Calculate ΔZX and ΔZY and splat the values to the respective vectors.

6. Calculate the Z crossings of the first x and y bounds for each ray, load the values into Z_{Xnext} and Z_{Ynext}
7. Calculate the initial entry cells for each ray, and set the values in Pointers.

The next step bears some explanation. If any rays have negative gradients, then the numbers in ONE or Aperture Columns effectively need to multiplied by -1. Once again, in the digital world, there is another way of accomplishing this; finding the 2's complement. This is essentially inverting all the bits in a word and adding one. The result is the negative of the same value. Thus NOT bitwise operator is applied and one is added. However, there is the possibility that the gradient is positive, and this is not required. Once again, attempting to avoid "if statements", the following process can be followed. This technique is applied extensively in the simulation process, and so bears explanation.

1. Perform the operation m < 0 and store the result.
2. AND the result with One. If the condition is true, then One is left untouched, if not, the result is 0.
3. NOT One and add 1. If One was zero, then it flips to all 1's, adding one resets to zero. If One was not zero, then the two's complement has been found. (ie, -1)
4. Perform the operation m >= 0 and store the result.(Note, use the original m)
5. AND the result with One.
6. Add the results of 3 and 5. Since only one condition can be true, this leaves the final, correct answer, regardless of whether the gradient is positive or negative.
7. Repeat the entire process with Aperture Columns.

This is essentially the same technique as is discussed in section 9.1.2.

All the information required to simulate a single row of rays has now been calculated and loaded. The Process is as follows. In the following procedure, assume that Vectors retain their values, and the results of the operations are stored elsewhere unless explicitly stated that the value is changed.

1. $Z_{Xnext} <= Z_{Ynext}$
2. AND Z_{Xnext} with Ones.
3. $Z_{Xnext} > Z_{Ynext}$
4. AND Z_{Ynext} with Ones.
5. Add results of 2 and 4, and use result to set Z_{Next}
6. $Z_{Next} <= Z_{end}$
7. AND Z_{Next} with result from 6.
8. NOT Z_{end}
9. Add results from 7 and 8. Set the result as Z_{next}

10. Subtract Z_{now} from Z_{next}
11. De-reference pointers.
12. AND results from 11 and 10. Add result to ΔZ.
13. AND result from 1 with ONE and add to Pointers.
14. AND results from 1 with ΔZX and add to Z_{Xnext}
15. AND result from 3 with Aperture_Columns and add to Pointers.
16. AND result from 3 with ΔZY and add to Z_{Ynext}
17. Gather Z_{Next} If the result is greater than 1, return to strep 1.

Thus, throughout the entire process, only a single branch can occur, either all rays have exited the aperture, in which case all in Z_{Next} will be greater than Z_{end}, and the next simulation can begin, or there are still rays within the aperture and loop must continue. One can see that the most complex operation performed during the operation is a simple addition. The cost, however, is effort and clarity. This would be a relatively trivial task if written in even standard c++ on a conventional processor. However, on an SPU it took a great deal longer, and while technically not extremely complex, the above procedure is certainly not intuitive.

9.3 Camera Distribution

It has already been decided that the camera is a natural choice for the end points of a ray object. It also makes sense, given the four word architecture of the SIMD vectors to simulate 16 (4 * 4) rays per cell, this is shown below in Figure 19.

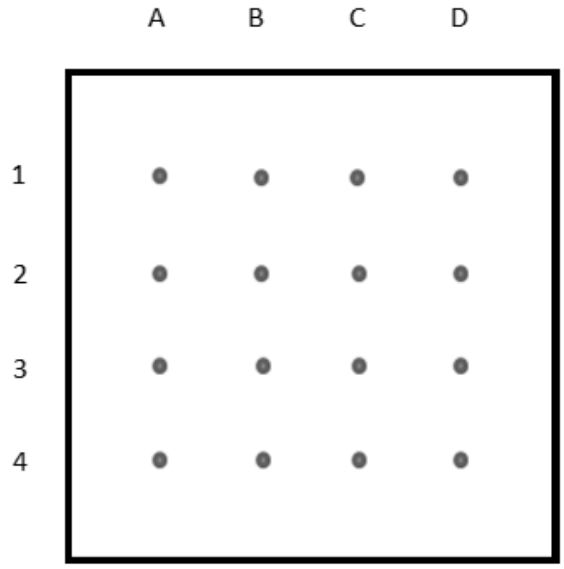

Figure 19: Ray distribution on camera cell

The decision to make the camera the end point for the rays, has two major advantages, the first is that uniformity of ray distribution is assured and the second is that the

distribution constants can be hard-coded. It is clear there are four vectors, containing the values shown below in Table 3.

Vector 1	1A	1B	1C	1D
Vector 2	2A	2B	2C	2D
Vector 3	3A	3B	3C	3D
Vector 4	4A	4B	4C	4D

Table 3: Vector Representation of Ray Distribution

Taking a closer look at the figure, there is a natural symmetry to the distribution constants. This is demonstrated in Figure 20.

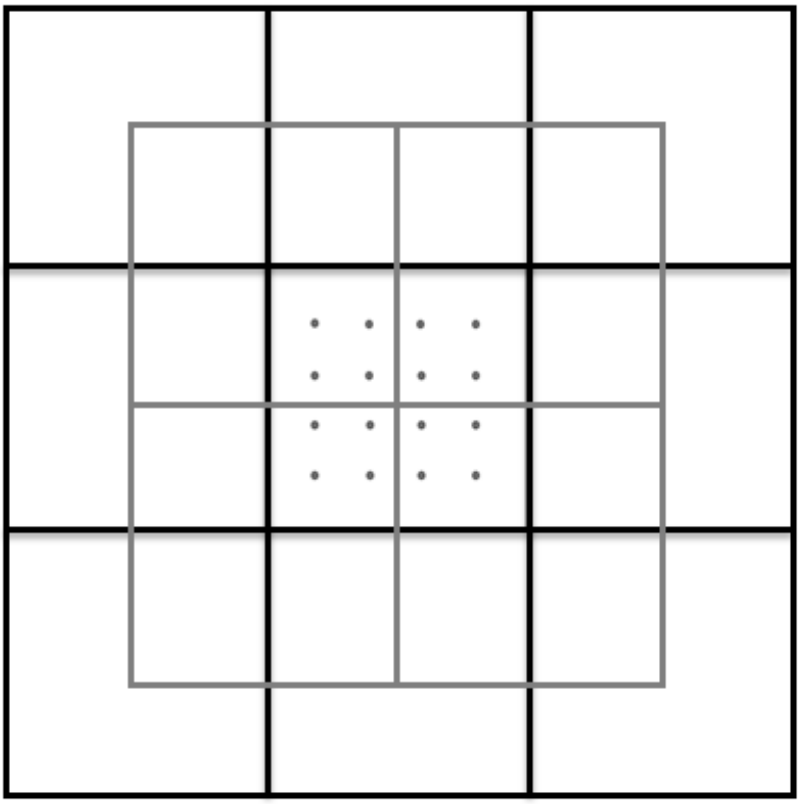

Figure 20: Illustration of symmetry of Distribution Constants

From this, it can be seen that, providentially, each dot affects four cells, and each cell is affected by four dots, this fits nicely into the natural four word architecture, however, from Table 3 above, it can be seen that the vectors are not quite right, as only two entries in a given vector affect the relevant four blocks.

A lengthy piece of code that deals with two blocks and two dots at once could be written, but a far more elegant solution presents itself, if the data is just re-arranged slightly. This is demonstrated in below in Table 4 and graphically in Figure 21.

By re-arranging the data in this way, we are left with four dots, each affecting four blocks, but the key point here is that a single vector loaded with pre-defined distribution values may be multiplied by each of the four vectors to attain a result.

Vector 1	1A -> 1A	1B -> 1B	1C -> 2B	1D -> 2A
Vector 2	2A -> 1D	2B -> 1C	2C –> 2C	2D -> 2D
Vector 3	3A -> 4D	3B -> 4C	3C -> 3C	3D -> 3D
Vector 4	4A -> 4A	4B -> 4B	4C -> 3B	4D -> 3 A

Table 4: Revised Vector Representation of Ray Distribution

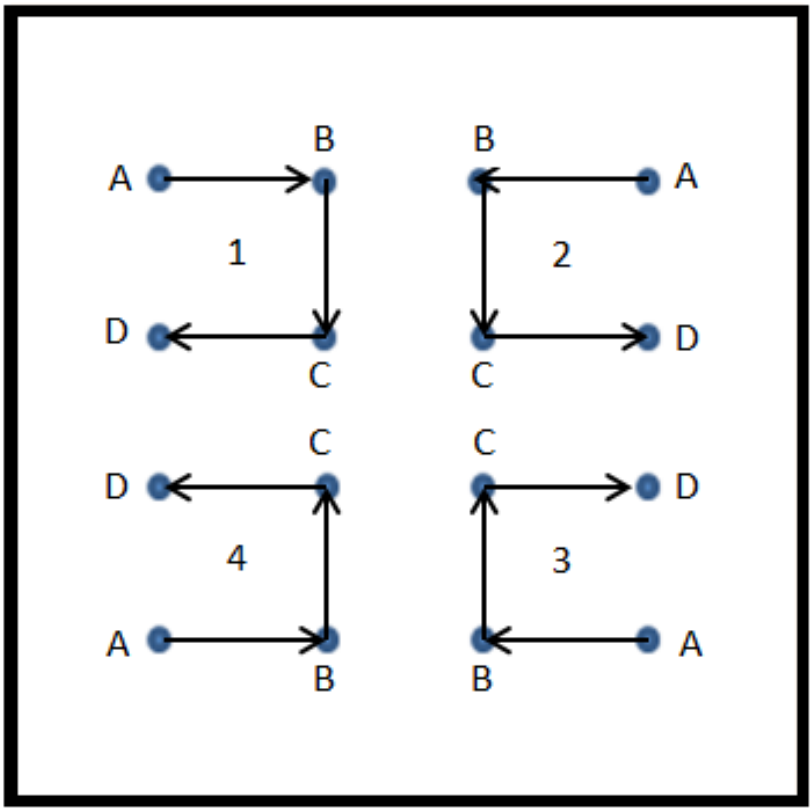

Figure 21: Reorganised Distribution Constants

Of course, each cell simulation affects only nine cells, not the 16(4 X 4) that have been calculated. The blocks must thus be recombined as shown in Figure 22.

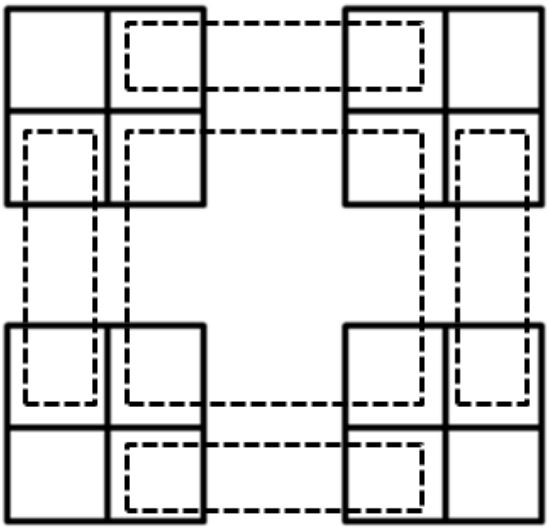

Figure 22: Camera cell Re-combination

9.4 Natural Exponent Calculations

The freeware SDK for the SPUs does not include a function for calculating natural exponentials as is required in section 8.5.3 in the form of

$$N = Se^{-\propto r}$$

Where S is the original intensity of the source, r is the amount of solid material passed through and α is constant dependant on the type of material and energy of the radiation.

There are two fundamental approaches to the problem [53, 54]. The first is to implement a recursive algorithm such as the Taylor series [53]

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

This is non-ideal, it can be seen that the above formulae contains a large number of multiplications, divisions and additions. It seems impractical to dedicate that kind computational complexity on what, in reality, is a relatively small piece of the problem.

The second method is to construct a lookup table for all values it is realistic to expect to encounter [54]. It was calculated that it is extremely unlikely to encounter a value of x < -9. It was further calculated that at least three decimal places of precision were required for the simulation to maintain accuracy. This means accounting for values between 0 and -9.999, a total of ten thousand values. Twenty thousand values for a full lookup table with a 1 to 1 correspondence, though this can be reduced if one is willing to calculate where a value is stored in relation to the start of the table. Even so, ten thousand floating point values will require 40 kB of memory. Once again, this is

unacceptable. With only 256 kB to work with, dedicating such a large portion of memory to the problem is just as unrealistic as implementing a recursive algorithm. Of course, this table could be stored in the local memory of the PPU, but that would require a request to and a response from the PPU, which has already been established to be a possible bottleneck.

The solution is again a simple one, and can be found in basic algebra. If 1.234 is written as 1.2 + 0.034, then the same can be done in the exponential, which leads to the following

$$e^{1.234} = e^{1.2+0.034} = e^{1.2}.e^{.0.034}$$

The table can therefore be broken into two parts, one containing values for the range of $e^{0.0}$ to $e^{-9.9}$, the other storing $e^{0.000}$ to $e^{-0.099}$. In this format, each table would have to store only 100 values each, for a total of 200 with a single multiplication required to join the two. 200 values correspond to 800 bytes of memory, which is perfectly acceptable. Of course, the tables could be broken down further into four tables of just ten entries each, however it was judged that space saved did not warrant two additional multiplication procedures.

Trials on a PC have shown that this method actually yields similar performance to a single table. The time taken for a single multiplication is apparently comparable to retrieving a value from a table that large.

Thus given a seemingly intractable problem, a simple, elegant solution has been found that may even benefit other, more traditional processing architectures.

9.5 Conclusion

Developing on the SPU is more reminiscent of developing on small, embedded microchips than on a fully fledged PCU. While they present a few limitations, especially in the amount of memory, the SPU's allow the Cell architecture a performance gain in the orders of magnitude of traditional, sequential processing units [23]. However, there is a steep learning curve involved, and a great deal of creativity is sometimes required to take full advantage of the processing power available.

Having seen a view of the finer details of the simulator, chapter 8 presents the findings of this dissertation, including a performance breakdown of the simulator, the conclusions of the project, and recommendations for future work.

10 Results

As discussed previously in section 8.4.1, floating point communication between the PPUs and the PC prevented the project as a whole from being completed and a number of systems, such as the final combining of the results on the PC were never implemented. However, these are areas of low computational intensity, and are unlikely to affect the final performance of the system. The core components, those associated with the high computational cost and prohibitive running times of a near field coded aperture simulator, were successfully implemented. The results of which are discussed below.

10.1 Performance Testing

While no complete simulations were run, enough functionality was established to run a comparison to the original PC counterpart. The results were partially dependant on the geometry of the coded aperture with relative performance increasing with thicker aperture plates and higher resolution patterns. Thicker apertures and more complex patterns increase the ratio of number of calculations done to communication between processing elements and memory transfers. Using configurations similar to those demonstrated in [51], it was found that the Cell processor could perform all necessary calculations on a single ray at approximately the expected 40 times performance gain seen in the Pi estimation application. This is demonstrated visually below in Figure 23: Relative performance of simulator versions. which shows the relative performance of the original Matlab simulator, its C++ counterpart and the CBEA simulator. Note the logarithmic scale of the y axis.

This figure is a little misleading, as most previous simulations ran at 3 x3 rays per cell of the camera, while the PS3 runs at a constant 4 x4 rays per cell, due to the nature of the SIMD architecture. Thus simulations will yield a $40\,\frac{9}{16} = 22.5$ times speed enhancement. Nevertheless, simulations which once took almost a day to run (and nearly 2 weeks on six machines before that) could now be completed in little under an hour on a single machine; twelve minutes running on the cluster. Additionally, there is a slight improvement in accuracy, and the option to easily add more nodes if required. The idea of running fifty slides of a three dimensional simulation is now quite plausible.

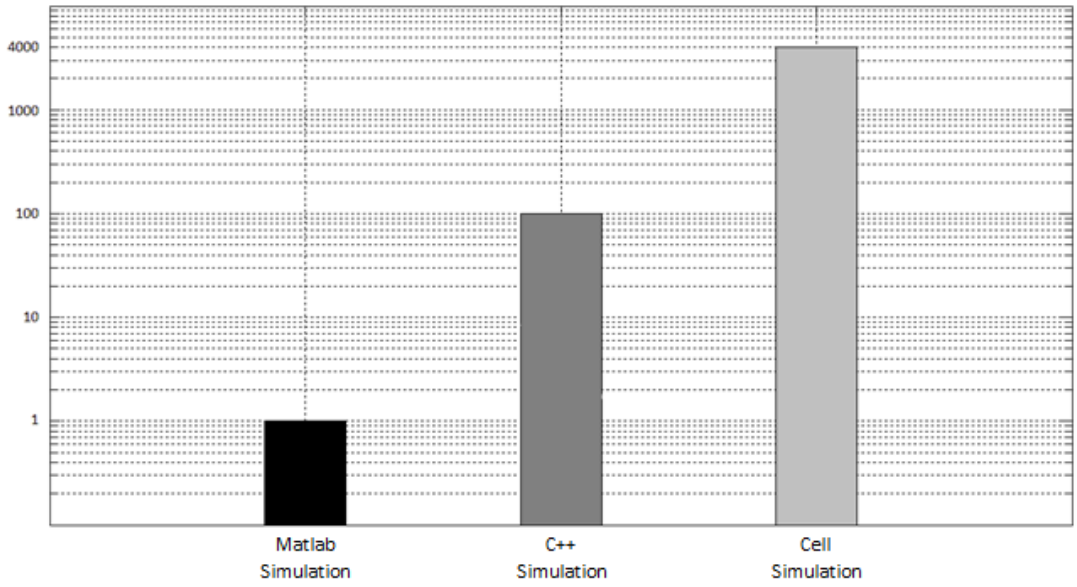

Figure 23: Relative performance of simulator versions.

10.2 Performance Breakdown

Due to the complex and sometimes unpredictable nature of a parallel application, it is vital to be able to find any unexpected bottlenecks that might hamper performance. In order to accomplish this, four main areas in the code were identified.

1. The generation of ray objects and their associated constants.
2. Calculating the total amount of solid material a ray passed through in both the X and Y axes.
3. The Conversion from distances in X and Y axes to actual distance of material the ray passed through.
4. Using the above figure and the initial intensity of the ray, calculating the attenuation of the ray.
5. Calculating the distribution of 16 rays to the targeted cell and its neighbours.

In order to identify the proportion of time each of the processes took, the following method was used.

1. Run a single, complete ray calculation in a "**for** loop" with enough iterations that it takes at least a few milliseconds, long enough for the clock function to accurately record any changes in time. This serves as a control time.
2. Each of the above 5 points represent separate function calls. The total running time of the process can then be broken down into the following equation.

$$T_1 = N + X$$

3. Where T_1 is the control time, N is the time of one specific function, and X is the time represented by the remaining four functions. Placing the target function in another for loop for another 101 iterations gives

$$T_2 = 101N + X$$

4. Subtracting T_1 from T_2 gives

$$T_2 - T_1 = 101N - N + X - X$$

5. Which yields

$$\Delta T = 100N$$

And N can be solved.

6. The percentage of the running time that N takes to execute can then be found by

$$\% \, Runtime = \frac{N}{T_1} \times 100$$

7. This process is repeated for each of the five functions listed above. The total should make up the bulk of run time, if not the same process can be used to narrow down the section of code acting as a bottleneck and remedy it.

The results of the above process on the five main functions are approximately

1. 10%
2. 40%
3. < 1%
4. 20%
5. 20%

It can be seen that the total comes to about 90% of the total running time, which is a good indication that there is very little wasted time, and that the computational power of the chip is being applied where it is needed.

Since no complete simulations were run, the accuracy of the results was verified both by hand calculation and comparison to a small sectional results created by the original simulators.

10.3 Discussion on Cell Broadband Engine Architecture

10.3.1 The Positive

Essentially, the Cell delivers what was promised. There is no doubt that as a processor it has at least 40 times the raw computing power of other standard, comparatively priced processors, while maintaining the flexibility that Graphic Processing Units lack. It has all the advantages of a distributed memory system, while the high speed Element Interconnect Bus and Memory Flow Controller units allow a skilled programmer to hide the time traditionally lost to memory latency in such systems. It also has a large, shared

address space. Thus, it combines the best aspects of distributed and shared memory models of parallelism. The Mailbox and Signal notification channels provide an excellent means of communicating commands or of synchronisation. The bitwise 'OR' mode of writing to a Signal notification channel proved particularly useful, as it allows one to queue up commands without having to worry about overwriting previously written ones. Although not specifically required in this application, the ability to switch contexts (that is, essentially loading a new program onto the SPU) on the fly will be invaluable in dynamic load balancing. An SPU application for each for each parallelisable task in a program must be written, and then simply loaded needed, allowing for the optimisation of a specific task in an assembly line fashion, but with the ability to adapt to bottlenecks as and when they occur. Overall, the Cell provides a number of original and useful innovations.

10.3.2 The Negative

The Sony PS3, while appearing to be a near perfect platform for scientific computing due to its low cost to computational power ratio has a number of weaknesses that prevent it from living up to its potential. The 256Mb of ram it has is simply not enough to run modern operating systems and development environments. As seen in section 5.5.2, this means coding and debugging through a text editor and a console, a process that has been obsolete for at least a decade. The difficulties of debugging through a console by writing text to the screen are compounded by the parallel nature of the chip, and it is nearly impossible to judge the timing of events, a critical component of any parallel application. In addition, including the libraries required to write text to the screen in the SPU code take up valuable space in its 256kB cache.

Also, the somewhat erratic behaviour and interactions of freeware operating systems and applications make the PS3 an inherently unstable platform to develop for. Finally, the ability to install an operating system over the game OS has recently been removed in the firmware. Ultimately, using the PS3 for scientific computing an unfeasible undertaking, a conclusion supported by [10].

The Cell is still a very new technology, and as such it still has a number of weaknesses. Often, the most difficult part of working on the Cell was simply the lack of documentation and support. As an example, the provided documentation made reference to the fact that the inbound Signal Notification Channel had an 'or' mode, but made no mention of how to enable it. It was eventually found, in a log detailing the changes between SDK versions 2.1 and 3.0, that a specific flag must be set when initialising the SPU, and while it described the register in which the flag bit was located, no mention was made of its actual position in the register, and so was found by trial and error. There are also places where processes are counter-intuitive: in order to initiate a DMA transfer, the address of a variable containing the address of the data to be sent must be supplied, rather than just the address containing the data. As discussed in

section 6.2, there is 5 bit tag, identifying the transfer. There is a 32 bit mask (5 bits = 32 possible values) which can be set to hide or show if DMA commands with the appropriate tags are complete or not, one might expect this mask to initially be set to look for all tags when, in fact, the reverse is true. These examples are by no means the full extent of the Cells idiosyncrasies, but they do serve to demonstrate the point. These complaints are not specific to the Cell itself, only to the general lack of documentation and knowledge in the community.

Working on the SPU's is more reminiscent of working on a microprocessor in Assembly language than on a CPU in C++. Indeed, many of the instructions directly related to the SPU are one-to-one conversions from Assembly to C++ [23]. Each SPU has 128 128-bit registers used for SIMD vector instructions. Because of this, care must be taken when aligning data, and must often be managed manually. DMA transfers also require the data to be sixteen bit aligned.

The real challenge represented by the chip itself is in the SIMD instruction set. Taking full advantage of operating on four data points at once, while attempting to avoid 'if statements' and 'while loops' certainly requires a little more creativity than traditional, sequential, coding. Taking this into account with the parallel nature of environment and potentially asynchronous memory transfers without the benefit of development environment and a de-bug mode, the whole process can become somewhat interesting.

While all three layers of software were implemented, tested and fully functional, barring some of the code on the PC required to recombine all results, it was the communication layer between the PC and the PPU's - the MPI interface - that failed. The reduced number of supported data types may well grant enhanced performance, but it also means the Cell may be incompatible with a number of common libraries and applications, such as MPI.

10.4 Recommendations for Future work

As discussed in section 8.4.1, should the decision be made to complete the work presented in this dissertation, the recommended starting point would be to revert the inter-nodal communication from MPI to the earlier PRV protocol.

Should the problem of a high speed, near field coded aperture simulator be revisited, the minimal amount of node intercommunication required and the high degree of processor independence, characteristic of all embarrassingly parallel applications, makes this an ideal candidate for grid computing. While the techniques described in chapter 9 have been adapted for SIMD computation, they should still provide a solid foundation for an efficient grid based simulator.

11 Conclusion

With a very high performance to cost ratio, the Sony PlayStation 3 sounds like an ideal platform for scientific computing. However, the relatively small amount of memory as well as the lack of standardisation and support detracts greatly from its appeal.

Similarly, the Cell Broadband Engine Architecture offers a lot of promise. The combination of a small, local memory cache with a larger shared address space with dedicated, on-chip memory transfer modules goes a long way to solving the problems of inter-communication between nodes in a parallel environment. The stripped down Synergistic Processing Elements offer floating point calculation performance previously only seen on GPUs, but without sacrificing the flexibility of a standard CPU. Unfortunately, the Cell is not without fault. The cost of performance is effort and clarity. The SIMD instruction set is powerful, but unwieldy, requiring the developer to carefully manage and manually align data. Avoiding branching code increases performance, but is often difficult and produces code that is very cumbersome for others to read and understand. Finally, the Cell is still a very new technology, and suffers a few unintuitive idiosyncrasies and a general lack of documentation and support. Generally however, for the purposes of scientific or academic computing, the benefits far outweigh the drawbacks.

Additionally, a second generation of CBEA processors is on the horizon, optimised for a 64 bit word architecture. It seems reasonable to assume that a second, more mature iteration of the Cell processor will have smoothed many of the original eccentricities, while documentation and support will become more widespread.

The qualities of the Cell processor naturally reflect the qualities of parallel processing as a whole. The technology is still new, and a number of issues have not been adequately resolved for general use. The difficulties of memory management, work allocation and synchronisation in a parallel environment are currently too complex to be dealt with in a complier and must be hand coded. In this new form of processing, development tools and practices have effectively been regressed by a decade or more. What's more, not all problems can benefit from parallelisation. The potential gains are limited by what percentage of a given problem is inherently serial in nature, as described in Amdahl's Law [19].

Under the correct circumstances, however, the results are impressive. It has been shown that the Sony PlayStation 3 can achieve a 40 fold gain in performance over similarly priced processors with a more traditional architecture. In addition, this figure is scalable. Unlike serial processors, more nodes may be added to a cluster more or less at will.

Like the relationship between modern, quantum physics and its older Newtonian counterpart, parallel processing is not yet ready for general use in all facets of computing, but it does provide a powerful tool for solving problems that warrant its cost.

References

12 Bibliography

[1] T. Garrison, Oceanography: An Invitation to Marine Science, 5th ed, Thompson Brooks/Cole, 2005.

[2] Bureau International des Poids et Mesures, “The International system of units (SI)” 2006.

[3] W. G. Hoover, “Smooth Particle Applied Mechanics: The State of the Art,” *World Scientific,* no. 25, 2006.

[4] J. Steinhoff, Vorticity Confinement: A New Technique for Computing Vortex Dominated Flows, John Wiley & Sons, 1994.

[5] P. Yuan and L. Schaefer, “Equations of State in a lattice Boltzmann model,” *Physics of Fluids,* vol. 18, 2006.

[6] H. Fliegel and R. DiEsposti, “GPS AND RELATIVITY: AN ENGINEERING OVERVIEW,” *Precise Time and Time Interval,* vol. 28, 1996.

[7] S. Adve et al., “Parallel Computing Research at Illinois: The UPCRC Agenda,” 2008.

[8] K. Asanovic, R. Bodik, K. Keutzer et al., “The Landscape of Parallel Computing Research: A,” EECS, 2006.

[9] O. Erik, E. Sevre, M. Christiansen et al., “Experiments in scientific computation on the PlayStation 3,” *Visual Geosciences,* vol. 13, pp. 125-132, June 2008.

[10] A. Buttari, P. Luszczek et al., “A Rough Guide to Scientific Computing On the PlayStation 3,” 2007.

[11] W. T. Moye, “ENIAC: The Army-Sponsored Revolution,” January 1996. [Online]. Available: http://bandwidthco.com/history/eniac/ENIAC%20-%20The%20Army%20Sponsored%20Revolution.pdf. [Accessed 11 January 2011].

[12] D. Bodanis, Electric Universe, Crown Publishers, 2005.

[13] B. Winston, Media technology and society: a history: from the telegraph to the Internet, Routeledge, 1998.

[14] K. Hafner, Where Wizards Stay Up Late: The Origins Of The Internet, Simon and Schuster Paperbacks, 1998.

[15] M. Malone, The Microprocessor: A Biography, vol. 1, Springer, 1995.

[16] “The top 500 Supercomputing Sites,” November 2003. [Online]. Available: http://www.top500.org/list/2003/11/. [Accessed jANUARY 2011].

[17] L. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 2002.

[18] J. Rabaey, Digital Integrated Circuits, Prentice Hall, 1996..

[19] M. Hill and M. Marty, “Amdahl's Law in the Multicore Era,” Published by the IEEE Computer Society, July 2008. [Online]. Available: http://www.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf. [Accessed 12 January 2011].

[20] A. Grama, Introduction to Parallel Computing, 2 ed., Addison-Wesley, 2003.

[21] R. Buyya, “Parallel Computing at a Glance,” 2000.

[22] M. Gardner, “Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life",” *Scientific American,* pp. 120-123, October 1970.

[23] IBM, *Cell Broadband Engine Programming Handbook,* IBM, 2009.

[24] E. Korpela, A. Werthimer and e. al, “SETI@HOME—MASSIVELY DISTRIBUTED COMPUTING FOR SETI,” *COMPUTING IN SCIENCE & ENGINEERING,* pp. 78-83, January/February 2001.

[25] B. e. a. Purcell, “Ray Tracing on Programmable Graphics Hardware,” *ACM Transactions on Graphics,* vol. 21, pp. 703 - 712, 2002.

[26] J. Owens, D. Luebke and e. al, “A Survey of General-Purpose Computation on Graphics Hardware.,” *State of the Art Reports,* 2005.

[27] N. Thrane and L. O. Simonsen, “A comparison of Acceleration Structures for GPU Assisted Ray Tracing.,” M.S. thesis, Unversity of Aarhus, 2005.

[28] T. Whitted, “An Improved Illumination Model for Shaded Display,” *Communications of the ACM,* 1980.

[29] N. A. Carr, J. Hall and J. C. Hart, “The Ray Engine,” *Proceedings of the ACM SIG-Grapgh/ Eurographics conference on Graphics hardware,* pp. 37-46, 2002.

[30] T. Purcell, *Ray Tracing on a stream processor,* Phd Thesis, 2004.

[31] C. Martin, *Ray Tracing on GPU - Diploma Thesis,* University of Applied Sciences Basel , 2005.

[32] IBM, “Cell BE architecture,” October 2006. [Online]. Available:

http://cell.scei.co.jp/pdf/CBE_Architecture_v101.pdf. [Accessed 25 12 2012].

[33] University of Massachusetts Dartmouth, “Playstation 3 Gravity grid,” [Online]. Available: http://gravity.phy.umassd.edu/ps3.html. [Accessed 11 11 2012].

[34] IBM, *IBM 1961 BRL Report,* 1961.

[35] M. Warren, J. Salmon and e. al, “Pentium Pro Inside: I. A Treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac,” IEEE, 1997.

[36] University of Kentucky, “University Of Kentucky Supercomputer Breaks The $100 Per GFLOPS Barrier,” August 2003. [Online]. Available: http://aggregate.org/KASY0/press.html.

[37] S. Graham, M. Snir et a.l, “Getting up to speed: The future of supercomputing,” THE NATIONAL ACADEMIES PRESS, 2005.

[38] M. Garrels, “Init run levels,” in *Introduction to Linux: A hands on Guide*, 2008, CreateSpace Independent Publishing Platform.

[39] “Oracle VM VirtualBox®: User Manual,” 2011.

[40] W. Gropp, E. Lusk and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed., MIT Press, 1994.

[41] 2009. Lam-MPI home page. Available: http://www.lam-mpi.org/.

[42] 2011. Open-Mpi home page. Available: http://www.open-mpi.org/.

[43] Network Working Group , “The Secure Shell (SSH) Authentication Protocol,” 2006.

[44] M. Anoop, “Public Key Cryptography: Applications Algorithms and Mathematical Explanations”.

[45] IBM, “Innovation Matters,” 23 January 2009. [Online]. Available: http://domino.research.ibm.com/comm/research.nsf/pages/d.compsci.brochure2006.html. [Accessed 13 December 2010].

[46] M. Gschwind, H. Hofstee, B. Flachs et al., “Synegystic Processing in Cells Multicore Architechure,” IEEE Computer Society, March/April 2006. [Online]. Available: http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf. [Accessed 13 12 2010].

[47] J. Liu, Monte Carlo Strategies on Scientific Computing, Springer, 2008.

[48] R. Henkin, D. Bova et al., Nuclear Medicine, 2nd ed., Mosby, 2006.

[49] P. Maoa, F. Harrison and e. al, “Development of grazing incidence multilayer mirrors for hard X-ray focusing telescopes”.

[50] J. In ’ t Zand, “Coded aperture camera imaging concept,” 1996. [Online]. Available: http://astrophysics.gsfc.nasa.gov/cai/coded_intr.html. [Accessed 26 12 2012].

[51] D. M. Starfield, “Towards clinically useful coded apertures for planar nuclear medicine imaging,” Ph.D Thesis, University of the Witwatersrand, 2009.

[52] P. Macklin, 2011. [Online]. Available: http://easybmp.sourceforge.net/.

[53] J.-M. Muller, Elementary functions, 2nd ed., Birkhäuser , 2006.

[54] J. Hart, Computer Approximations, Wiley, 1968.

[55] R. Vuducy, A. Chandramowlishwaran et al., “On the Limits of GPU Acceleration,” 2010.