1-1-2011

Memory Access Patterns for Cellular Automata Using GPGPUs

James Michael Balasalle
University of Denver

Memory Access Patterns for Cellular Automata Using GPGPUs

A Thesis

Presented to

the Faculty of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

James Balasalle

March 2011

Advisor: Dr. Matthew J Rutherford

Author: James M Balasalle
Memory Access Patterns for Cellular Automata Using GPGPUs
Advisor: Dr. Matthew J Rutherford
Degree Date: March 2011

ABSTRACT

Today's graphical processing units have hundreds of individual processing cores that can be used for general purpose computation of mathematical and scientific problems. Due to their hardware architecture, these devices are especially effective when solving problems that exhibit a high degree of spatial locality. Cellular automata use small, local neighborhoods to determine successive states of individual elements and therefore, provide an excellent opportunity for the application of general purpose GPU computing. However, the GPU presents a challenging environment because it lacks many of the features of traditional CPUs, such as automatic, on-chip caching of data. To fully realize the potential of a GPU, specialized memory techniques and patterns must be employed to account for their unique architecture. Several techniques are presented which not only dramatically improve performance, but, in many cases, also simplify implementation. Many of the approaches discussed relate to the organization of data in memory or patterns for accessing that data, while others detail methods of increasing the computation to memory access ratio. The ideas presented are generic, and applicable to cellular automata models as a whole. Example implementations are given for several problems, including the Game of Life and Gaussian blurring, while performance characteristics, such as instruction and memory accesses counts, are analyzed and compared. A case study is detailed, showing the effectiveness of the various techniques when applied to a larger, real-world problem. Lastly, the reasoning behind each of the improvements is explained, providing general guidelines for determining when a given technique will be most and least effective.

Acknowledgments

I sincerely wish to thank my advisor, Dr. Matthew Rutherford, for not only his continued patience, but also a strong interest in my research, academic career, and professional future.

I would also like to express my gratitude to Dr. Mario Lopez whose enthusiasm for teaching made the transition back to academia much easier.

Dr. Paulius Micikevicius provided a significant amount of technical help and guidance related to Nvidia graphics cards; his feedback made it possible to pass through some difficult barriers.

Lastly, I want to acknowledge my wife, Aileen. Without her support, encouragement and reassurance, none of this would be possible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

General Purpose Graphics Processing Unit (GPGPU) computation is quickly becoming an important area in the field of high performance computing. However, the use of a GPU for general computation requires a significant change not only in algorithm development but also in programming environments. This work describes several performance improving techniques and patterns that apply when implementing algorithms for GPGPU environments. Specifically, the performance improvements presented here apply to problems normally organized in the form of a Cellular Automaton.

Problems modeled using cellular automata were chosen because they span a wide range of disciplines: economics[15], biology[10], cryptography[25], and imaging[32], as well as many others. The work presented here does not include algorithms for implementing specific cellular automata, but it presents generic methods applicable to the implementation of all cellular automata using GPGPUs. The main focus is on data organization and patterns for accessing that data. Because the execution environment of GPGPUs is considerably different from that of traditional processors, alternative methods for simple operations like memory access must be employed to fully realize

the capabilities of the underlying hardware. Memory access is a fundamental aspect of all computing problems, and the optimization of such can be beneficial to memory intensive applications such as those modeled by cellular automata.b In addition, the price of modern GPUs is extremely attractive, as they can be several thousand times cheaper than more traditional high performance computing platforms. These factors are the motivating force behind the work presented here. It is clear that GPGPU processing can enable researchers, scientists, and engineers to achieve a level of performance previously attainable only by governments and large corporations. The main goal of this work is to aid in the advancement of science and technology by increasing computational power without a significant cost.

Cellular automata models are the backdrop for this work because, as stated above, they span an wide range of disciplines. The main focus is to enable as many people as possible to benefit, not just those in a specific area. Also, cellular automata naturally lend themselves to parallel processing due to their state based definitions and neighbor calculations. Lastly, cellular automata models are intended to model complex systems with relatively *simple* rules. This allows us to focus on cellular automata theory and techniques that apply to it as whole, instead of being mired in the details of complex systems.

Many cellular automata are computationally simple, and because of that the main performance hurdle for implementing them is memory throughput. A large portion of this work is dedicated to describing certain techniques that improve memory performance. To illustrate that these techniques do, in fact, increase performance they have been implemented in various cellular automata models. Three different models have been implemented: Conway's Game of Life [6], elements of Lowe's SIFT algorithm [23], and finally a "real world" application that models surface water flow initially presented by Parsons in his Masters Thesis [30]. A baseline implementation

is presented for each CA, successive techniques are applied, and execution time as well as memory throughput are compared. Theoretical details of each technique are also presented and the different problems are classified by their arithmetic intensity.

A major concentration of ideas presented here is related to the ratio of the number of memory accesses to the number of instructions. Due to the hardware platform and execution environment this ratio can be an important in optimizing performance. They key contribution made by this work is the analysis of the memory-access-to-instruction ratio and the presentation of ideas that minimize the number of memory accesses that must be made. Properties of generic cellular automata are exploited to realize certain improvements. There are two central ideas that repeat in this work: 1) memory organization to increase alignment and 2) reduce the number of instructions and compute multiple elements whenever possible. Ultimately, however, this work aims to present a set of improvements that can be applied across a wide range of disciplines and applications.

In the short term, the methods presented in this work are intended to aid researchers, scientists, and engineers in achieving real and significant performance improvements. These improvements will allow a much faster turn-around time on computational runs or allow the processing of more data in an acceptable amount of time to reach more accurate results. In many cases it is possible to modify existing applications or create new applications that utilize GPGPUs in that short term; that is, within two or three years.

A longer term benefit of this work is the study of cellular automata in a parallel processing environment. CA models are both simple and widely applicable, and thus attractive to a broad audience. Modeling complex systems as cellular automata has many benefits, one of these being ease of simulation. Cellular automata models are inherently parallelizable; as the number parallel processors continues to grow and the

research community begins to move in a more parallel direction, the performance improvements presented herein will become increasingly important.

This body of research is presented as follows. First, in Section 2, a substantial amount of background is given: parallel computing, GPU architecture, cellular automata, Conway's Game of Life, Lowe's SIFT algorithm, and Parsons' surface water model are all discussed in detail. Relevant work is then discussed and investigated in Section 3: similar research is presented and analyzed and parallels are drawn, where applicable, to this work. Next, in Section 4, the two central themes are expounded upon, those being memory organization and multiple data processing. During the course of describing these two themes, tangible techniques are presented and discussed in detail. Once a sufficient understanding of the techniques has been achieved their performance characteristics are presented in Section 5. This is done through experimental comparison of different implementations of each technique. Performance improvements are also given. Section 6 details a case study, in which the techniques and patterns are applied to a real-world problem. Lastly, Section 7 concludes this work with some higher-level discussion of certain observations made while completing this research.

Chapter 2

Background

2.1 Parallel Computing

Moore's Law states that the number of transistors that can be placed on an microchip, in a non-cost-prohibitive manner, will double every two years [36]. The first 1GHz processor was released in March of 2000, approximately 10 years ago. One may wonder why, then, do we not yet have 32GHz processors? Transistor count does not equate to processor clock rate. More transistors often means more features, not just more speed. In the last few years, new transistor space has been primarily dedicated to improving parallel execution. Modern processors have multiple sources of parallelism, hyper-threading and multicore architectures being chief among them [19]. A multicore processor is one that contains two or more full, but independent, standard processors on a single chip die. The different cores may or may not share the memory subsystem, caches, or system resources depending on their design and intended use. Multicore processors are becoming more prevalent and the number of cores on a single chip die continues to increase. At the time of this writing, 6-core processors are available for desktop use. The growth of multicore processors is also present in embedded systems

applications. Processors such as the XMos [34] continue to bring multicore solutions to resource constrained environments. The drawback to such processors is that to reap their benefits, applications must be written (or modified) to specifically harness the extra processing power, and a high percentage of today's software does not do this.

In addition to multicore processors, there is also a class of newer processors called *many-core* processors. These processors have on the order of hundreds of individual cores. The main difference between these and desktop multicore processors is that many-core processors are limited in both their capabilities and independent execution [26] [37] [38]. Modern graphics processing units (GPU) are many-core processors. GPUs are extremely efficient at floating point math used for graphical computations such as pixel shading; however there are many other applications that require floating point arithmetic. In the last half of the 2000 - 2009 decade, researchers have been trying to use the inherent parallelism of GPUs to solve problems other than pixel shading. Initially this was done by attempting to deceive the hardware into thinking it was still doing graphical computations by packing data into already defined structures and executing computation by using existing graphics APIs [29]. While this method did garner notable performance improvements, it was not an easy or maintainable solution. In 2007 Nvidia released a software development kit that enabled the use of their GPUs specifically for non-graphical computation, and with this release the field of General Purpose (GPGPU) computing was born [21]. They released a framework, called CUDA (Compute Unified Device Architecture), that allows developers to use the GPU as a computation engine without having to use the existing graphics APIs: OpenGL [49] or DirectX [2] [3]. Current Nvidia cards offer over a teraflop [41] of computing power at a reasonable price, while the limit of an Intel hexa-core CPU is 108 gigaflops [47]; an order of magnitude less than the Nvidia GPU.

Moore's Law also applies to GPUs. The parallelism achieved by these processors will only continue to increase. It is clear that as processors move towards a higher level of parallelism, the software that runs on them must also evolve to keep pace.

2.1.1 Supercomputers

The definition of a *supercomputer* is constantly changing. Generally, it implies one of the "fastest" computers in the world. Typically these are inordinately expensive machines funded by governments and used to solve simulation problems in the fields of nuclear physics or astrophysics. A project called The *TOP500* tracks and ranks the current fastest computers in the world. Figure 2.1 depicts the current # 2 machine on the TOP500 list: Jaguar. It is not surprising that every single one of the computers currently on this list utilizes more than one processing core. Some computers have hundreds of thousands of standard CPUs, while others are a hybrid of CPU and GPU cores. Several of these machines break the petaflop barrier: one quadrillion floating point operations per second. Because many of these computers are 1) built for solving a specific problem and 2) cost prohibitive they are not a focus of this work. However, since many of these machine utilize GPUs as building blocks for their computations the work presented here is still relevant.

2.1.2 Parallelism via Interconnected Independent Machines

Parallelism does not have to be achieved via transistors and hardware. Multiple, independent computers can be used to solve a problem in parallel. A classic and well known example of this is the SETI@Home project [1]. The SETI@home project allows any person with a computer connected to the Internet to donate the unused CPU cycles of that computer to be used in the search for extraterrestrial intelligence.

Figure 2.1: The # 2 super computer on the Top500 list: Jaguar

Through software, the SETI@Home project is able to achieve almost a petaflop level of computing power without the need for expensive hardware. SETI@Home, and projects like it, utilize distributed computing to increase computational throughput and increase parallelism. Grid computing is another example of utilizing multiple machines to increase parallelism and ultimately performance [8]. Typically a computing *grid* is a variegated collection of machines used in concert to accomplish either a single task or a group of related or unrelated tasks. Globus [12] is a specific software system used to enable the construction of computing grids.

Clusters

Clusters use the same principles of distributed computing as SETI@home to increase parallelism. A cluster is a group of independent machines that are interconnected via

a fast medium, usually a high-speed network such as InfiniBand [22], etc. A cluster can be built with off-the-shelf components, making it thousands of times cheaper than a single machine that is comparable in performance. The term "cluster" is a general term which encompasses several types of solutions. These solutions range from a group of heterogeneous, off-the-shelf servers running a parallel operating system to several independent computing nodes communicating via a message passing system such as MPI [45].

2.1.3 Multicore Processors

Most of today's desktop and laptop computers utilize some form of a multicore processor. At the time of this writing the prevalent desktop processors are Intel's Nehalem architecture and AMD's Phenom architecture. Both of these processors are multicore, offering 2 to 6 cores. Personal computers also contain a video device or graphics card consisting of one or more many-core processors with gigaflops of raw computing power. Also, gaming devices, such as Sony's Playstation 3 are built on top of a multiprocessor: IBM's Cell Processor [18].

PC Multicore Processors

Multicore processors increase parallelism by allowing the processor to execute multiple threads of execution at the same time. These threads of execution are managed by the operating system and can be full processes or lighter weight threads. However, it is important to note that the use of multiple cores does not come for free. Special software and operating system support must be present to allow the simultaneous use of all computing cores.

Cell Processors

Another processor architecture worth noting, created by IBM, Sony, and Toshiba, is the Cell Processor [18]. This is a hybrid processor that incorporates a moderately powered central processor and several dedicated, functional units. The central unit is called a "Power Processor Element" and is similar to 64-bit PowerPC architectures. The various functional units are called "Synergistic Processing Elements" and exhibit a SIMD architecture. Typically, the SPE units are each loaded with different code and then linked together, with the output of one SPE serving as the input to the next SPE. Because each SPE is a separate SIMD processor, more parallelism is possible. The most notable application of the Cell Processor is in Sony's gaming console, the Playstation 3. Researches have created clusters of PS3 devices to further increase the throughput of these processors, seen in [33]. The widespread adoption of the Cell Processor is still under question, due to the challenging programming environment caused by its hybrid architecture.

GPUs

GPUs, the focus of this work, are many-core processors, currently with up to 500 cores. These cores are not as full featured as a standard CPU core. For example, Nvidia's Tesla architecture does not offer layer 2 caching features. GPU cores use a Single Instruction Multiple Data (SIMD) architecture. This means that each core executes the same instruction, but operates on different data. Certain problems, such as graphics processing, require that the same calculations be performed on a large set of data. It is in such circumstances where SIMD processors excel. A GPU is not a standalone unit, however. GPUs are co-processors with their own memory space, which require a standard CPU and chip-set controller to function. GPUs do not run a

typical operating system, nor do they provide a multi-process environment. Instead, the GPU manages its administrative tasks in hardware reducing overhead costs by a significant amount. Since the GPU is an external device from the perspective of the CPU, data must be sent to the GPU via the system bus. This is a non-negligible cost which can have serious performance impacts. However, this work focuses on memory performance during execution on the GPU and not interactions between the host computer and the GPU.

2.2 Software Support for Parallelism

2.2.1 Operating System

Operating systems have long provided support for simulating parallel execution of user programs. For many years, most computers only contained one processing unit. However, even computers with only one CPU were able to service the requests of multiple users. This is done by multiplexing not only user programs onto the CPU, but also operating system execution. The multiplexing of programs is largely transparent to the user. The same principle exists in a multicore system, except the operating system now has additional resources (more CPUs) on which it can schedule jobs. In either environment, programmers can utilize logical execution abstractions known as threads to increase parallelism and reduce overhead [5]. A classical example is a producer-consumer model where a producer thread generates some set of data upon which the consumer thread operates. A benefit of this model is that both threads can be implemented in such a way that if there is no work to be done the CPU is relinquished to allow other processes to run. Threads of execution can be provided by the operating system as separate processes or system libraries such a *pthreads* [9].

Languages such as Java provide their own implementation of threads.

2.2.2 MPI - Message Passing Interface

MPI is an application programming interface that allows communication between computational elements running on separate physical machines [45]. MPI is a mechanism for creating clusters as described in Section 2.1.2. The MPI specification is language and platform independent, allowing clusters to expand across a collection of heterogeneous hardware. In addition to increasing parallelization through the use of multiple processors, MPI also provides a distributed memory environment. Because MPI is defined through the specification of an API, it must be specifically incorporated by the programmer.

2.2.3 OpenMP

OpenMP is a compiler framework that allows the programmer to easily utilize all of the processing cores in a multicore environment [7]. The central idea is to automatically spawn multiple threads for parallelizable sections of code through the use of simple compiler directives. OpenMP is implemented as a compiler extension, thus removing much of the burden of multi-threaded programming. A feature of OpenMP is portability: the ability to dynamically determine the number of available cores, removing the need for platform specific implementations. OpenMP also allows for programmer controlled granularity through the definition of work units, and handles data dependencies via standard semaphore and mutex concepts.

2.3 Cellular Automata Models

A *cellular automaton* (CA) is a discrete mathematical model used to calculate the global behavior of a complex system using (ideally) simple local rules [44]. The space of interest is tessellated into a grid of cells and the behavior of each cell is captured in state variables whose values at any instant are functions of the state of a small neighborhood around the cell. The dynamic behavior of the system is modeled by the evolution of cell states, which are computed repeatedly for all cells in discrete time steps. CA-based models are highly parallelizable as, in each time step, the new state is determined completely by the neighborhood state in the previous time step. When parallelized, these calculations are generally memory-bound since the number of arithmetic operations performed per memory access is relatively small (i.e., the *arithmetic intensity* is low). A benefit of using a CA to model a given problem is that implementing a CA is usually a straightforward task. CA models generally exhibit less complexity than other models because the next generation is based on a small neighborhood of cells instead of a complicated mathematical function or system of differential equations.

2.3.1 Conway's Game of Life

The Game of Life is a simple cellular automaton created by John Conway in 1970 [6]. It is interesting because of the large amount of different patterns that can arise, many even self replicating. In the Game of Life, cells exist in one of two states: *alive* (black), or *dead* (white). The Game of Life uses an octile neighborhood when determining the next state of a cell. There are four simple rules that determine the state of a cell in the Game of Life[1]:

[1] `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

1. Live cells with fewer than 2 live neighbors die (under-population);

2. Live cells with more than 3 live neighbors die (over-crowding);

3. Live cells with exactly 2 or 3 live neighbors live to the next generation; and

4. Dead cells with exactly 3 live neighbors become alive (reproduction).

In Figure 2.2, the cell in the center transitions from *dead* to *alive* through the application of rule #4 while the alive cells at t_i transition to *dead* at t_{i+1} due to under-population (assuming cells outside the boundary are *dead*). As with all CA, these rules are applied to each cell for every generation that is calculated – typically using two memory regions, one for the "current" state, and one for the "next" state that are swapped as the system evolves over many generations. In the simple CUDA implementation of this algorithm, each thread calculates the state of one cell in the *next* array by accessing values in the *current* array (handling boundary values appropriately).

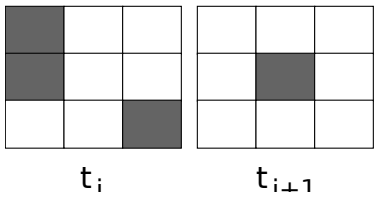


Figure 2.2: Game of Life Cellular Automaton

2.3.2 Image Processing: Scale Invariant Feature Transform (SIFT)

In the study of image processing there are certain methods that are used frequently, to solve many different problems. For example, convolution is widely used for applying image filters, while scale spaces are used in both edge and blob detection [23].

Also, both image filtering and scale spaces are used extensively in feature matching and tracking problems. The work presented here concentrates on several utility methods, such as convlolution and scale space usage, since they are so widely applicable. However, these methods are studied and presented in the context of a larger, computationally intensive image processing pipeline: Lowe's scale-invariant feature transform (SIFT) algorithm.

The SIFT algorithm has many applications, most of which involve some form of feature recognition. The detection of these features, known as keypoints, is not susceptible to changes in scale or illumination, nor is it affected by image noise. For these reasons, SIFT is a staple in computer vision applications. However, the algorithm itself is a complicated pipeline of successive steps, each of which is computationally intensive, making it problematic for use in real-time or near-real-time environments. A real-time implementation of the SIFT algorithm would have actual and immediate benefit to many applications, therefore this work focuses on several aspects of the SIFT pipeline, and how these different aspects can be viewed as cellular automata models, subject to the various performance improvements described herein. SIFT is representative of many image processing and computer vision algorithms, making its study widely applicable.

Algorithm Overview

The SIFT algorithm is a multi-step process which identifies certain pixels of an image as keypoints, and assigns those points a vector of values which describes the keypoint and its orientation. The central idea is that keypoints will occur at minima and maxima of the image; in fact finding these extrema is the first of 4 main steps of the SIFT pipeline. The second step is a refinement step where sub-pixel accuracy is used to find even stronger keypoints. After refinement, an orientation is calculated

for each keypoint, making the point robust to changes in rotation. Lastly, the vector of values comprising the keypoint is generated; this vector is used for matching one image against another. This work concentrates on step one of the SIFT pipeline; in particular the treatment of Gaussian blurring (a convolution operation), difference of Gaussians, and neighborhood extrema detection operations as cellular automata models.

Extrema Detection. The first step of the algorithm is to find these extrema as possible keypoints. A scale space is created by successively applying Gaussian filters, convolving the image. As the image is blurred it is also downscaled, creating a set of blurred images of different sizes. Each different size category of the image is called an octave; and each successively blurred image in the octave is called an interval. Once the intervals and octaves have been generated a subtraction operation is performed on adjacent intervals, producing a number of difference of Gaussians appearing at the different octaves. Minima and maxima found in the difference of Gaussians results are the initial set of keypoints; see Figure 2.3.

Keypoint Refinement. Once the initial set of keypoints has been found it must be refined by removing unstable keypoints. A simple check is to compare the contrast of each keypoint pixel to a threshold; those keypoints with a contrast below a given threshold are ignored because they are susceptible to noise. Another refinement step is to find the actual subpixel location of the keypoint. The image itself is discrete, but the actual location of the extrema may lie somewhere between pixels. Finding the actual location improves stability. The actual location is found by interpolation of the pixels surrounding the one in question; interpolation is done using the Taylor series expansion of the difference of Gaussian function applied in the previous step. The

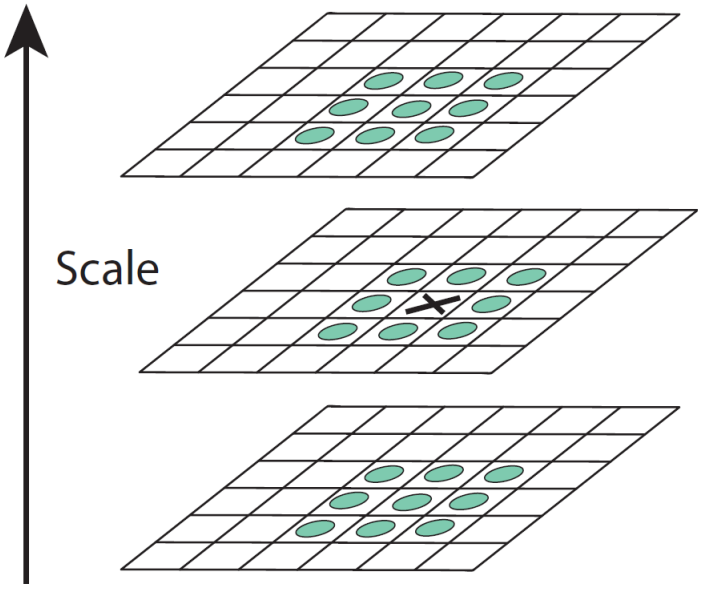


Figure 2.3: Neighborhood extrema detection in SIFT, from [23].

last refinement step is to eliminate edges which are not robust to image noise. Edges are eliminated by inspecting the principle curvature of the difference of Gaussian function, finding points whose curvature is larger than some threshold.

Orientation. Now the set of keypoints has been defined, an orientation must be calculated for each point, making the point robust to changes in rotation. This is done by examining the image gradient around each keypoint. The magnitude and direction of the image gradient around each keypoint is calculated; this is done for a neighborhood of points around the keypoint. These magnitudes and directions are stored in a histogram, allowing the dominant vectors to be easily found and assigned as orientations.

Keypoint Descriptor. The last step in the SIFT algorithm is to create a keypoint descriptor for each keypoint. A keypoint descriptor is an identifier of a pixel that is robust to changes in scale, rotation, and illumination. The descriptor is generated by

another histogram operation which calculates magnitude and orientation values in a region around the keypoint. These values are then used to create the descriptor.

2.3.3 Surface Water Flow

In his Masters thesis, Parsons proposes a cellular automaton model that uses digital elevation maps to predict surface water flow resultant from a rain event [30]. This method is further refined in [31]. Using a digital elevation map discretizes an area of land into a grid of cells, each with a known elevation above sea level. From this map, not only are relative distances easy to calculate, but also the slope of the land from one cell to another. Using this information Parsons can predict how water will flow over the map. The input to this model is a digital elevation map of the area in question and rainfall rates. The output is the amount of water remaining in each cell after the rainfall has ended. The model also takes into account geological properties of the land such as surface infiltration rates based on the content of the soil. Our work uses the CA presented by Parsons, and his Java based implementation, as a case study to apply a collection of the techniques presented here and evaluate both effectiveness and facility of each.

2.4 Nvidia GPU Architecture

At a very high level the Nvidia GPU consists of one or more streaming multiprocessors (SM), each with 8 processing cores, two special functional units, and access to a global DRAM space [21]. Each processing core contains a single 32-bit floating point unit that can also operate on integers. Each core must share the two functional units which handle operations such as transcendentals and double precision operations. The architecture of Nvidia's 8800 GPU can be seen in Figure 2.4; the Nvidia 8000 is

a GPU with 128 distinct processing cores cores and also one of more powerful CUDA enabled processors when it was first released.

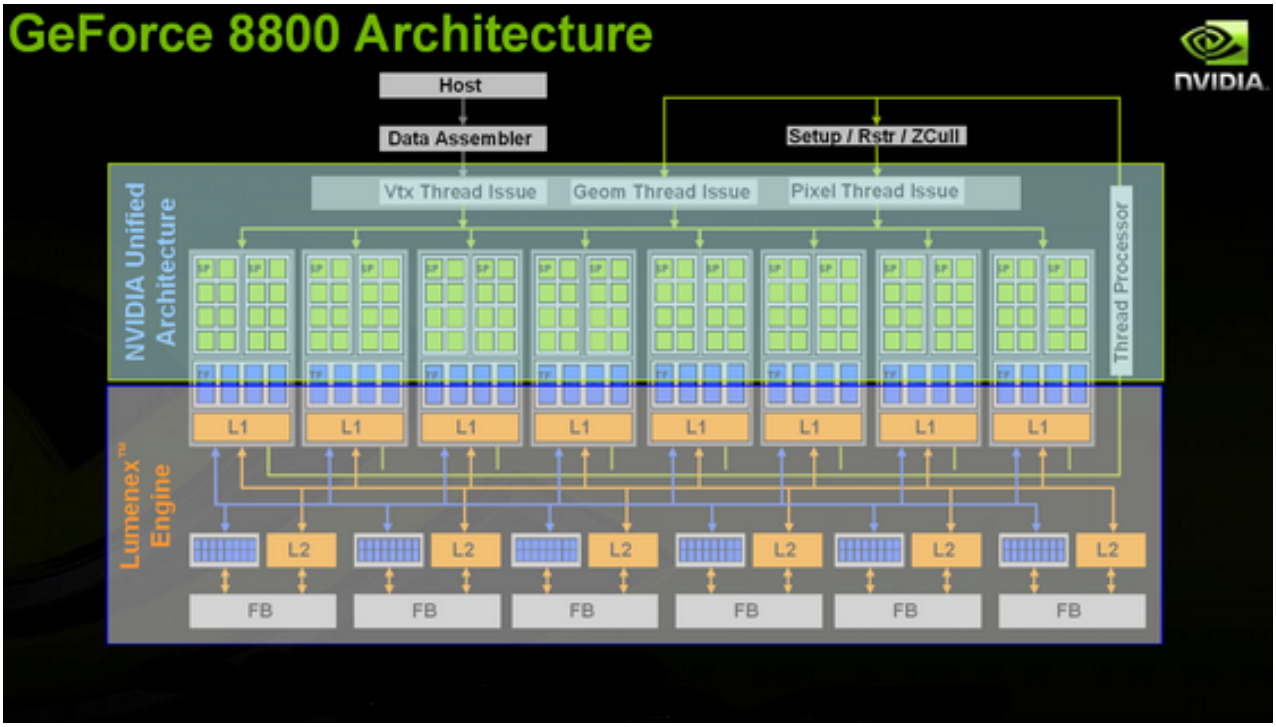


Figure 2.4: High-level architecture of an Nvidia GPU

2.4.1 Single Instruction Multiple Data (SIMD)

Nvidia GPUs utilize a *single instruction multiple data*, or SIMD, architecture for executing instructions across their multiple cores. SIMD means that, on a given SM, every processing core executes the same instruction at the same time. Forcing every core to execute the same instruction may seem highly restrictive, but it enables the hardware to maintain only one set of instruction registers, thereby improving context switching performance and lowering the amount of thread management data required. For problems with a high degree of data parallelism, SIMD architectures perform very well. The SIMD architecture provides a basis upon which the CUDA threading model

is built. This architecture also influences implementation details such as the use of conditionals and other branching constructs.

To understand the power of parallelism that the Nvidia SIMD architecture affords, consider a CA that computes the next generation for a single cell in 10ms, on average for a CPU. When this CA runs on a data-set of 10,000,000 items, it takes approximately 100,000,000ms or about 27.78 days to execute. If we implemented this CA in a GPU environment with 500 cores where it took 100ms to compute the next generation of a single cell, execution would take approximately 33.33 minutes. This is simply due to the fact that the GPU can calculate the next generation of 500 cells at one time. This is a theoretical example meant to illustrate the power of a SIMD architecture in the context of a CA model.

2.4.2 Memory Regions

There are several different memory regions available for use on the GPU. Each of these regions has special characteristics and is used for different purposes. The available memory regions are, a generic global memory, a small, on-chip, user-managed cache memory called shared memory, texture memory, and constant memory. This work primarily deals with global memory and shared memory.

Global Memory

Global memory for the GPU is analogous to main memory for a CPU: it is an off-chip piece of hardware that provides storage for data and must be accessed via a system bus. Global memory access is expensive and must be carefully managed. However, it is the only large memory area available to application developers. Typically, data is copied from the host, to the graphics device and stored in global memory; ker-

nels executing on the GPU then access that memory over the dedicated bus on the graphics device, perform some calculations, and finally write results back to this same global memory region. The results are then copied back from the graphics device and inspected or further processed by applications running on the CPU. A single memory load request on the GPU is issued in 4 clock cycles, but there is also an additional 400 - 600 cycles of latency to access global memory [27]. Global memory is not automatically cached in the Tesla architecture, and each access request pays this latency penalty; the Fermi architecture does offer an L2 cache of 768KB intended to add better memory performance for applications that do not have a high degree of spatial locality in their memory access patterns, and as such, is beyond the scope of this work.

Shared Memory

In the Tesla architecture, shared memory is a 16KB, on-chip, user managed cache. This has been increased to 64KB in the Fermi architecture. Shared memory is used when a set of threads need to access the same piece of data. The data element is read from global memory and copied into the shared memory region where all threads can then access that element in 4 clock cycles (subject to read-after-write delays). Using shared memory eliminates the need to repeatedly request data from global memory, thus improving performance. In a typical kernel, shared memory is first *staged*. This means that each thread loads one or more values from global memory, copying them to the shared memory space. Once all the required values have been copied the computation commences, reading data from shared instead of global memory.

Texture Memory

Texture memory is an area frequently used by the GPU when doing graphics operations. This memory region provides a two dimensional locality caching mechanism: when a processing core requests an element from texture memory the surrounding elements are automatically read and loaded into the cache, improving access times for subsequent requests from different threads. Texture memory also has attached hardware called texture units which are capable of doing certain floating point operations, such as linear interpolation, in specialized hardware. Surprisingly, a major drawback of texture memory is the cache. When writing to texture memory there is no guarantee of cache coherence, so subsequent reads from the same kernel will not reflect the latest modifications. Texture memory does not exist in different physical hardware than global memory; in fact, texture memory is mapped onto global memory, thus changing the access mechanism. This work does not employ texture memory due to its specialized nature.

Constant Memory

Constant memory is a cached region for unmodifiable data. Since the data cannot be modified, maintaining cache coherence is simpler and there is no read-after-write problem as with texture memory. For constant memory to be employed effectively however, each thread must request the same location of constant memory. The cost scales linearly with the number of different addresses requested [27]. Due to this limitation, the work present here does not use constant memory.

Kernels

A kernel is a programmer defined unit of work that is executed on each of the processing cores of a streaming multiprocessor. The same instruction runs on each core, only the data upon which it operates is different. The kernel is the portion of the algorithm that is actually run on the GPU. Different kernel implementations are the main contribution of this work.

Blocks and Threads

CUDA accomplishes a high degree of parallelism by supporting thousands of threads in hardware. The organization of these threads is an important task. Even though each thread is going to execute the same instructions, there must be a mechanism that instructs each thread which element to load, increment, or modify. In CUDA, threads are grouped into one, two, or three dimensional units called thread blocks. These dimensions facilitate mapping each thread to a particular data layout. For example, data that are naturally oriented in a grid will usually require a two dimensional threadblock. Threadblocks are organized into a one or two dimensional grid. When a streaming multi-processor is ready to begin execution, a block is scheduled on that SM, and execution commences. The threads in the threadblock run, each deriving a unique identifier from the location within the threadblock and the location of the threadblock within the grid. A graphical representation of threadblocks can be seen in Figure 2.5; note, this example shows a two dimensional grid of blocks where each block contains a two dimensional organization of threads. It is also possible to arrange the threads in three dimensions, thus facilitating mapping to three dimensional data.

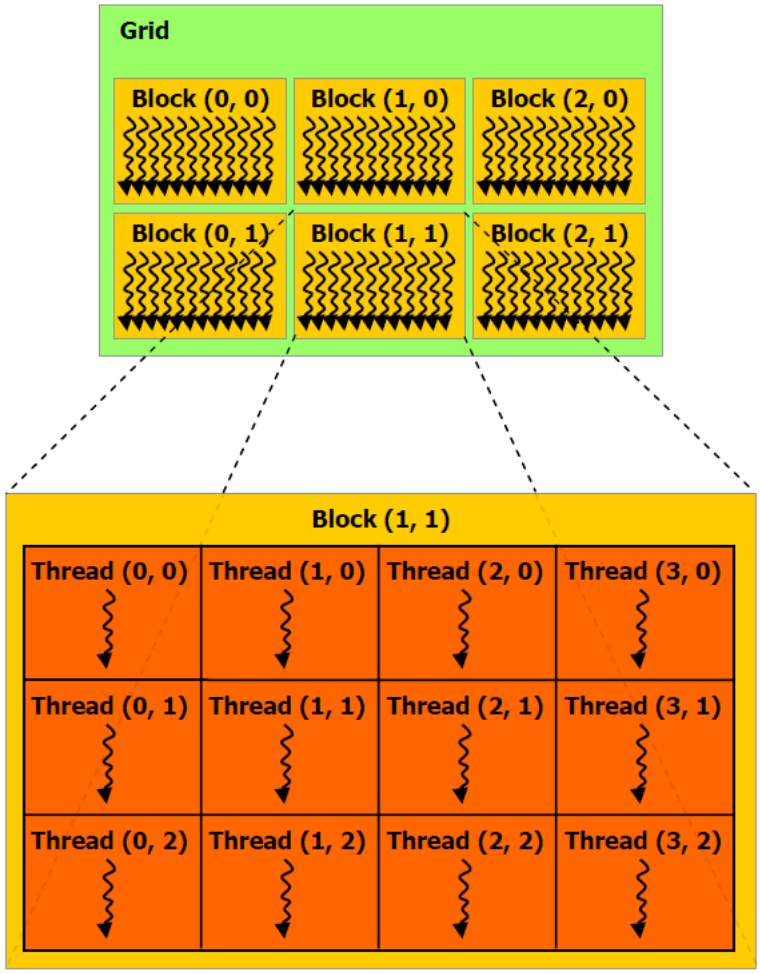


Figure 2.5: CUDA threadblocks and the grid into which they are organized; from [27].

Warps and Half Warps

Threads are further broken up into warps and half-warps, which are simply groups of 32 threads or 16 threads, respectively. Warps are artifacts of the SIMD architecture: the smallest unit of threads that the hardware is able to execute. Warps are not programmer controlled or accessible, but they are important to understand because many instructions are implemented in the hardware on a half-warp boundary, especially memory instructions.

Coalescence

Memory coalescence is an important topic in the Tesla and Fermi architectures. A natural result of the single-instruction-multiple-data paradigm is that when a memory load instruction is executed, all 8 processing cores on a streaming multiprocessor issue a memory load request. Waiting on 8 memory requests would waste a considerable amount of time. To overcome this problem, the memory subsystem offers a significant performance improvement known as coalescence. When specific criteria are met, 8 separate loads are coalesced into a single request, thus reducing the total memory latency and improving memory throughput. To fully understand when coalescence is achieved it is important to know something of the underlying memory hardware. Memory is divided into multiple, equally sized modules called *banks*. Each bank can be accessed simultaneously [27]. Therefore, if each thread accesses a different bank, coalescence is possibile. However, if two threads access the same bank, there is a *bank-conflict*, and the requests must then be serialized. The details of coalescence are described in Section 4.2.2.

Branching and Divergence

Another result of using a SIMD architecture is that conditional logic becomes problematic. When a conditional instruction that is dependent on data is executed, each of the eight processing cores could return a different result forcing the processor to jump to different places. This is called branch divergence and impacts performance because all eight processors are executing a single instruction. For example, an if-statement may cause 31 threads of warp to branch one way, and a single thread to branch to a different execution path. When the half-warp that contains the solitary thread executes, only 1 processing core executes a meaningful instruction, thereby wasting 15

cores over two clock cycles. This work does not focus on how the hardware deals with this problem, but only tries to minimize the number of branch divergent situations that could possibly arise.

Chapter 3

Related Work

In general, the work presented here can be categorized into two large areas: (1) cellular automata and (2) GPGPU computing; the focus being the intersection of these two areas. Much research has been done on CA theory, as well GPGPU computing; however, the current body of work involving CA models using GPGPU computing is rather small. In this section, existing CA theory is discussed, in particular, the optimization thereof, followed by a selection of current GPGPU research. The small collection of work comprising the intersection of these two areas is also described, thereby putting the work presented here into an appropriate context.

3.1 Cellular Automata Theory

The study of cellular automata theory hearkens back to the initial days of computer science, when Alan Turing and John Von Neumann were busy discovering the fundamental principles of computation. Cellular automata grew out of Von Neumann's fascination with Turing's computing machine, while he was studying self-replicating entities. Von Neumann's seminal paper "The General and Logical Theory of Au-

tomata" [44] gave rise to a whole new class of modeling tools and enabled scientists from many domains (chiefly physics, at the time of Von Neumann's initial article) to study their areas from a different perspective.

Because of their simplicity, CA models are used in fields such as economics, physics, chemistry, biology, computer science, and mathematics. The intent of this work is not to survey example CA models from each of these concentrations, but to study the basic definition of a CA so that improvements proposed herein can be applied to CA in many different fields. As such, classical examples, which exemplify CA and their rules, are given from several fields of study.

In 2002 Stephen Wolfram released a book entitled "A New Kind of Science," [48] in which he discusses the importance of mathematically modeling the complexity of various systems and argued that traditional methods are not sufficient to describe the complexity found in many realistic systems. He uses different cellular automata as examples of these systems. "A New Kind of Science" exposed cellular automata to a wider audience, and put CA into the context of computation, making CA more attractive to researchers and thus more widely used.

3.1.1 CA Models for Physical Science

In [43], Vichniac describes several topics in physics which can be modeled by a cellular automaton. He discusses the topic of nucleation, where physical particulates cause the growth or attraction of other physical particulates and form a nucleus or centralized collection of that material. An example of nucleation is the seemingly spontaneous generation of carbon dioxide bubbles in a carbonated beverage: these bubbles form around minute particles suspended in the solution. Vichniac postulates that this can be modeled by a simple neighborhood voting mechanism: the function which

determines the value of a region does so by analyzing the "popularity" of a given state in the surrounding neighborhood. An example of a voting rule is: when the number of neighbors is above a given threshold, the rule always returns a one or true value. Vichniac goes on to say that this type of rule causes the growth of clusters, until a stable state is achieved.

Toffoli, a contemporary of Vichniac, makes an even stronger claim in [42]: in physics models, the mathematical tools typically used contain too much formality, and that because of this formality, physicists spend much of their time working around the nuances of the mathematical base and not enough time working on the physics of the problem. Toffoli's main argument is for the replacement of differential equations by cellular automata models. He argues that because many differential equations do not have a closed form solution and the only means of solving them rely on numerical computation using a computer, why then should the physics models themselves be based on a mathematical system that may or may not be related to the physics at hand? Toffoli gives the example of the heat equation: $c\frac{\partial T}{\partial t} = k\nabla^2 T$ which does not perform well for small volumes or time steps. He proposes that partial differential equations whose independent variables are space and time can be viewed as CA models. The continuous elements of the equation, space and time, are replaced by a discrete grid, but the state at each point (cell) remains a continuous entity (i.e. value). The derivatives are then replaced by the differences of the values of each cell.

Ermentrout and Edelstein-Keshet in [10] define three classes of CA models, deterministic, lattice gas, and solidification models that represent different examples of certain natural biological phenomena. The deterministic model is one that most relevant to the patterns and techniques discussed herein; predator-prey interactions and excitable media are modeled using a deterministic model. A deterministic model is also readily described by evolution equations, which are frequently expressed as par-

tial differential equations. A lattice gas model is similar, but includes the introduction of random events which further modify the state of each cell. Lastly, the solidification model is one where a cell cannot change once it has reached a certain, predetermined state; fungus growth can be modeled with this type of CA. Ermentrout and Edelstein-Keshet's work defines the states space and rules for several different domains such as the predator-prey problem, bacteria growth, self organizing ant trails, and occular dominance columns in the visual cortex of the brain. In each of these example, they use the CA to generate grid-based visual data and compare that with data gained by experimental means. The visual data generated by each of the CA models closely match that which is collected empirically. Their conclusions maintain the need for strict mathematical models, but suggest that CA models may, and should, be used at first to quickly ascertain a general understanding of the processes involved.

3.2 GPGPU Computing

General Purpose GPU computing is a relatively new field, yet many researchers are utilizing these many-core processors to investigate new methods of achieving previously unattainable goals. For example, Szerwinski and Güneysu have investigated the feasibility of using a GPU to offload cryptographic operations to reduce CPU workload and increase throughput for client-server applications [40]. Their goal focuses on using the GPU to replace cryptographic accelerator cards, which are significantly more expensive than GPUs. To that end, they present multiple algorithms for implementing modular arithmetic in a SIMD environment. An important element of their work is manipulating very large numbers and doing so efficiently in an environment that does not natively support numbers of the required size. For example, on the Nvidia platform the division operation is a computationally expensive operation and

it is suggested that it should be avoided [27]. Therefore Szerwinski and Güneysu have used techniques which do not require division, such as Montgomery's Technique [16] and using a Residue Number System [14]. Implementation details are also discussed, such as the decision to use shared memory, the number of registers used by each thread, and thread synchronization. Of particular relevance is their decision to use a more computationally expensive modulus in order to reduce the number of memory accesses. The results are presented in terms of throughput: modulo operations per second. The throughput performance achieved by their methods is better than many other methods described in the literature; however, the GPU implementations suffer from a higher latency.

Graph theory is used in a host of different applications, from network flows and connectivity to, more recently, social engineering. However, pervasive as graph theory problems are, they are traditionally difficult to parallelize [40]. Recently, GPUs have been used to narrow this gap: in [17], Harish and Narayanan have devised multiple graph search algorithms that run on Nvidia GPUs. The authors introduce several searching algorithms: breadth first search, single source shortest path, and all pairs shortest path. The authors state that spacial locality information is difficult to derive from the graph itself, therefore the use of shared memory is not feasible. The BFS version presented is a two part method which uses a GPU kernel that is repeatedly called by CPU code which determines when the graph has been completely searched. The algorithm uses a "frontier" to determine which vertices of the graph it should consider next. The single source shortest path implementation is actually two separate kernels; these are called successively to deal with synchronization problems. These synchronization problems are artifacts of the hardware platform and CUDA driver version used in the publication; newer versions of the drivers offer synchronization features which obviate the need for a two step kernel. The all pairs shortest path

implementation presented is a GPU version of the Floyd-Warshall [11] [46] method algorithm. However, this method is $O(V^3)$ in memory, so it is not useful for large graphs due to the memory constraints of today's graphics cards. The results gained by the above methods are compared to serial implementations run on an AMD processor platform and the GPU versions achieve up to a 70x speed up. These results clearly show promise for the continued research into parallel graph algorithms using GPUs.

In Section 2.3.2 the SIFT algorithm was introduced as being computationally expensive, thereby making it a good candidate for performance improvements using GPUs. This fact has not escaped the research community. In [39] Sinha et al. present a GPU-based SIFT implementation that runs at a real time frequency of 10Hz, approximately 10 times faster than known CPU implementations. They present this work in the context of object tracking through successive image frames supplied by a video feed. The focus of [39] is an implementation of the SIFT algorithm that is as fast as possible; whereas our approach is to apply cellular automata theory to specific steps of the SIFT algorithm thus highlighting the benefits and features of CA theory in the context of a real world problem. Also, the implementation presented by Sinha et al. is a full implementation, whereas our work concerns only portions of the SIFT pipeline. Specifically, our work does not complete any of the orientation calculations and is therefore not used for any keypoint matching. Sinha et al. use techniques and methods of the Nvidia platform that we intentionally avoid: texture memory is heavily used as is a separable convolution kernel. Also, Sinha et al. implement portions of the pipeline on the CPU due to the serial nature of the latter steps of the SIFT algorithm. Lastly, their work does not use the CUDA environment, but instead they utilize OpenGL and various shading languages for their implementation. Another example of SIFT is the work of [20] wherein the authors use SIFT as a mechanism for eye blink detection. Their work uses the OpenVIDIA [13] GPU implementation

of SIFT which is readily available and they report good results.

3.3 Cellular Automata Using GPUs

The above literature discusses cellular automata theory and GPGPU programming in mutually exclusive contexts as the body of research in the union of these two areas is currently small. There has been, however, some notable work that incorporates topics from both CA theory and GPUs. In [35], the authors compare different CA models implemented using various techniques on both single core, multi-core, and GPU processors. Seven different algorithms are presented which implement a generic CA model in different ways. For example, the authors differentiate a brute force algorithm that computes the next state for every cell from the case where computation is only done for those cells whose neighbors changed. Multi-threaded versions of these two cases are also presented and tested, while two GPU algorithms are given. The only difference in the two GPU algorithms is that intermediate generations are read back from the device. In their results, the GPU implementations are clearly superior in several cases. However, the authors have found certain situations where the CPU versions can outperform GPU ones: namely, when the number of candidate cells to be processed is small. For example, in their Game of Life results, the CPU version has a much higher throughput for small game boards because a relatively stable state is reached after only a few generations and therefore the number of possible changes from one generation to the next is small. However, for large boards, the GPU implementations are superior.

The research most related to ours is that of Micikevicius [24] in which he presents a method to compute three dimensional finite differences. This method uses successive 2D slices of 3D data to compute the difference over a whole volume. A large

portion of this work is dedicated to memory organization in order to maximize data reuse and increase throughput. Micikevicius presents a novel approach to utilizing shared memory to store required data, including data halos. A redundancy metric is introduced that relates the number of elements read from (or written to) memory and the number of elements that are actually processed. We use this metric in several cases, and it is especially useful while analyzing the multi-generation technique described in Section 4.3. Also, we use the non-square threadblock technique he presents in order to reduce the required number of halo elements and increase bus utilization. The nature of the 3D finite difference computations is similar to CA models in terms of the memory organization and access patterns, hence our interest.

Chapter 4

Memory Access Patterns

For cellular automata, the detailed memory access patterns proposed here fall into two broad categories. First discussed are those that are related to memory layout and organization. Due to the differences in memory architecture discussed above, aligning the data in memory on a 16-byte boundary is critical. In addition, because most cellular automata models operate on a grid of data, special care must be taken when computing values at or near the edges because a certain cell's neighbors might not exist. For example, it is completely reasonable for a CA implementation to be implemented in the following way:

```
if (row == 0 || row == width - 1) {
  // some special case handling here
} else {
  // load the data
}
```

However the *if-statement* in the above code contains on the order of 10 processor instructions. If the application ultimately launches 100,000 threads then around 1,000,000 instructions are executed just to handle this edge case. Threads that are

not executing on the boundary still execute this code. One way to remove this logic is to pad the data in memory and then fill these padded neighbors with a reasonable sentinel or default value so that the cells on the edges do, in fact, have neighbors. This added padding is called a *halo* and we shall see that adding halos can significantly reduce instruction count but also decrease memory throughput, so care must be taken in how we access them.

The second broad category of optimizations deals with reducing the memory-access-to-instruction ratio by doing more computation per memory load, or conversely, reusing previously computed values to reduce instruction count. These techniques require each thread to compute the values for more than one cell.

4.1 Naïve Implementations

Due to the parallel nature of GPUs, an initial implementation of a CA model may yield a significant performance improvement over a serial implementation of the same model. The performance gain is a direct result of the increase in processing cores. Where a CPU can calculate the result of only one cell at a time, the GPU can calculate many. Because the GPU is operating on many elements at the same time, more simultaneous data access is required leading to a situation where performance becomes memory bound. These memory bound situations can be further improved by the techniques described in the following sections.

4.2 Memory Organization

This section presents four different methods of organizing memory to maximize bandwidth and increase arithmetic intensity. Discussed first is the use of shared memory.

While, relatively obvious, the use of shared memory can make a dramatic difference in performance. Second, padding data with halos to reduce edge case logic is presented. Next, the effects of aligned memory access are analyzed in detail. Lastly, the shape of an effective memory region and its impact on performance is disserted.

4.2.1 Shared Memory

As mentioned above, the memory system of the Nvidia Tesla architecture has no caching for general global memory accesses. The Fermi architecture contains two levels of caching but these are relatively small caches which do not scale with the number of possible threads. Therefore, if the same memory location is accessed multiple times, each request results in a bus transaction to retrieve the value from main memory and bring it into a register on the processor. Thinking about a CA model that is based on neighboring cells, it is easy to see that this situation would arise immediately. Many cellular automata consist of rules which operate on an *octile neighborhood* of each cell. An octile neighborhood is defined as the 8 cells which immediately border a central cell. Another way to visualize this is as a 3x3 grid of cells with a singular central cell and its surrounding 8 neighbors. In an octile neighbor environment, two adjacent cells share six neighbors, a fraction of 2/3 when the cells in question are included. The neighborhood of two cells and their shared neighbors can be viewed in Figure 4.1.

The Nvidia memory architecture addresses this by supplying something called *shared-memory*. This is an on-processor memory area to which all threads in single threadblock have access. Once a value is written to this area, it can be retrieved in about 4 cycles; a new bus transaction to global memory, by contrast, can take

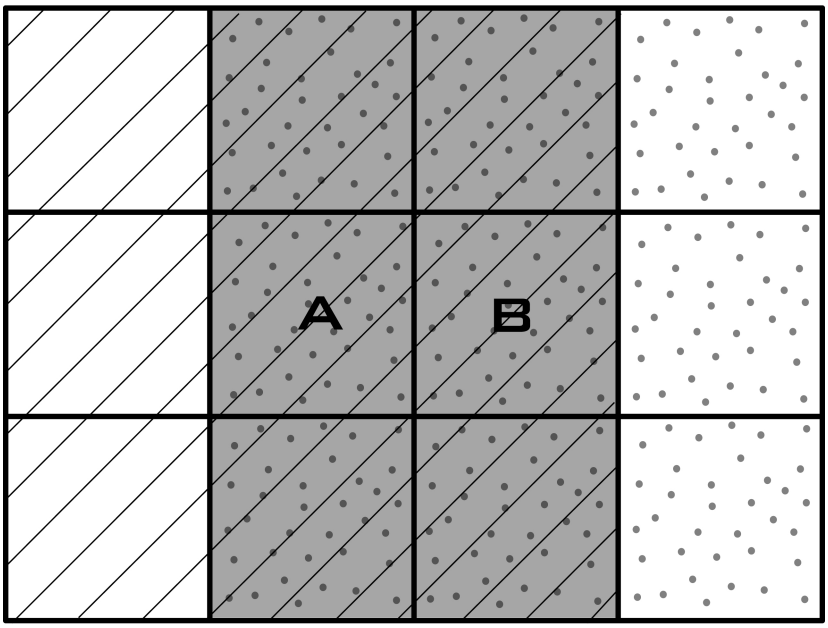


Figure 4.1: A depiction of how cells share neighbors. The shaded cells are neighbors of BOTH A and B. The diagonally lined cells are neighbors of A, and the dotted cells are the neighbors of B.

hundreds of cycles. Typically, shared memory is explicitly staged before computation takes place; this is a manual step whereby the programmer instructs each thread to load a value from global memory into shared memory. Shared memory amounts to a user-managed cache. Once the required values have been loaded and all threads synchronized, computation can commence, requesting those values from the shared memory region local to the processor. Following is an example of staging shared memory:

```
int globalIdx = r * dataStride + c;                                    1
int sharedRow = threadIdx.y;                                           2
int sharedCol = sharedIdx.x;                                           3
int sharedIdx = sharedRow * blockDim.x + sharedCol;                    4
__shared__ float shared[blockDim.x * blockDim.y];                      5
                                                                       6
shared[sharedIdx] = data[globalIdx];                                   7
```

```
                                                                    8
__syncthreads();                                                    9
                                                                    10
// start computation                                                11
```

Clearly, maximizing the use of shared memory is a good practice. However, there is one major downside to using shared memory: there is a limited amount of space. Not only is it limited, but it also draws from the same pool of resources as the register file. Therefore, complicated kernels which require many registers have reduced shared memory capacity. At the time of this writing Nvidia has two different architectures in the field: Tesla and Fermi. Tesla has 16KB of shared memory while Fermi supplies 64KB. This memory fills up quickly, especially when dealing with CA models that have several pieces of data associated with each cell.

4.2.2 Memory Alignment

One of the drawbacks of the SIMD architecture is that memory load instructions are issued at the same time, putting a significant strain on the memory bus. As discussed in Section 2.4.2, a memory load request can take up to several hundred cycles. If all 16 memory load requests from a half-warp were serialized, performance would suffer. To get around this problem, the Nvidia memory architecture provides a concept known as memory coalescence. When certain criteria are met, the memory bus can coalesce 16 separate memory load instructions into one bus transaction, thereby considerably reducing the time required to service all the memory requests of a half-warp. The criteria for memory coalescence are as follows [27]:

- Threads must access
  - Either 32-bit words, resulting in one 64-byte memory transaction,

  - Or 64-bit words, resulting in one 128-byte memory transaction,
  - Or 128-bit words, resulting in two 128-byte memory transactions

- All 16 words must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words). This means that for a 64-byte memory transaction, for example, all word must lie in a contiguous 64-byte area of memory.

- Threads must access the words in sequence: The *kth* thread in the half-warp must access the *kth* word.

Memory coalescence is critical for memory bound applications like cellular automata. Incorrect alignment of data can cause performance degradation and render most other techniques discussed in this work ineffective.

4.2.3 Halos

Memory halos are used to minimize instruction count caused by logic dealing with edge or boundary cells. For example, the cells in the top row of a grid of data do not have a neighbor above them. Instead of checking to see if each cell is on the top row, an extra row of data is inserted above the top row, so that now each cell in the top row does contain a top neighbor. The padding can also be added on each side of the memory region, removing the need for almost all edge case code for loading memory and neighbors. Not only are these instructions removed, but removing the conditionals that encompass these instructions prevents divergent branches which further reduce efficiency.

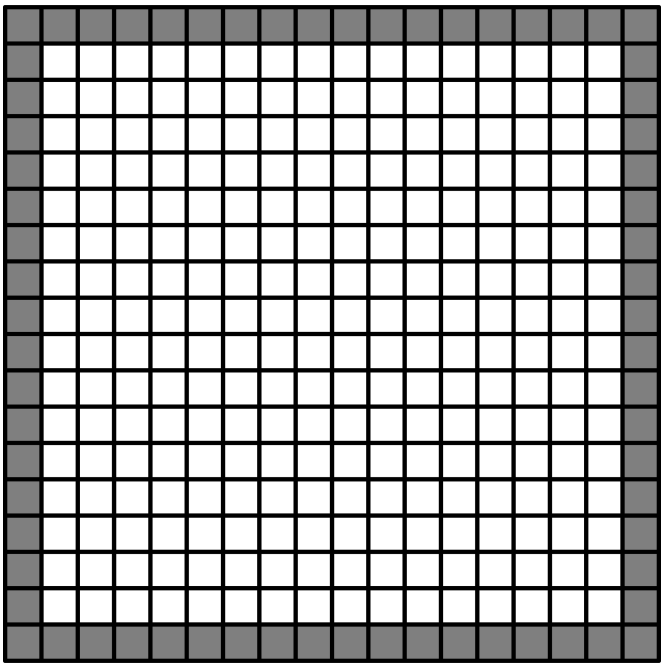


Figure 4.2: 16x16 Effective Region with a one cell halo

Halos and Alignment

The importance of memory alignment is discussed above, however, when halos are added to data, does this affect the alignment? The answer is yes, and care must be taken to maintain alignment. The trouble lies not with the top and bottom rows of the halo, but the new left and right columns that are added. A whole row of data can be easily aligned, but when one cell is added, maintaining alignment is more difficult.

The central idea is that the actual data is aligned along the correct memory boundaries, while the halo cells are in unaligned positions. Since the bulk of memory accesses are for actual data and not halo cells, the ratio of aligned to unaligned accesses will be high. It is only when a halo cell is loaded that the cost of an unaligned access

is paid. Unfortunately, it is impossible to avoid all unaligned accesses when using memory halos. However, when dealing with a large enough data set the cost of the unaligned accesses is generally hidden by the latency of the memory system: even aligned memory requests can take up to two hundred cycles. The central hypothesis of this work is that this is a cheaper price to pay than increasing the instruction count by adding special logic to handle the edge cases.

Halos and Bus Usage

As stated above, adding a halo changes the memory alignment characteristics of a data set. From Figure 4.2, it is clear that actual data is completely aligned. But the halo cell has, in effect, become part of each row's padding. This means that whenever an edge cell thread requests one of the halo cells a complete memory transaction occurs, receiving a full 32 bytes from the system bus. If each halo cell is only 4 bytes then 7/8 of this bus transaction is wasted for every halo cell request. The wasted bytes are, unfortunately, unavoidable. However, the number of halo bytes which causes this waste can be minimized, as we shall see below.

4.2.4 Effective Memory Region Shape

When using a memory halo, loading the right-most halos from memory incurs a significant throughput penalty because though only one data element has been requested the memory subsystem returns a full 32 bytes across the system bus. However, it is possible to fetch the same number of effective data elements while reducing the number of right-edge bytes requested. This is done by changing the shape of the effective memory region upon which the warp is operating. Up to this point, square threadblocks have been discussed and demonstrated. A 16x16 threadblock operates

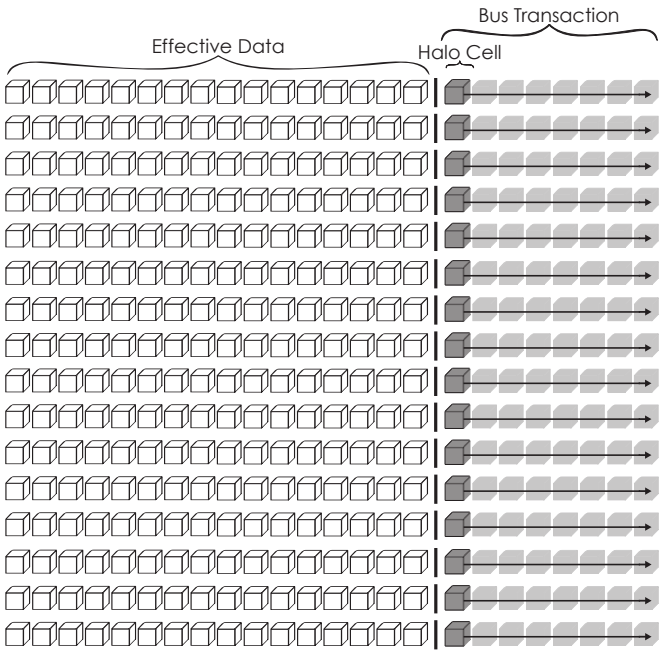


Figure 4.3: The cost of loading a halo element: when loading the right most halo element, possibly only 2 or 4 bytes, the memory system returns a full bus transaction worth of data which is 32 bytes.

on 256 elements. A 64x4 threadblock also operates on 256 elements. The main difference is that the 64x4 thread block is much wider than it is tall. Also, only the right edge halo bytes incur the bus throughput penalty. In a 16x16 threadblock there are 16 such halo cells, but in a 64x4 threadblock there are only 4 such cells. By simply "reshaping" the effective memory region, the number of bus transactions with wasted data has been significantly reduced. Figure 4.4 illustrates this.

4.3 Multiple Data Per Thread

All of the techniques described thus far have either reduced the number of memory requests required, or ensured that each memory request returns a minimum of superfluous data. The following methods attempt to increase the arithmetic intensity of

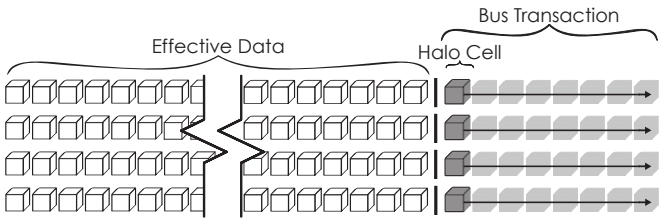


Figure 4.4: Using a rectangular kernel reduces the number of right-side halo elements, thus reducing wasted bytes. The width of the region in this image is abbreviated: it is actually 64 elements wide, thereby equaling a 16x16 region in total elements while reducing halo traffic.

a kernel. As discussed above, the arithmetic intensity measure indicates what ratio of a kernel's instructions are devoted to actual computation as opposed to memory operations.

4.3.1 Two Elements Per Thread

One method of increasing the arithmetic intensity of a kernel is to reduce the number of overhead instructions that kernel executes. For example, almost every kernel that operates on a large set of data has an index calculation to determine which element a thread should process. Typically, that calculation is implemented in the following manner:

```
unsigned int r = blockIdx.y * blockDim.y + threadIdx.y;                1
unsigned int c = blockIdx.x * blockDim.x + threadIdx.x;                2
const unsigned int idx = r * stride + c;                               3
```

When this code is compiled it produces on the order of 10 to 12 machine instructions. This overhead must be paid by every thread executing in every block. Once this calculation is completed, however, it is comparatively inexpensive to compute relative indices. A relative index can be generated by adding a constant value to the already computed index in one machine ADD instruction. By reusing index computations the

number of overall instructions is reduced, resulting in noticeably better performance. It is important to note that elements found at relative indices must still be requested from global memory and the memory patterns in Section 4.2 still apply.

4.3.2 Data Packing and Interleaving

As discussed in Section 4.2.2, memory coalescence merges several memory load requests into one or two bus transactions with a maximum of 128 bytes per bus transaction. It is common for each thread to load an element from global memory and then perform some operations on that element. Many kernels, such as the blur kernel discussed in Section 5.3.1, operate on elements of type `float`. Assuming memory is correctly aligned, when all the threads in a half-warp request an element from global memory, one 64-byte request is sent to the memory subsystem. However, the maximum number of bytes in a single transaction is 128, and the 64-byte request causes 64 bytes to go unused even though it is coalesced. The Nvidia architecture provides some built in vector data types which allow the full utilization of the bus. For example, the `float2` data type consists of two 4-byte floats in the form of a struct. However, since the `float2` type is native to the GPU the memory subsystem accesses the `float2` type as one 8-byte word instead of two 4-byte words. The `float2` type can be easily used to increase bus utilization and improve memory throughput. If the blur kernel requests elements of `float2` instead of simply float, the 16 memory requests of the half-warp are coalesced into one 128 byte request, thus not only fully realizing the data throughput of the bus but also reducing the number of total memory request by a factor of 2. While this is an appreciable improvement, it does not come without cost. Storing the 8-byte `float2` type in local registers or shared memory requires twice as much space as storing a 4-byte float. For kernels that require a large amount

of data to process a single element, using the `float2` type may not be practical due to register pressure.

4.3.3 Multiple Generations Per Kernel

A GPU kernel that implements the logic of CA model normally computes a single generation, outputting the results in to an area of memory separate from the input data. The output of a single generation can be used as the input to the same kernel for the next successive generation. When a large number of generations is to be computed, it is possible to reduce the number of kernel launches if the kernel computes more than one generation. Computing multiple generations in a single kernel launch has two main benefits:

- reduce the number of instructions, reusing already computed indices, similar to the idea presented in Section 4.3.1
- reduce the number of global memory load instructions

To illustrate this point, Conway's Game of Life is used for a small example. Consider a 16x16 game board (for the more conventional single generation approach), where each thread loads a cell into shared memory. Once shared memory has been populated, the computation takes place, writing the results to an output area of memory. To compute the next generation, that output area is then staged into shared memory by the next launch of the same kernel, and the computation takes place again. This processes is repeated until the desired number of generations has been reached. In order to compute two generations, the kernel makes 768 total global memory loads, since each 16x16 region requires an 18x18 area ($18 * 18 * 2 = 768$). Now consider the case where the kernel computes two generations: shared memory

is similarly staged and the computation is done, this time writing the results to a temporary area of shared memory. The computations are then repeated using the temporary shared memory as input, and the results are written back to global memory, thus completing the second generation with only 400 global memory loads. To see why a two generation approach for a 16x16 area requires 400 unique memory loads see Section 4.3.3.

Instruction Reduction

In Section 4.3.1 the number of instructions were reduced by leveraging the spatial locality of data as it resides in memory. That is, once the index of a single element has been computed, indices of other elements can be computed as an offset from this element in a single add or subtract instruction. A multi-generational approach is appealing for a slightly different reason: the number of kernel launches is reduced by a factor of the number of generations the kernel computes. If a kernel computes 4 generations in a single launch, then number of overall launches is reduced by a factor of 4. Since the kernel is executing fewer times, the number of index calculations that kernel computes over all the generations is reduced. However, the kernel still computes the specific CA logic four times, thus increasing the arithmetic intensity. For the Game of Life the arithmetic intensity is increased by executing the following line 4 times in the same kernel:

```
next[idx] = (liveCount == 3) || (liveCount == 2 && tile[sharedIdx]);
```

1

Memory Load Reduction

Similar to the instruction reduction effects of the previous section, multi-generational kernels also realize a reduction in the number of memory loads required to compute

the desired number of generations. Initially, the kernel loads each element from global memory into shared memory; the subsequent generational computations operate on values existing in shared memory. Because the kernel uses shared memory as temporary data store, the need to read and write to global memory is removed. As with instruction reduction, the number of memory loads is also reduced by a factor of the number of generations computing by the kernel.

Multi-Generational Pitfalls

There are several aspects of the multi-generational method which require more discussion. First, the next generation of a cell is dependent on that cell's current state and that of its neighbors. Therefore, to compute the next generation of a 16x16 block of cells, an 18x18 input area is needed. We refer to the resultant generation as the effective region of computation. That is, 324 elements are required to be input in order to compute an effective region of 256 elements. The ratio of effective computed region size to the number of required memory loads is simple to calculate: 256/324. However, if the desired effective region is 64x4 the ratio becomes 256/396, indicating that the ratio is dependent upon the effective region shape. Since this ratio is calculated for the computation of one generation it can be used as an upper bound when comparing the performance of kernels that compute multiple generations.

In the process of computing multiple generations, the number of cells used during computation of the same 16x16 region is even larger. It is easier to analyze this problem by working backwards. As stated above, an 18x18 region is required to compute the next generation for a 16x16 region. However this 18x18 region must be the result of computing the first generation; meaning that in order to compute the 18x18 region, a 20x20 region is required. Unfortunately, as the number of generations a kernel computes rises, so too does the amount of input data required. The ratio

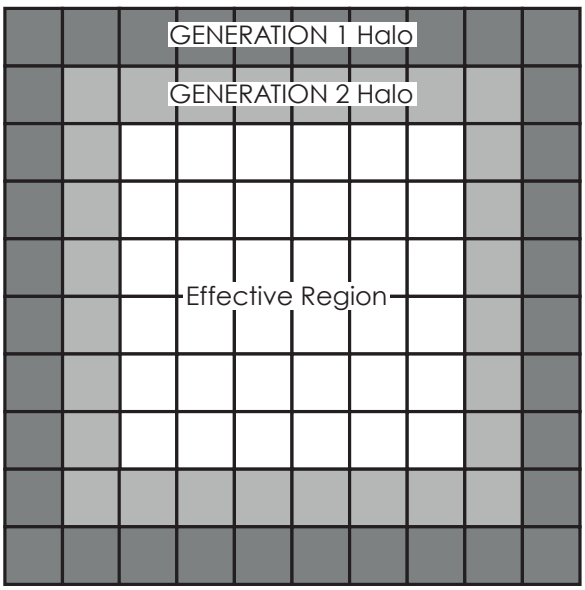


Figure 4.5: Each additional generation that a kernel computes requires more data to be read from memory. In this example, the effective region is 36 elements, but the kernel reads almost 3 times that amount: 100 elements.

of effective region size to the number of required memory loads for a kernel that computes two generations of a 16x16 effective region is 256/400, which is significantly less than the ratio for only a single generation.

The amount of shared memory used by the kernel increases when computing multiple generations because temporary space is allocated in the shared memory area removing the need to write to and then read the intermediate generation from global memory. Depending on the application, this increase in shared memory usage may make multiple generations infeasible.

Another important consideration is that of thread organization; there are two

methods which can be employed. Each threadblock can be allocated to match the effective region: a 16x16 effective region could be allocated with a 16x16 threadblock or a 64x4 region could be allocated with a 64x4 threadblock. Since an 18x18 region is required to compute an effective region of 16x16, then a threadblock with 16x16 threads requires that some threads load more than one element from global memory. Determining which threads load one element and which threads load two elements can be difficult and lead to divergent branching. The alternative is to size the threadblock based on the input region size. Creating an 18x18 threadblock ensures that each thread loads only one element from global memory, however, only 16x16 threads are required for computation, thus under-utilizing the threads in subsequent generations.

Chapter 5

Experimental Analysis

To further investigate the patterns and techniques presented in Chapter 4, a number of computations are considered, and the methods are applied successively to measure the improvements, or lack thereof. The computational models investigated, and henceforth known as *subjects*, are (1) Conway's Game of Life, and three steps of the SIFT pipeline, namely (2) Gaussian blurring implemented as a non-separable convolution, (3) the difference of Gaussians, and (4) extrema detection. This chapter introduces each subject, shows key elements from their implementations, and also presents the result of applying each technique. Results are presented for each subject independently, that is, Game of Life performance is not compared with Gaussian blurring performance. Lastly, techniques are applied in order of increasing complexity: simpler methods were attempted first, working up to the more complicated methods. Certain techniques are either not beneficial or not practical so these were not implemented for every subject; using global memory exclusively is an example of this.

5.1 Method / Setup

The machine on which all of the following experiments were run contains 2 CPUs, each of which is a 64 bit quad-core Intel Xeon E5504 processor running at 2GHz. The CPUs each have 4MB of on-chip cache and 4 hardware threads. A total of 8GB of memory is available. The GPU used in the experiments is an Nvidia GeForce GTX 285, running CUDA version 3.20. The GPU has 30 streaming multiprocessors for a total of 240 computing cores, running at 1.48 GHz. The theoretical memory bandwidth limit of this card is 159 GB / sec[1] and the card has 1GB of physical memory. At the time this research began, this was one of the more capable GPUs available; however during the course of our research, Nvidia released the Fermi architecture. It was decided that switching to a Fermi-based card would constitute a large task with few gains and therefore it was avoided. The ideas presented here are still applicable to the Fermi architecture and it is left for further work to investigate the effects of the Fermi changes such as the global memory caching facility.

Performance is measured in the standard way: $Throughput_A/Throughput_B$. This results in a mechanism where A can be said to be X times faster than B. In most cases, however, it makes more sense to give percentage improvements: a 20% improvement is easier to comprehend than a 1.2x speedup. Percentage improvements are calculated as follows: $(Time_B - Time_A)/Time_B$. Also, performance metrics are given for the largest data sets, and smaller data sets are only shown to indicate how each kernel scales. Using a GPU for computation enables the solving of much larger problems, and this is reflected by reporting metrics on the largest problems solved.

The implementations presented here are timed using features supplied by the CUDA libraries. The times given are averages of ten runs of each kernel. It is

[1] `http://http://www.nvidia.com/object/product_geforce_gtx_285_us.html`

important to note that data transfers between the host and graphics device are not included in any of the timing; only the execution time of the kernels is considered. Generally, when comparing a GPU implementation of an algorithm to a serial, CPU implementation, this time must be considered; for it is an unavoidable cost of using the GPU. It is not included here because the focus of this work is on techniques to improve memory access time on the GPU. For multi-generational execution, as in the Game of Life, the whole of the execution is counted in the timing as seen in 5.1. The memory bandwidth metrics presented measure *application bandwidth*, not the raw bandwidth on the bus. Certain factors, such as halo cell retrieval and memory coalescence, cause the application bandwidth and bus bandwidth to differ. This discussed in more detail in Section 5.2.2.

```
cudaEvent_t startEv, stopEv;
cudaEventCreate(&startEv);
cudaEventCreate(&stopEv);
cudaEventRecord(startEv, 0);

for (int i = 0; i < numGen; i++) {
  golKernel<<<dimBlocks, dimThreads>>>(dCurrent, dNext, width);
  checkCUDAError("launch");
  CUT_CHECK_ERROR("Kernel execution failed\n");

  // always swap the memory regions
  unsigned int *tmp = dCurrent;
  dCurrent = dNext;
  dNext = tmp;
}

// stop the timing
cudaEventRecord(stopEv, 0);
cudaThreadSynchronize();
float elapsedTime = 0.0f;
cudaEventElapsedTime(&elapsedTime, startEv, stopEv);
```

Figure 5.1: Multi-generational timing.

For large data sets, the inclusion of the small amount of host code in the time

values is imperceptible and ignored. It is only mentioned here because other kernels, such as the SIFT extrema detection, do not exhibit this behavior.

An important tool that was employed is the CUDA Visual Profiler [28]. This is a tool provided by Nvidia that allows for detailed inspection of various performance parameters of the kernels running on the GPU. The Visual Profiler reports data such as the number of instructions a kernel executes as well as the number of coalesced and uncoalesced memory loads. These metrics are collected using special hardware counters available on one of the streaming multiprocessors. Since the hardware counters that enable these metrics are not available on every SM, the results cannot be treated as absolute. It is possible that collection happened from a block that accomplished less work due to the data on which it was operating. While the information provided by the Visual Profile is not exact, it is still useful for analyzing trends and tendencies of the kernel. The Visual Profiler is an invaluable tool in confirming that a change made to the memory alignment does indeed result in better memory coalescence.

5.2 Game of Life

The Game of Life was chosen as a subject for two reasons: first, it is simple to understand and implement and second, it is a well known problem. This section first presents improvements that are implemented in the context of kernels that use only global memory; this is the only time that global memory exclusive implementations are provided. Next, in Section 5.2.2, improvements due to the use of shared memory are presented. Also included here are kernels with differing effective region shapes and increased arithmetic intensity. Following is Section 5.2.3, which details the experiments dealing with multi-generational kernels.

5.2.1 Global Memory

The first kernel investigated is a global memory implementation that does not include any data halo and therefore uses conditional statements to handle edge cases. This is called our *baseline* kernel for the Game of Life subject. The kernel is split into two functions: one that counts the live neighbors of a given cell and the main kernel which uses the number of live neighbors to compute the next generation. Counting the live neighbors is the step that accesses the neighborhood of a given cell, and is therefore subject to restraints dealing with the size of the memory region in question. There are two concerns here: first, if the cell is on the top or bottom row then the neighbors above and below, respectively, do not exist. Second, if a cell is in the first or last column of the data grid, then cells to the left and right, respectively, should not be considered because even though they are adjacent in memory they are not neighbors in the game world. The `GETVALUE` macro handles the top and bottom row cases by checking for invalid memory addresses. An abbreviated form of the conditional memory loading is shown in Figure 5.2. The contents of memory are values of 0 (representing a dead cell) and 1 (a live cell), randomly assigned by the host code and copied to the GPU memory before kernel execution begins. Since live cells contain a value of 1, counting them only requires the addition of neighbor values.

Every thread executes this code, even threads that are not on grid boundaries, which are the vast majority of threads for large dimensions. The conditional statements seen in Figure 5.2 are indicative of all CA models that operate on finite grid. The kernel itself is extremely simple: it uses the number of live shared neighbors to determine the value of the next generation. The rules of the Game of Life CA are such that they can be expressed with only 4 logic operations, and implemented in one line of C code as shown in Figure 5.3. Note the first three lines of Figure 5.3, as

```
__device__ int countLiveNeighbors(unsigned int *current, int idx,
                                  int width, unsigned int size)
{
  unsigned int count = GETVALUE(current, idx - width, size) // above
                     + GETVALUE(current, idx + width, size); // below
  if ((idx + 1) % width == 0) {
    // count neighbors to the RIGHT
  }
  else if (idx % width == 0) {
    // count neighbors to the LEFT
  }
  else {
    // count ALL neighbors
  }
  return count;
}
```

Figure 5.2: Conditional loading from global memory.

they show 6 arithmetic instructions used in the computation of a single index value. This subject is revisted in Section 5.2.2.

```
__global__ void golConditional(unsigned int* current,
                               unsigned int* next, int width, int size)
{
  unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
  unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
  int idx = row * width + col;
  char currentState = current[idx];

  char liveCount = countLiveNeighbors(current, idx, width, size);
  next[idx] = ( liveCount==3 ) || ( liveCount==2 && currentState );
}
```

Figure 5.3: CA rules implementation.

To illustrate the effect that conditional statements have on this kernel, another implementation is presented, in Figure 5.4, that relies on a one cell memory halo. A visual description of a one cell memory halo is shown in Figure 4.2. Not only does this kernel obviate the need for several conditional statements, it is easier to implement and understand. Without a halo, a statement such as `current[idx - stride]` may

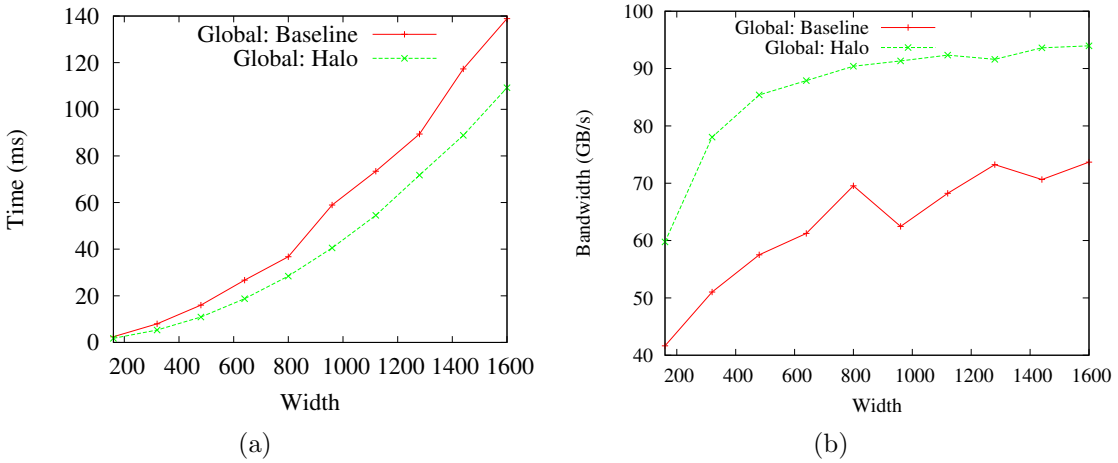
refer to an undefined memory location if the thread executing it is processing any element in the top row. The conditionals in Figure 5.2 are used to prevent this type of erroneous access. For implementation purposes, the pointer to the data, `current` in this case, actually points to the first memory location containing actual data; that is, to a location in memory just after the halo. Therefore, when accessing `current` with a negative index, such as `current[idx - stride]`, the resultant memory address is one that corresponds to an element in the halo. This is only possible if the halo region is bigger than the value of `stride`, which it is. Ensuring that every cell has a full complement of legal neighbors makes for a much simpler neighborhood access pattern with no conditional statements. See Figure 5.4, lines 8 - 12.

```
1  __global__ void golPadded(unsigned int* current, unsigned int* next,
2                            int stride)
3  {
4    unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
5    unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
6    unsigned int idx = row * stride + col;
7
8    unsigned int liveCount =
9      current[idx - stride - 1] + current[idx - stride] +
10     current[idx - stride + 1]  + current[idx - 1] +
11     current[idx + 1] + current[idx + stride - 1] +
12     current[idx + stride] + current[idx + stride + 1];
13
14   next[idx] = ( liveCount==3 ) || ( liveCount==2 && current[idx] );
15 }
```

Figure 5.4: CA computation using a one cell memory halo.

The timing results of running these two kernels are shown in Figure 5.5, times are given in milliseconds and the largest problem solved contains 1600x1600 (2,560,000 cells), and 100 generations are executed. In the largest problem, 256,000,000 iterations of the Game of Life rules are executed in approximately 100ms. As expected, the kernel that uses a one cell halo performs better. In fact, the halo-based kernel is

approximately 18% faster (for large data sets) than the one that uses conditional statements to deal with the data boundaries. The reason for this is due to the decreased number of instructions. The CUDA Visual Profiler reports that the conditional kernel executes approximately 354,114 instructions more than the halo based kernel. The kernel that uses a halo actually executes more memory load instructions (due to loading those halo cells), but it is still faster because of the significantly reduced instruction count and lack of divergent warps caused by the conditional statements.



(a) (b)

Figure 5.5: Plots of (a) execution time and (b) application memory bandwidth for conditional memory loading and using a one cell halo.

5.2.2 Shared Memory

Shared vs. Global Memory

Since many threads access a single cell, a clear improvement is the use of shared memory. Section 4.2.1 gives a detailed description of shared memory. The next kernel presented, in Figure 5.6, utilizes a one cell halo and stages all of the data required for a threadblock into shared memory before executing the computation of the next generation. The shared memory region is defined in line 1 of Figure 5.6

and its dimensions are a square region with `BLOCK_WIDTH + 2` on each side: there are (`BLOCK_WIDTH`)x(`BLOCK_WIDTH`) threads and each boundary has a halo row or column, hence the two additional rows and columns. Lines 3 and 4 offset the indices into shared memory, so that the calculated row and column for each thread match an actual data element and not a halo element; note that the index into global memory `idx` does not contain this same offset, because the halos exist only on the boundaries of shared memory. The shared memory region is of size (`BLOCK_WIDTH + 2`)x(`BLOCK_WIDTH + 2`), but there are only (`BLOCK_WIDTH`)x(`BLOCK_WIDTH`) threads in the threadblock; it follows then, that some threads need to load more than one element into shared memory. The conditional statements of lines 8 - 23 accomplish this extra loading. Threads with a column index of 0 (within the threadblock) load the halo elements immediately to their left and also at the right-side of the memory region, lines 10 - 11. Two elements are loaded in this case, and doing so reduces the need for additional conditional logic. Threads with a row index of 0 are used to load the halo elements to the top and bottom of the region, lines 15 - 16. Lastly, a single thread is used to load the 4 corners elements of the halo. The use of the `__syncthreads()` function, in line 27, acts as a barrier and forces all the threads to wait until every thread in the threadblock has reached the barrier. This ensures that shared memory is completely staged before computation begins.

Once shared memory is staged, computation can begin; the code, in Figure 5.7, is similar to the halo-based global memory kernel except that the next generation is computed using shared memory instead of global memory. Even though shared memory is being accessed, approximately 15 additional arithmetic instructions are used in the index calculations of the neighbors. The issue of unnecessary index calculations is discussed in more detail in Section 5.2.2.

```
__shared__ int s_input[BLOCK_WIDTH + 2][BLOCK_WIDTH + 2];

int tx = threadIdx.x + 1;
int ty = threadIdx.y + 1;

s_input[ty][tx] = input[idx];

if( threadIdx.x == 0 )
{
  s_input[ty][0] = input[idx - 1];
  s_input[ty][blockDim.x+1] = input[idx + blockDim.x];
}
if( threadIdx.y == 0 )
{
  s_input[0][tx] = input[idx - width ];
  s_input[blockDim.y + 1][tx] = input[idx + blockDim.y * width];
}
if( threadIdx.x == 0 && threadIdx.y == 1 )
{
  s_input[0][0] = input[idx - 2* width - 1];
  s_input[0][blockDim.x+1] = input[idx - 2* width + blockDim.x];
  s_input[blockDim.y+1][0] = input[idx + (blockDim.y-1)*width - 1];

  s_input[blockDim.y+1][blockDim.x+1] =
    input[idx + (blockDim.y-1)*width + blockDim.x];
}
__syncthreads();
```

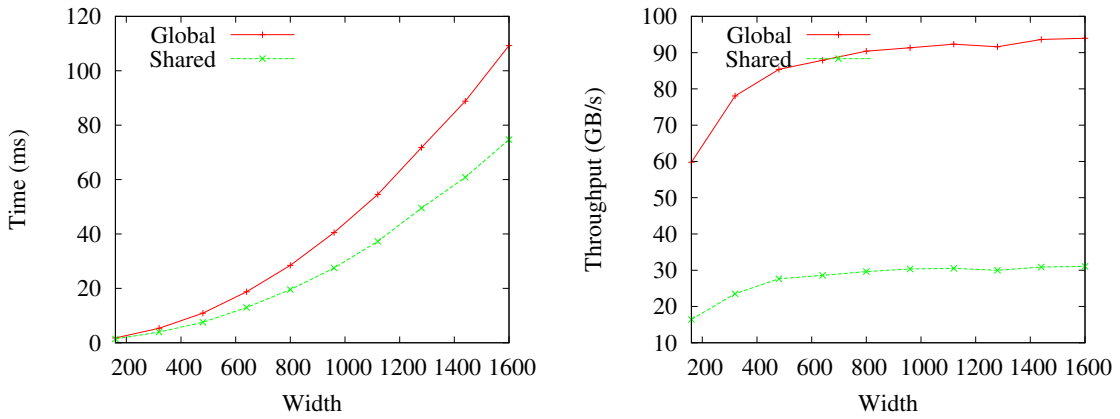
Figure 5.6: Shared memory staging, assumes one cell halo.

The introduction of shared memory significantly increases performance, as depicted in Figure 5.8. Since these kernels both use a one cell halo around the data, the performance difference is accounted for solely by the use of shared memory. The introduction of shared memory garners an approximately 30% performance improvement. A closer inspection is required to determine the source of the performance improvement. In the global memory version each thread makes 9 read requests of global memory, resulting in a total of $9 * 256 = 2304$ total requests. In the shared memory kernel, each thread requests one element, resulting in a total of 256 global memory requests. Therefore, due to the high latency of each global memory request,

```
int neighbor_count = s_input[ty-1][tx-1] + s_input[ty-1][tx] +
  s_input[ty-1][tx+1] + s_input[ty][tx-1]  + s_input[ty][tx] +
  s_input[ty][tx+1] + s_input[ty+1][tx-1] + s_input[ty+1][tx] +
  s_input[ty+1][tx+1];

 output[idx] = (neighbor_count==3) ||
               (neighbor_count==2 && s_input[ty][tx]);
```

Figure 5.7: CA computation using shared memory.

the shared memory kernel spends significantly less time waiting for data to be returned thus improving performance. The shared memory implementation exploits the *spatial locality* pattern of a neighbor based calculation.



(a) (b)

Figure 5.8: (a) total time and (b) application bandwidth for global and shared memory.

Aligned Memory

The next improvement to discuss is that of memory alignment, which is explained in Section 4.2.2. The kernels described above use a halo, but do not add lead or row padding, thus rendering the bulk of data unaligned. The idea is to pad the memory so

that all accesses to the actual data are aligned, but the requests for the halos access unaligned memory. To accomplish this an initial lead pad is added to the beginning of the memory area, and each row has padding at the end. An example of how padding is implemented can be seen in Figure 5.9. To include a halo, the width and height of the region are increased, as seen in line 1. The `row_pad` variable, in lines 3 - 4, is used to represent the bytes that get added to the end of the row. For example, if a row contained 60 bytes, 4 bytes are added to the end ensuring that every element in the next row is aligned. The right-side halo is incorporated in the `row_pad`. Lines 5 and 6 contain a `lead_pad` element that handles the left-side halo element. The halo in question is one cell and the `lead_pad` variable establishes that this initial halo cell is the last byte in an aligned region. Adding the lead pad causes the byte after the first halo element to be the first byte of an aligned region. In line 8, an `offset` to the first element of actual data is calculated, and this offset is used when passing the memory address of the data region to the kernel.

```
unsigned int actualWidth = width + 2;                                    1
                                                                         2
int row_pad    = ((actualWidth + segmentWidth - 1) / segmentWidth ) *    3
                 segmentWidth - actualWidth;                             4
int lead_pad   = segmentWidth - 1;                                       5
int num_bytes = (actualWidth * (actualWidth + row_pad) + lead_pad) *     6
                 sizeof(unsigned int);                                   7
int offset = 1 + lead_pad + actualWidth + row_pad;                       8
unsigned int *dCurrent, *dNext, *hostRegion;                             9
CUDA_SAFE_CALL(cudaMallocHost((void**)&hostRegion, num_bytes));          10
                                                                         11
// other initialization                                                  12
                                                                         13
golPadded<<<dimBlocks, dimThreads>>>(dCurrent+offset, dNext+offset,      14
                                     actualWidth +row_pad);              15
```

Figure 5.9: Padding memory to maximize aligned memory accesses.

A benefit of aligning the memory in such a fashion is that, assuming the kernel is

expecting the presence of a halo, the implementation of the kernel does not change. What changes is the alignment of the data in global memory, but that is transparent to the kernel. Now, a thread in column 0 reads an element that exists on a 64-byte boundary. The next thread, column 1, reads the next element, which is contiguous to the previous element on the 64-byte boundary. Each successive thread requests the next element, resulting in a single, contiguous 64-byte block of memory that starts on 64-byte boundary, to be requested. The memory subsystem coalesces all these requests into a single response, returning the data in a single bus transaction. However, the halo cells are in unaligned positions, and reading them results in uncoalesced accesses. But, since halo access only happens for threads on the boundary, the cost is small over lifetime of the kernel.

The code in Figure 5.6 is also used for an aligned memory kernel, and it is useful to indicate the aligned and unaligned accesses that this kernel makes. Since the `input` variable points to the first element of real data, and exists on an aligned 64-byte boundary, the initial load in line 6, results in coalesced memory access for the entire block, resulting in 16 bus transactions. Loading the halos is more complicated. Threads with a column index of 0 load the halo element immediately to their left and also at the right-side of the memory region, lines 10 - 11. These memory loads are uncoalesced: the element to the left is not aligned on an even boundary; the element to the right is on an even byte boundary, but the following bytes are not requested or used. These unused bytes are further investigated in Section 5.2.2. Threads with a row index of 0 load the halo elements immediately above and below, lines 15 - 16. Loading the above and below halo elements results in coalesced reads because these bytes begin on a 64-byte boundary. The 4 corners of the halo are loaded by a single thread, resulting in 4 uncoalesced memory reads, lines 20 - 25. Reading the halo values increases the number of uncoalesced memory accesses, however, the increase

is small because these loads only happen for threads on the perimeter of the memory region.

Aligning the memory on a larger boundary, for example 32 bytes vs. 64 bytes, enables the memory subsystem to return the data in fewer requests, thus reducing latency and execution time. Figure 5.10 shows the times for alignment boundaries; as expected, 128-byte alignment performs the best while 32-byte alignment is the slowest. Reading 256 bytes from unaligned memory requires 256 bus transactions, while reading from a 64-byte aligned region requires just 16 bus transactions, and a 128-byte aligned region incurs only 2 bus transactions. Clearly, memory alignment is a critical element in improving performance. The remainder of the kernels presented for Game of Life are aligned on a 128-byte boundary.

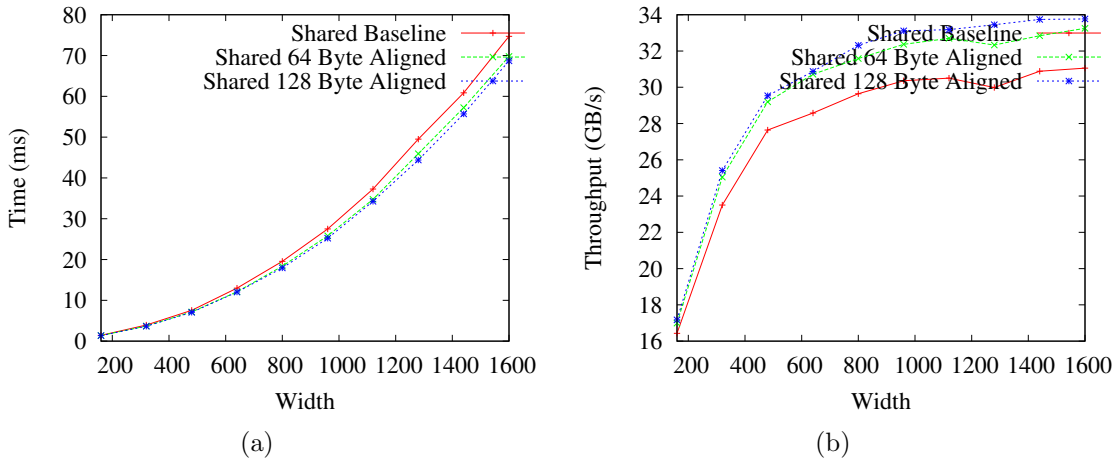


Figure 5.10: A comparison of different memory alignments. 32-byte alignment is considered to be unaligned while 128-byte alignment is maximally aligned. Total time is shown in (a) and application bandwidth is shown in (b).

Memory Region Shape

A further improvement implemented is modifying the "shape" of the threadblocks, thereby changing the specific cells that are accessed. The technique involves using wider rectangularly shaped threadblocks, for example, 4 threads high by 64 threads wide, while the above kernels use a 16x16 threadblock. Both threadblocks contain a total of 256 threads, therefore both kernels still process the same number of elements. However, as shown in Section 4.2.4 the wide rectangular kernels cause less halo traffic over the bus. Section 5.2.2 demonstrated that right-side halos incurred unaligned memory access and wasted bus traffic. Each bus transaction transfers a minimum of 32 bytes and right-side halo accesses are requesting only 1 element, or 4 bytes, incurring a waste of 28 bytes. Therefore, minimizing the ratio of right-side halo loads to above/below halo loads will reduce the amount of wasted bus traffic, thus improving throughput and performance. Similar to the move from unaligned to aligned memory as described in Section 5.2.2, the code for a wide rectangular kernel is almost exactly the same as a square kernel, except that dimensions of the shared memory arrays are different; the rest of the changes are handled by the hardware because the threadblocks are sized differently. There is one minor drawback to this approach: more shared memory is used. A 16x16 effective region requires an 18x18 shared memory array, or 324 elements. A 64x4 rectangular region requires a 66x6 shared memory region, or 396 elements. Depending on the type of data being used this slight increase could cause a reduction in occupancy, a metric further discussed in Section 5.3.3. Using a rectangular region yields positive results that can be seen in Figure 5.11.

The plot in Figure 5.11 also contains the baseline shared memory implementation

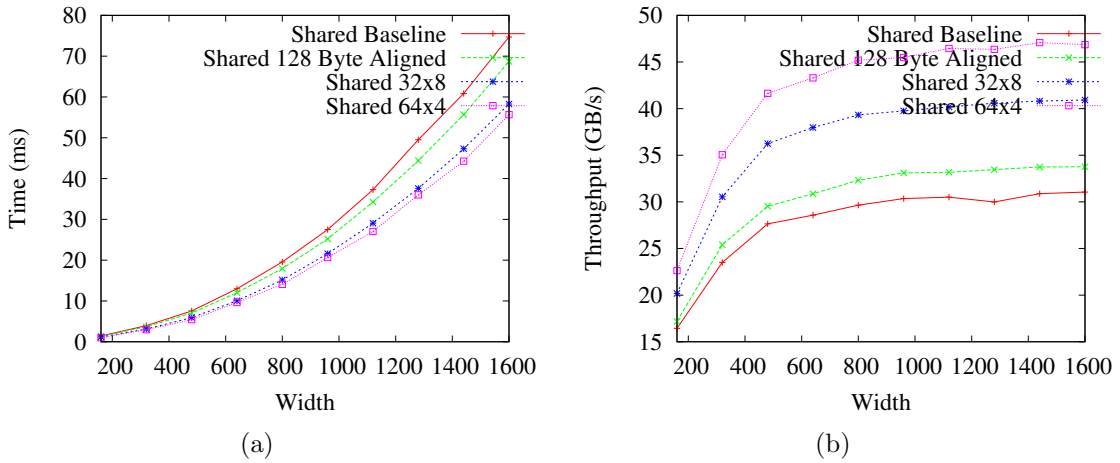


Figure 5.11: Rectangular vs. square regions: (a) total time, (b) application bandwidth.

and the 128-byte aligned implementation, since it had been the fastest one shown. However, it is clear that the 64x4 rectangular region is now the winner; it has 4 less rows, and thus 4 times less halo data than the 8x32 implementation, and is therefore just slightly faster. The 64x4 implementation is approximately 16% faster than the 128-byte aligned kernel, a significant improvement. A closer analysis reveals the actual savings a rectangular region achieves: a 16x16 block makes 16 right-side halo element requests, resulting in $16 * 28 = 448$ wasted bytes, per block, while a 64x4 block only makes 4 right-side halo requests, resulting in $4 * 28 = 112$ wasted bytes. Both kernels require 10,000 blocks, but the rectangular block wastes only 1,120,000 bytes via right-side halo loading, while the square block wastes 4,480,000, yielding a 75% reduction in wasted bus traffic. The rectangular region kernels operate on aligned data, so this method is used in conjunction with the aligned memory method.

Two Elements Per Thread

The final technique employed for shared memory is one that increases the arithmetic intensity and reduces index calculation overhead, and is fully detailed in Section 4.3.1. Implementation of this techniques requires a shared memory region that is twice the size of the one used in previous techniques because each thread processes two separate elements. Loading twice the data into shared memory is straightforward but requires additional code that is detailed in Figure 5.12. As shown in Section 5.2.2, using a wider region reduces wasted bus traffic and improves performance. A similar approach is taken here: a 32x16 block, or in the case of a rectangular block, a 128x4 region is staged into shared memory. Note that in line 4, the rows to accommodate halo cells are still included, resulting in a 34x18 or 130x6 region for square and rectangular kernels, respectively. Each thread calculates the index of the element upon which it will operate (line 1), that element is then loaded, as demonstrated in previous kernels, resulting in a coalesced memory access. A new index is then calculated as shown in line 10, and the element at the new index is loaded. Note that the new index is relative to the initial index and is exactly BLOCK_WIDTH elements away; since BLOCK_WIDTH is defined to be a multiple of 16, this next memory load is also coalesced. The additional cells must also be loaded for the halos as well (not shown). The loading of the halo cells follows a similar pattern to that shown in Figure 5.6: left and right halos incur uncoalesced loads, while the top and bottom halo loads are coalesced. For the computation step, after the original index is computed and results written to global memory, the second element is computed using the new indices in the same fashion, also not shown.

Processing two elements per thread significantly reduces the number of instruc-

```
unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
int idx = row * width + col;
__shared__ int s_input[BLOCK_WIDTH + 2][2 * BLOCK_WIDTH + 2];

// load the first element
s_input[ty][tx] = input[idx];

// load the second element
s_input[ty][tx + BLOCK_WIDTH] = input[idx + BLOCK_WIDTH];

...
```

Figure 5.12: Loading shared memory such that each thread processes two elements.

tions executed over the life of the kernel launch. To illustrate this, an example is given. Assume that each index calculation takes 15 total instructions, including load instructions. Over the life of kernel operating on a 1600x1600 data set, there are 2,560,000 such index calculations that require a total of 38,400,000 instructions. When processing two elements per thread, the initial index calculation still remains, 15 instructions, and the second index calculation must occur resulting in a total of 16 instructions. However, since each thread processes two elements, half as many threads are required, resulting in a total of 20,480,000 instructions dedicated to index calculations: approximately a 47% improvement. The two-element kernel presented here also operates on an aligned memory, making it suitable to compare to both the basic 128-byte aligned version and the rectangular region. A comparison of these techniques is depicted in Figure 5.13.

From the plot in Figure 5.13, it is clear that the two-element kernel is much faster. In fact it is approximately 44% faster than the base 128 byte aligned kernel and 36% faster than the 64x4 rectangular region kernel. The improvement stems from the significant reduction in executed instructions.

Each technique has now been illustrated, however there is still one further im-

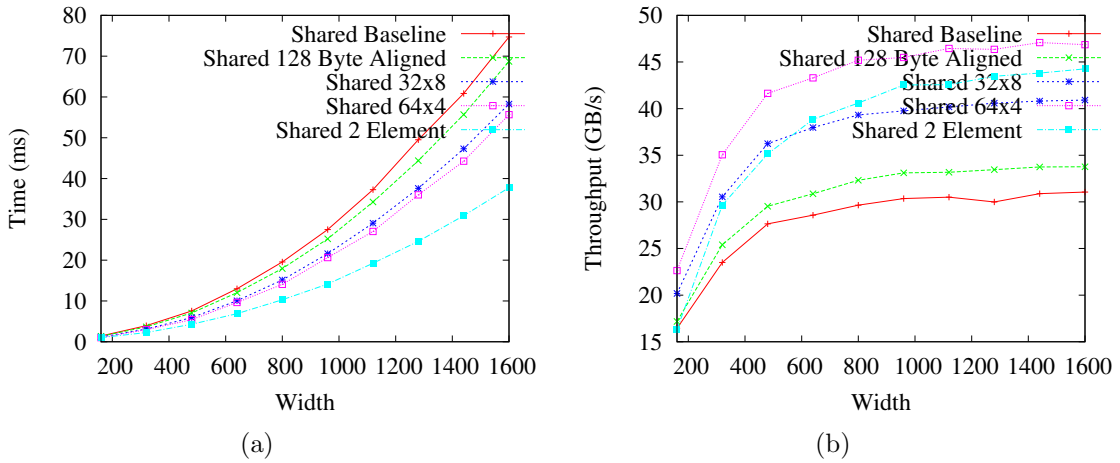


Figure 5.13: A comparison of a two-element kernel, a rectangular region kernel and a basic 128 byte aligned kernel: (a) shows total time and (b) depicts application bandwidth.

provement to make: combining the two-element and rectangular region techniques in a single kernel. Implementation of this simply entails modifying the two-element technique kernel to operate on a differently sized shared memory region as well as modifying the threadblock allocation. The results of this combination are shown in Figure 5.14.

The combination of these two techniques yields an addition 5% speed increase over the two-element kernel. The combination of these two results accomplishes two things: first, a significant reduction in the number of overhead instructions dedicated to index calculations; and second, a reduction in the number of wasted bytes due to right-side halo loading. The instruction reduction is significantly larger than the wasted bytes, and as such, dominates the improvement, resulting in only a small improvement over the two-element technique alone.

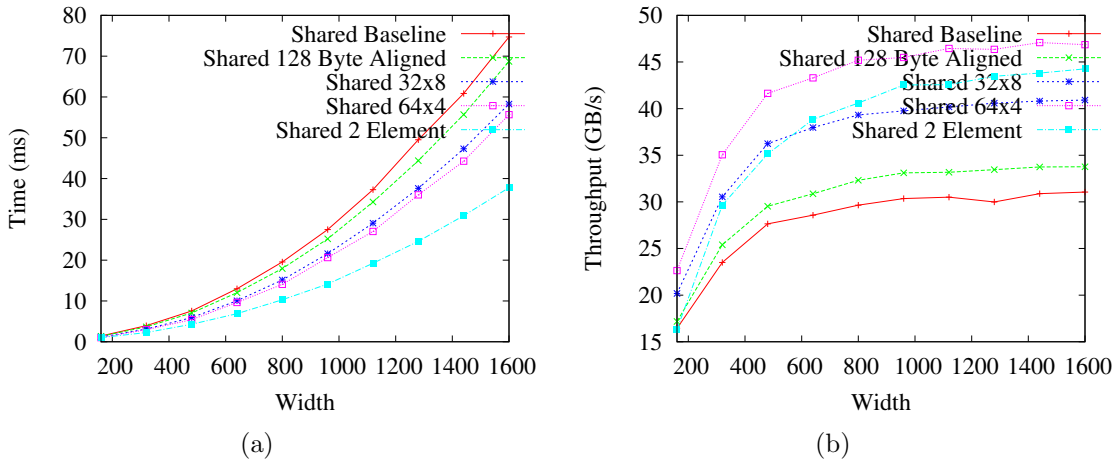


Figure 5.14: Execution times of all the major kernels discussed to this point; (a) is total time while (b) is application bandwidth. The two-element rectangular region kernel is the clear winner

5.2.3 Multi-generational Kernels

Before concluding the Game of Life investigation, there is one more topic that must be covered: multi-generation kernels. The details of multi-generational kernels are given in Section 4.3.3, and the essence of this technique is to increase arithmetic intensity by reducing the number of memory transactions. For example, if a kernel can compute one generation by loading 324 elements then it takes 768 loads to compute two generations. But, it is possible to compute the first generation and since the results are readily available, immediately compute the generation. This requires more memory loads initially, 400 for example, but still less than two iterations of a single generation kernel. In Section 4.3.3, it is stated that there are two methods of loading the required extra data in the two generation kernel: (1) 256 threads load 400 elements or (2) 400 threads load 400 elements, but only 256 thread are used for computation. The data presented here is the result of implementations where the effective region matches the threadblock dimensions; meaning that there are only 256 threads, some

of which load, and process, more than one value. The extra bytes are halo bytes, and they are loaded as shown previously.

```
s_input[ty][tx] = input[idx];                                                    1
                                                                                 2
if( threadIdx.x == 0 ) {                                                         3
   s_input[ty][0] = input[idx - 2];                                              4
   s_input[ty][1] = input[idx - 1];                                              5
   s_input[ty][blockDim.x+1] = input[idx + blockDim.x];                          6
   s_input[ty][blockDim.x+2] = input[idx + blockDim.x + 1];                      7
}                                                                                8
if( threadIdx.y == 0 ) {                                                         9
  // similar to previous if statement except get above/below                     10
}                                                                                11
// now load the corners                                                          12
if( threadIdx.x == 0 && threadIdx.y == 1 ) {                                     13
   // column 0                                                                   14
   // these loads are UNCOALESCED                                                15
   s_input[0][0] = input[idx - 3* width - 2];                                    16
   s_input[1][0] = input[idx - 2* width - 2];                                    17
   s_input[blockDim.y+1][0] = input[idx + (blockDim.y-1)*width - 2];             18
   s_input[blockDim.y+2][0] = input[idx + (blockDim.y)*width - 2];               19
                                                                                 20
   // column 1                                                                   21
   // these loads are UNCOALESCED                                                22
   s_input[0][1] = input[idx - 3* width - 1];                                    23
   s_input[1][1] = input[idx - 2* width - 1];                                    24
   s_input[blockDim.y+1][1] = input[idx + (blockDim.y-1)*width - 1];             25
   s_input[blockDim.y+2][1] = input[idx + (blockDim.y)*width - 1];               26
                                                                                 27
   // similar for column blockDim.x andy blockDim.x + 1                          28
   ...                                                                           29
}                                                                                30
__syncthreads();                                                                 31
```

Figure 5.15: The complicated process of staging shared memory in preparation for a two-generation kernel.

A portion of the code for loading shared memory can be seen in Figure 5.15. Line 1 starts with a standard memory load, which retrieves an actual data element from the effective area. The following loads are dedicated to the halo, which is a now a two cell halo: one for the first generation and another for the second generation. Lines 4

- 7 show that instead of loading only the left halo element, a single thread now loads the two elements on the left, and two elements on the right. These additional loads are uncoalesced. Since a two cell halo is being used, each corer now consists of 4 cells, resulting in a total 16 cells at the corners, shown in lines 16 and following. Notice, the index calculations that are computed when loading the corners of the halos: up to 5 extra arithmetic instructions are required per index. These extra index calculations play an important role in explaining the results, shown below in Figure 5.17. The loading of only two corners is shown in Figure 5.15. Also, it is important to point out that all of the two-generational kernels operate on a rectangular memory region, due to the benefits detailed in Figure 5.13. Figure 5.16 shows a comparison of the baseline two-generation implementation and the two-element rectangular kernel implementation detailed in Section 5.2.3.

Even with half the kernel launches, the two-generational kernel takes considerably

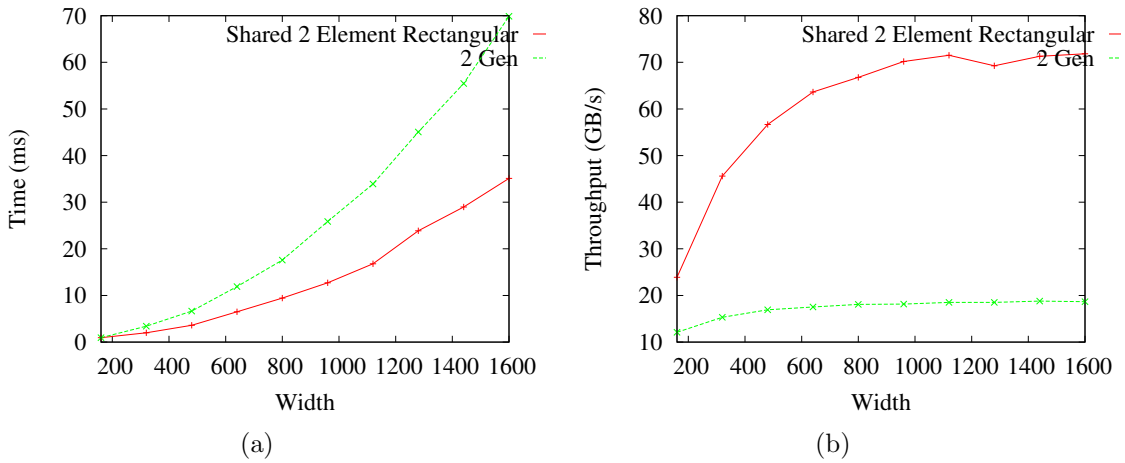


Figure 5.16: Comparing the two-element rectangular kernel against a baseline two-generation kernel: (a) is total time and (b) is application bandwidth.

longer than the two-element rectangular kernel. To further investigate the performance of each kernel, the CUDA Visual Profiler tool was used. This tool is described

in Section 5.1. A quick analysis using the Visual Profiler reveals a likely cause: one launch of the two-generation kernel executes more instructions than two launches of the two-element, single-generation, rectangular kernel. Table 5.1 shows the data captured by the Visual Profiler; the two-element kernel executes 100 times, while the two-generation kernel executes 50 times, both resulting in 100 generations.

Table 5.1: Two-element and two-generation Visual Profiler Results

	Two-Element	Two-Generations
Instructions	259,276	563,418
Coalesced Memory Loads	36,000	32,000
Uncoalesced Memory Loads	36,000	32,000

The data shows that the number of coalesced and uncoalesced memory reads each kernel makes is similar, however the two-generation kernel executes a significantly larger number of instructions. Code analysis shows that these instructions are the result of index calculations, which arise from two places: (1) loading of more than one element (all of which are halo elements) by the majority of threads, and (2) calculating the first generation which requires calculating the values for the inner-most halo. The halo elements are not represented by their own thread, therefore each access to a halo element incurs up to 5 extra instructions of index calculations. An added problem is that there are more halo elements to process: the rectangular one generation kernels process 140 halo cells, but the recantular two-generation kernel processes 286 halo cells: a 100% increase. The code for calculating the first generation of the halos can be seen in Figure 5.17. Line 2 shows the standard Game of Life computation to compute the elements in the effective region. Line 10 starts the calculations for the halos: note this process is repeated for each of the first generation halo cells (not shown). The number of index calculations required is significantly higher.

Any access to a halo cell requires some form of index calculation; some of these

```
// first gen for effective region
int neighbor_count = s_input[ty-1][tx-1] + s_input[ty-1][tx] +
   s_input[ty-1][tx+1] + s_input[ty][tx-1]  + s_input[ty][tx] +
   s_input[ty][tx+1] + s_input[ty+1][tx-1] + s_input[ty+1][tx] +
   s_input[ty+1][tx+1];

s_temp[ty - 1][tx - 1] =
   (neighbor_count==3) || (neighbor_count==2 && s_input[ty][tx]);

// now set the halos for the first gen result.
if( threadIdx.x == 0 )
{
   // count for the cell to the left
   neighbor_count = s_input[ty-1][tx-2] + s_input[ty-1][tx-1] +
      s_input[ty-1][tx] + s_input[ty][tx-2]  + s_input[ty][tx-1] +
      s_input[ty][tx] + s_input[ty+1][tx-2] + s_input[ty+1][tx-1] +
      s_input[ty+1][tx];

   s_temp[ty - 1][tx - 2] =
      (neighbor_count==3) || (neighbor_count==2 && s_input[ty][tx-1]);
}

...

neighbor_count = s_temp[ty-2][tx-2] + s_temp[ty-2][tx-1] +
   s_temp[ty-2][tx] + s_temp[ty-1][tx-2] + s_temp[ty-1][tx-1] +
   s_temp[ty-1][tx] + s_temp[ty][tx-2] + s_temp[ty][tx-1] +
   s_temp[ty][tx];

output[idx] = (neighbor_count==3) ||
              (neighbor_count==2 && s_temp[ty-1][tx-1]);
```

Figure 5.17: Calculation of first generation halo cells.

calculations require 5 or more arithmetic operations. It is easy to see why this kernel has such a high number of instructions. One way to reduce the number of instructions is to use constant values where possible. Notice that some of the code refers to `blockDim.x` which is a built-in variable that stores the number of threads in the x-direction. However, the number of threads is known at compile-time, so it is possible to replace these with constant values, and doing so achieves a modest performance improvement.

The above multi-generational kernels use the shared loading scheme presented in the shared memory Section 5.2.2. The scheme uses threads on the top row to load the top halo, threads on the bottom row to load the bottom halo, and similar for left and right. It then uses a single thread to load the corners. This mechanism maximizes memory coalescence, however, as seen above, it requires a substantial increase in arithmetic operations due to index calculations. An alternative is to "re-center" the threads so they start at the first cell of the halo. This creates a stencil that covers the first 64x4 rows and columns of a 68x8 region. Once those cells have been loaded, a relative index is calculated and the threads each load one more value. This method reduces memory coalescence but improves instruction count and is known as *linear* loading. An example implementation of this method can be viewed in Figure 5.18. The relative index calculation is similar to the two-element kernels presented in Section 5.2.2. Lines 14 - 15 show the relative index calculation. Since the number of halo cells required is not exactly $2 * numThreads$ a conditional statement is required, as shown in line 17. The linear loading method removes the need to check if a thread is on the top row, left column, or bottom row, and in doing so reduces the number of conditional statements.

Performance results of the linear method are shown in Figure 5.19, and the performance improvement is clear. Using the Visual Profiler it is easy to see why the

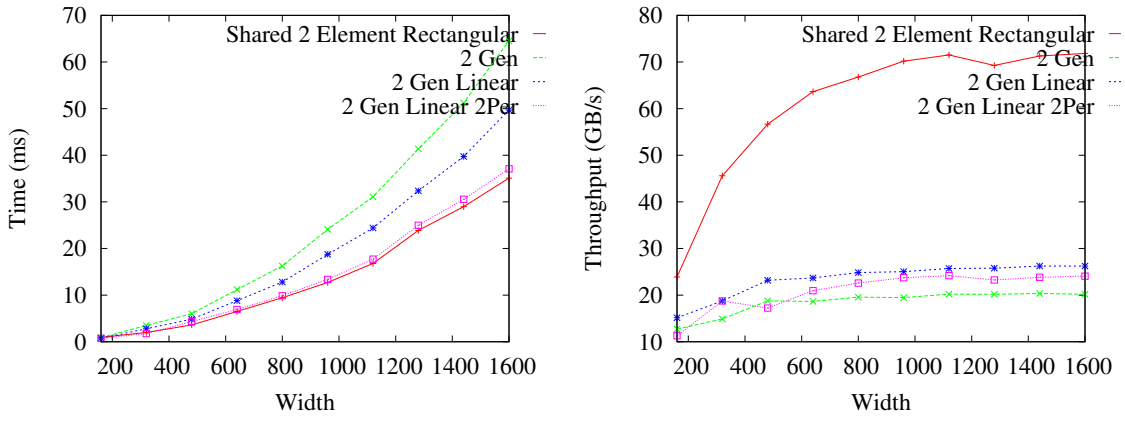
```
unsigned int srow = 1 + threadIdx.y;
unsigned int scol = 1 + threadIdx.x;

int neighbor_count
  = s_input[srow-1][scol-1] + s_input[srow-1][scol] +
  s_input[srow-1][scol+1] + s_input[srow  ][scol-1] +
  s_input[srow  ][scol+1] + s_input[srow+1][scol-1] +
  s_input[srow+1][scol] + s_input[srow+1][scol+1];

s_temp[srow - 1][scol - 1] =
   (neighbor_count==3) || (neighbor_count==2 && s_input[srow][scol]);

// now the second value for the first generation, based on index
srow += blockDim.y;
scol += blockDim.x;

if(srow <= HEIGHT && scol <= WIDTH){
  neighbor_count =
    s_input[srow-1][scol-1] + s_input[srow-1][scol] +
    s_input[srow-1][scol+1] + s_input[srow][scol-1] +
    s_input[srow  ][scol+1] + s_input[srow+1][scol-1] +
    s_input[srow+1][scol] + s_input[srow+1][scol+1];

  s_temp[srow - 1][scol - 1] =
    (neighbor_count==3) || (neighbor_count==2 && s_input[srow][scol]);
}
```

Figure 5.18: Linear loading method.

performance of the linear loading method is better. Table 5.2 shows a comparison of these two methods, for a single execution of the kernel. Linear loading executes approximately 110,000 less instructions, for a single kernel launch. This equates to approximately 5,500,000 less instruction over the course of computing 100 generations. The number of uncoalesced memory accesses does not change, thus owing the entirety of the improvement to the reduction in instructions. Also shown in Figure 5.19, is a two-element version of linear loading. This two-element kernel is a combination of the linear loading method and the two-element kernel presented in Section 5.2.2, which stated that the performance improvement is largely due instruction reduction due to fewer index calculations. The reason for the drastic improvement of the two-element linear method has not changed: after executing all those index calculations, a single add instruction enables the kernel to correctly access another element at lower instruction cost. However, the two-element linear loading for two-generation kernels does not perform quite as well as its single generation counterpart, although it does come close. Table 5.2 gives some insight as to what is slowing down the two generation kernel: instruction count. The single generation kernel still execute less instruction across the computation of all generations, and since the number of memory loads, coalesced and otherwise, is virtually identical, the number of instructions is the limiting factor for the two generation kernel. Further work should investigate the alternative method of using more threads to compute a smaller effective region: 16x16 threads that compute two-generations of a 12x12 region, for example.

Table 5.2: Visual Profiler Results for two-generation Game of Life kernels.

	Two-element	2 Gen.	2 Gen. Linear	2 Gen. Two-element Linear
Instructions	259,276	528,483	410,595	269,925
Coalesced Loads	36,000	32,000	32,000	30,720
Uncoalesced Loads	36,000	32,000	32,000	15,360



(a) (b)

Figure 5.19: Performance of the two-generation kernels. (a) total time and (b) application bandwidth.

5.3 Image Processing Methods

In the previous Section 5.2, all of the techniques and patterns are detailed and source code is supplied where appropriate, save one: data packing. The research into the SIFT pipeline was completed after the Game of Life work and therefore the lessons learned while implementing the various kernels were applied; meaning that less effective techniques were not included here. For example, none of the rectangular kernels operate on a 32x8 region since that is not effective as a 64x4 region. This section does not repeat information presented previously, and source code is only provided when it makes sense to do so. Also, this section is not organized by technique, but by subject: Gaussian blur, difference of Gaussians, and extrema detection.

5.3.1 Gaussian Blur

Gaussian blurring is an example of a convolution operation. When a function is convolved with the Gaussian function, $\mathrm{P(x)} = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$, the result is a general smoothing of the data. The Gaussian function is used in many computer vision

applications, including SIFT, to reduce initial image noise and soften edges, and it is integral in creating the scale space. SIFT requires that Gaussian blurring is done on an image, 48 times under normal circumstances, before the difference of Gaussians can be computed.

Convolution operations have been implemented many different ways, even on GPUs. In fact, the CUDA SDK released by Nvidia contains a sample convolution kernel. Fast Fourier transforms have been used, as well as standard matrix multiplication. To compute a convolution, one multiplies what is known as a *kernel* (different than GPGPU programming), which is simply a square matrix of values, by the value of a function, and its neighbors. In the case of an image, the value of the function is a pixel value, and its neighbors are the surrounding pixels. This operation has a very similar pattern to cellular automata computations, especially with respect to memory access, which is why it has been included here. It is important to note that the Gaussian blurring implementation presented here is not implemented as a separable convolution. A separable implementation would indeed be faster, but the techniques presented here would not be quite as effective and their effects less demonstrable.

```
float value =                                                                    1
   k0*tile[sharedIdx - SHARED_WIDTH - 1] +                                       2
   k1*tile[sharedIdx - SHARED_WIDTH] +                                           3
   k2*tile[sharedIdx - SHARED_WIDTH + 1] +                                       4
   k3*tile[sharedIdx - 1]+k4*tile[sharedIdx] + k5*tile[sharedIdx + 1] +          5
   k6*tile[sharedIdx + SHARED_WIDTH - 1] +                                       6
   k7*tile[sharedIdx + SHARED_WIDTH] +                                           7
   k8*tile[sharedIdx + SHARED_WIDTH + 1];                                        8
                                                                                 9
output[idx] = fabs(value);                                                       10
```

Figure 5.20: Core of the convolution kernel.

The core of the convolution kernel can be viewed in Figure 5.20. Each thread

multiplies elements of the convolution kernel (the supplied 3x3 matrix of values) by a data element and its neighbors, and the result is written to global memory. Lines 2 - 8 access the shared memory area named `tile`, which contain a cell and its neighbors, by the appropriate value of the convolution kernel. The absolute value of the result is then written to appropriate address in global memory. Figure 5.20 does not display the shared memory loading, which is done exactly as demonstrated in the Game Of Life kernels, e.g. Figure 5.6. The Gaussian blur kernel operates on a 64x4 rectangular region of aligned memory. The only improvement to be made, based on the Game of Life results, is processing two elements per thread. Figure 5.21 depicts a comparison of the two blur kernels: aligned rectangular vs. two-element aligned rectangular. As expected, the two-element kernel performs significantly better. The reason for this improvement is the same as above: a significant reduction in instruction count. The two-element blur kernel executes 166,270 instructions less than the single element variant while incurring less than half the uncoalesced memory loads.

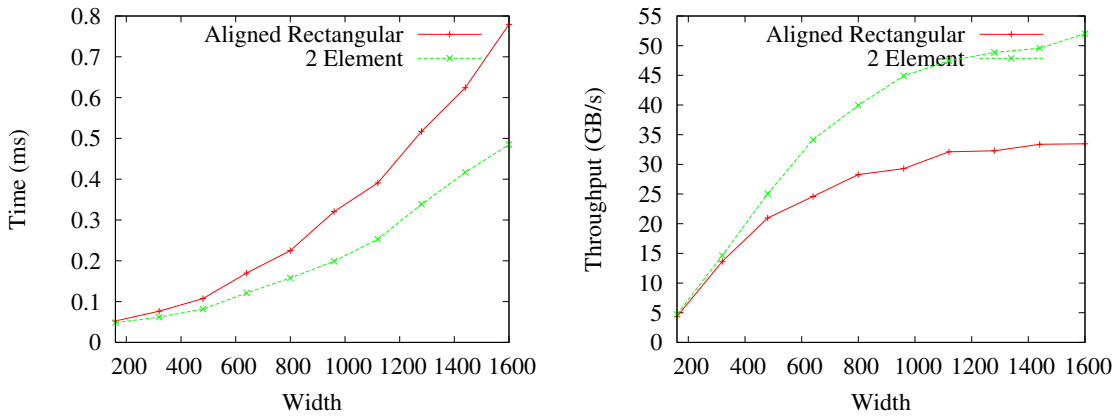


Figure 5.21: Comparison of blur kernels: aligned rectangular and two-element aligned rectangular; (a) is total time and (b) is application bandwidth.

5.3.2 Difference of Gaussians

Computing a "difference of Gaussians" involves taking the output of one blurring operation and performing a cell-by-cell subtraction with the output of another blurring operation. Implementation of such an operation, serially or on the GPU, is trivial and is shown in Figure 5.22.

This kernel is so compact there is not much that can be optimized. Since this kernel

```
__global__ void d_dogGlobal(float *img1, float *img2, float *out,
                            int h, int stride)
{
  int r = blockIdx.y * blockDim.y + threadIdx.y;
  int c = blockIdx.x * blockDim.x + threadIdx.x;
  int idx = r * stride + c;
  if (r < h && c < stride)
    out[idx] = img2[idx] - img1[idx];
}
```

Figure 5.22: Entirety of the difference of Gaussians kernel.

does not require a halo, many of the techniques, including the rectangular region, do not apply. Assuming the memory is already aligned, the only technique demonstrated so far that can be applied is the two-element technique, and the results of applying this technique are shown in Figure 5.23. However, there is one last technique mentioned in Section 4.3.2 that has not yet been detailed: data interleaving and packing. Data packing allows each memory request to return more data. Memory requests are made on a half-warp boundary and there are 16 threads in a half warp thus it follows that when requesting `float` data, 64 bytes are returned. The Nvidia memory bus can support 128 bytes in one request, however. To fully realize this bandwidth, each thread must request not 4, but 8 bytes of data. CUDA provides a `float2` which packs two floats into one 8 byte word as a vector type. Using a `float2` reduces the number of bus transfers by half for the difference of Gaussians kernel. Figure 5.23 also includes

the performance data for the data packing kernel. The different implementations are barely discernable in Figure 5.23 due to multiple reasons. First, the effective computation of the kernel is just one instruction, a subtract instruction. Second, there are no halos to introduce an easily measurable latency. Third, there is no need for shared memory. The kernel is so simple that once the memory is aligned only small improvements can be made. However, the CUDA Visual Profiler can provide a clear approximation to what is happening on the device and from that information it is possible to extrapolate the results for more complicated and mathematically intensive kernels. Table 5.3 shows exactly what is expected: the two-element kernels execute fewer instructions and the packed data kernels require half the memory loads. Therefore, we surmise that these techniques, when applied to more intensive problems, will yield positive results.

Table 5.3: Visual Profiler results for difference of Gaussians.

	1 Element	Two-element	Packed	Two-element Packed
Instructions	64,160	39,712	61,304	35,872
Coalesced Loads	32,000	30,270	16,000	30,720
Uncoalesced Loads	36,000	32,000	16,000	15,360

5.3.3 Extrema Detection

Once the difference of Gaussians has been computed, each element in the difference is tested for the presence of an extrema. This is done by checking that element against all of its neighbors, both in its own interval and the two surrounding intervals: if an element is the maximum or minimum value of its neighborhood, it is considered to be an extrema. The detection is a shown more closely in Figure 2.3, and based on the figure it is natural to implement the neighborhood as a three dimensional array. In implementing the extrema detection, 4 separate kernels were written: aligned, aligned

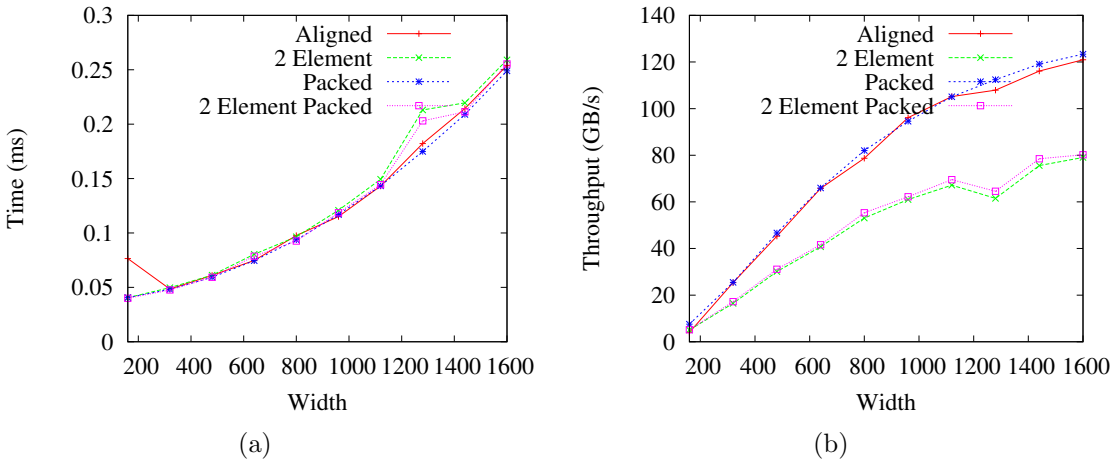


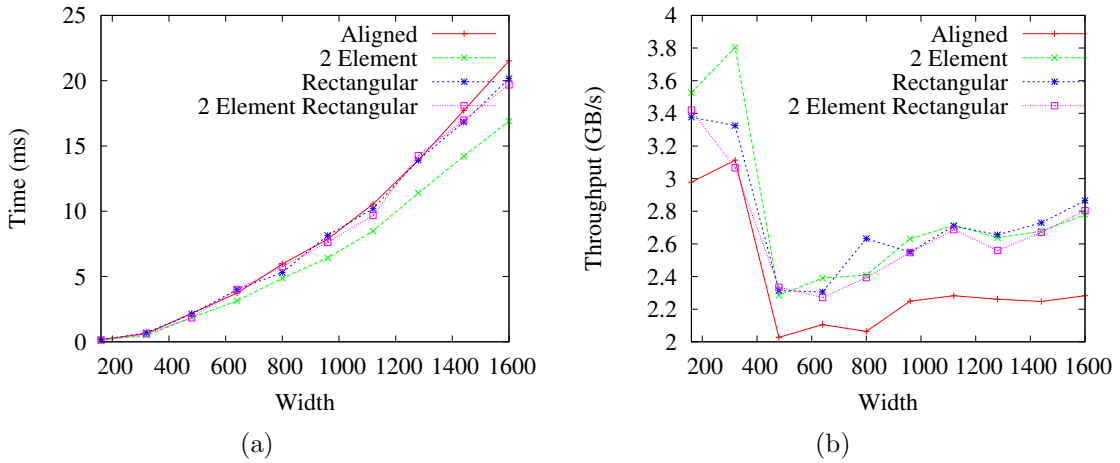
Figure 5.23: (a) shows total time while (b) shows application bandwidth for difference of Gaussian kernels.

two-element, aligned rectangular, and aligned rectangular two-element. Since each cell requires checking against all neighbors, a one cell halo is added to the data. In most previously demonstrated kernels, not of the two-element variety, each thread loads a single element into shared memory. For the extrema detection, however, each thread must load 3 elements: the element being checked for extremity and the elements "above" and "below" in the same scale space, as shown in Figure 2.3. Shared loading can be seen in Figure 5.24. Halo loading is not shown, but is similar to Figure 5.6, except that in the extrema kernel, each halo load is replaced by 3 halo loads.

The performance results for the various extrema detection kernels are somewhat surprising, and can be viewed in Figure 5.25. The most obvious discussion point about these results is that the two element rectangular kernel is not the fastest kernel, as it has consistently been in the previous experiments. To further investigate these findings, the CUDA Visual Profiler is useful. Table 5.4 contains selected fields

```
int r = blockIdx.y * blockDim.y + threadIdx.y;                          1
int c = blockIdx.x * blockDim.x + threadIdx.x;                          2
int idx = r * stride + c;                                               3
int x = threadIdx.x + 1;                                                4
int y = threadIdx.y + 1;                                                5
__shared__ float s[3][SHARED_WIDTH][SHARED_WIDTH];                      6
                                                                        7
s[0][y][x] = in[idx - h * stride];                                      8
s[1][y][x] = in[idx];                                                   9
s[2][y][x] = in[idx + h * stride];                                      10
```

Figure 5.24: Extrema shared memory staging, note the 3 dimensional array.



(a) (b)

Figure 5.25: (a) diagrams total time and (b) application bandwidth: extrema detection.

from the Visual Profiler output (many fields are not displayed because they display non-performance related information). The table contains one row that requires explanation: occupancy. Occupancy is the ratio of active warps to the maximum number of active warps. If the number of active warps is less than the maximum, it is usually due to some form of space constraint: a kernel uses too many registers per thread and therefore one warp takes more than half of the register file. An occupancy of less than 1 is not necessarily a terrible thing, however, all things being equal a kernel with a higher occupancy will usually perform better. The rows entitled *Register per*

Thread and *Shared Mem. Per Block* give an indication as to the occupancy value. Kernels that use lots of shared memory will have a smaller occupancy ratio.

Table 5.4 indicates that the occupancy for the 2 element rectangular kernel is low.

Table 5.4: Extrema detection: Visual Profiler results.

	Aligned	two-element	Rect.	two-element Rect.
Instructions	371,686	192,755	356,474	185,343
Coalesced Loads	54,000	54,000	71,964	69,120
Uncoalesced Loads	108,000	54,000	32,285	17,280
Branches (Divergent)	50218 (3754)	25751 (1951)	63,739 (2016)	24,785 (855)
Occupancy	1	0.5	0.75	0.25
Registers Per Thread	14	13	12	13
Shared Mem. Per Block	3,942	7,388	4,796	9,404

In fact, the occupancy value is significantly different from the next fastest kernel: the two-element kernel. The reason for this low occupancy is because the shared memory per block for the two-element rectangular kernel is so high. Only one block is able to fit on the SM at a given time. This means if all the threads are waiting for memory requests, there is no other block that can be executed, and the SM is essentially idle. Table 5.4 demonstrates that the two-element kernel has the best performance, and that is reflected in Figure 5.25. The table also clearly shows that the two-element rectangular kernel is being held back by its occupancy since all the other values are comparable to the two-element kernel. Note the use of primitives such as `blockDim.x`, `blockDim.y`, `threadIdx.x`, and `threadIdx.y`. These are built-in variables made available by the CUDA run-time system, and they utilize space in the register file which increases the number of registers in use by each thread. For many cases, block dimensions are known at compile time, enabling these primitives to be replaced by constant values. Doing so would reduce the register need of each thread, and possibly allow more threadblocks to fit on an SM, thus increasing occupancy. However, replacing the primitives with constants causes code that is harder to read

and maintain. The rectangular kernel executes approximately 150,000 more instructions than the two-element kernel slowing it down, a result consistent with previous experiments. The standard aligned kernel performs the worst due to the high number of uncoalesced memory loads, as this is to be expected with a one cell halo. The extrema kernels are interesting because they incorporate issues found in real world problems, such as available memory space. Sometimes, due to various issues the best techniques are not always feasible.

Chapter 6

Surface Water Flow: A Case Study

A cellular automaton model is proposed in [30], and further refined in [31], that models the movement of water over an area of land during and after a rain event. In [30] Parsons presents a Java program which implements this CA, resulting in a digital elevation map that indicates the depth of the water in each cell. Cells which represent areas in the bottom of valleys or riverbeds, for example, contain the most water, since they are, in effect, local minima. Intuitively, this makes sense, since water flows downhill. Computing exactly where the water will be, however, and estimating how much water there is becomes a much more complicated problem. The implementation by Parsons is intended to be clear and simple; no consideration is given to performance. The reason for conducting this case study is simple: apply the techniques and patterns described in Section 4 to an existing, real-world problem, to further determine their validity and usefulness. The intent of this section is to explain the process of implementing Parsons CA on a GPU and is organized thus: Section 6.1 explains the details of the environment and experiment setup. In Section 6.2 a brief overview is given, followed by an explanation of porting the supplied code to C++. Next, in Section 6.3 an initial GPU implementation is discussed, which leads to in-depth

analysis of the existing rules and implementation given in Section 6.4. Section 6.5 explains a second GPU implementation, completed after the analysis done in 6.4. Lastly, Section 6.6 contains a conclusion and sums up the lessons learned during the process.

6.1 Method / Setup

Due to technical issues related to driver versions, the machine used to run the code described in this chapter is different from the one on which the previous experiments were run. The CPU is a 64 bit Intel T9400 Core2 Duo processor running at 2.53GHz. The CPUs each have 6MB of on-chip cache and 2 hardware threads. A total of 3.5GB of memory is available. The GPU used in the surface water experiments is an Nvidia GeForce 770M, running CUDA version 2.20. The GPU has 4 streaming multiprocessors for a total of 32 computing cores, running at 1.25 GHz. The theoretical memory bandwidth limit of this card is 35.6 GB / sec and the card has 500MB of physical memory. The surface water case study requires more data to complete computations than either the Game or Life or SIFT kernels. Also, the surface water application, is a complete application and the performance timing is focused on execution from start to finish.

6.2 Overview of Existing Work

The main contribution of [30] is the development of a CA model; the Java implementation is secondary contribution that validates the first. Our works focuses on the geological theories presented in [30] only insofar as they help in understanding the implementation and ways to enhance performance. The Java implementation can be

broken down into 5 major steps:

1. Add rainwater to each cell.

2. Remove the necessary amount of "infiltration," that is, water absorbed by the soil.

3. Calculate how much water each cell "discharges:" inspect neighbors and determine which are at a lower elevation, and based on the elevation difference, determine what fraction of the discharge volume each neighbor will receive.

4. Iterate through all cells: for each cell, iterate through its neighbor list adding the appropriate amount of water to each neighbor. This step modifies the volume of a neighbor cell.

5. Iterate through all cells a second time, adding the aggregated, temporary amount of water to each cell, and subtracting the discharge volume.

Parson's Java implementation contains a two-dimensional array of cell objects and 5 main methods, one for each element in the above list. Each method loops over every cell in the two-dimensional array, applying the specific rules. Once all 5 methods have executed, a new amount of rain fall for the next time-step is calculated, and the process begins again. The implementation also includes a complicated set of timing rules that only "release" water from a cell after certain conditions are met. These conditions deal with water velocity and volume, only moving the water if there is enough critical mass, in effect. The timing rules do not change how much water moves from cell to cell, only when it moves. The initial port was done for two reasons: (1) to gain a better understanding of the Java code, and (2) to create a serial implementation whose performance could be compared to a GPU version, since comparing a GPU based program to a Java program is not very useful. The port is

implemented in C++, mimicking the Java class hierarchy and tested to ensure both versions produce the same results, within a certain tolerance.

6.3 Initial GPU Implementation

Implementation using C++ showed the aspects of the original code that would benefit from parallelization: namely the 5 steps listed in Section 6.2 that iterate over all the cells. These computations could, for the most part, be done in parallel. The first two steps, adding the rain water, and then removing the infiltration volume are trivially implemented in GPU kernels. The addition of the water can be seen in Figure 6.2. In fact, the add and remove kernels are so similar that they could be combined into a single kernel, or the amount of water to be removed could simply be subtracted from the rain fall, eliminating the need for a second kernel. It was decided that, for the purposes of generality, each cell could possibly contain a different soil type and therefore a different amount of water would be lost to infiltration. For this reason, the add and remove kernels remain separate. Note the index calculations in lines 4 - 6: they are the same as those presented in the Game of Life an SIFT kernels depicted in Section 5. The conditional statement in lines 7 and 8 is an artifact of discretizing a digital map. Since the boundary cells have no neighbors, no water would leave or enter these cells, essentially creating an artificial dam, therefore the boundary cells are not processed. An external viewer was also written to aid in verifying correctness of the implementations and its output can bee seen in Figure 6.1.

For steps (3), (4), and (5), the initial GPU kernels simply calculate the index of the cell to be processed and then replicate the implementation in the serial version; the only difference being the GPU kernels operate on only one cell. The goal of this implementation was to get the code running on the GPU as fast as possible, while

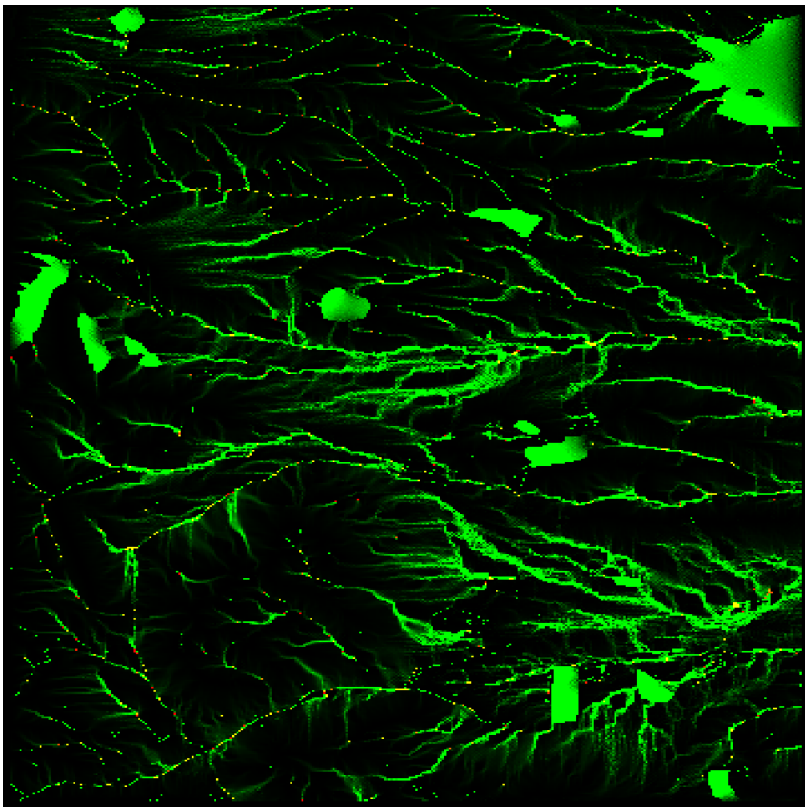


Figure 6.1: Viewing the result of the CA model

producing correct results. While this goal was achieved, in hindsight, this was a poor choice. In Parson's original implementation there is a single design decision that adds a significant amount of complexity: in steps (3) and (4), while processing a cell, the code modifies state values of other, neighboring cells. Strictly speaking, doing such does not fall under the cellular automata definition, and when our CA techniques and patterns are applied to such a situation, results are mixed, at best. Because one cell is modifying the value of another cell, the order in which operations are performed is important. It became very difficult to correctly implement these rules, and while a working implementation was finally created, the code was difficult to understand and unmaintainable, making it arduous to apply performance improvements.

Due to the neighbor based nature of the computations, the initial GPU imple-

```
__global__ void static addPrecipKernel(int rows, int cols,
                                       float tmpVol,  GpuTCell *data)
{
  int r = blockDim.y * blockIdx.y + threadIdx.y;
  int c = blockDim.x * blockIdx.x + threadIdx.x;
  int idx = r * cols + c;
  if (r != 0 && r != (rows - 1) && c != 0 && c != (cols - 1) &&
      r < rows && c < cols)
  {
    data[idx].vol += tmpVol;
    data[idx].newVolume = true;
  }
}
```

Figure 6.2: Kernel that adds rain water to each cell.

mentation used shared memory. Memory alignment and rectangular kernels were not implemented. Table 6.2 contains a comparison of total time between the initial GPU version and serial C++ version, the times are given in seconds. The experiment consists of running the different implementations against a digial elevation map of Golden, Colorado, for 500 generations. The map of Golden is furnished by the U.S. Department of the Interior and measures 463x358 cells. The initial results indicate that the GPU version is approximately 3.9 times faster than serial version. This speed up is gained from simply porting the C++ version to a GPU version, not many of the techniques described herein were used. Performance improvements such as this are artifacts of using a SIMD architecture and further discussed in Section 2.4.1. Table 6.2 also shows the performance of various steps in the process; the data is meant only as an indicator of performance trends. Since the serial and GPU implementations perform different work at the various steps, the most useful data point is the total time.

Table 6.1: Initial Surface Water Implementations

	Serial C++	GPU Shared
Total time	77.2817	20.3935
Add time	2.9500	0.6700
Remove time	26.6331	1.9606
Process time	47.6751	17.7561

6.4 Surface Water Flow: Revisited

Due to reasons given in Section 6.3, it was determined that the original implementation was not a true CA. Because of this, attempts to add performance improvements were either too difficult to implement, or yielded poor results. To address these issues, the original algorithms were simplified, and each step was constructed to fit a CA model: the state of cell is solely based on the current state and the state of its neighbors. Steps (3), (4), and (5) listed above, were condensed into two, logically simple, steps: calculate how much water a cell discharges, and calculate how much water is gained from neighbor cells:

1. Add rainwater to each cell.
2. Remove the necessary amount of "infiltration," that is, water absorbed by the soil.
3. Calculate how much water each cell loses, and subtract this volume.
4. Calculate how much water each cell gains, and add this volume.

The calculation of how much water a cell discharges remains largely the same as the initial implementation. However, determining the amount of incoming water is significantly different from the original model. The amount of incoming water is the sum of the water a cell receives from each of its neighbors. To determine the volume

of water received from a neighboring cell, all of the neighbors of that neighboring cell must be analyzed. Figure 6.3 depicts a visual representation of this calculation. The initial reason for separating out these steps was to reduce the number of cells that are accessed. However, on the GPU, this type of operation is typical and fast with the use of shared memory.

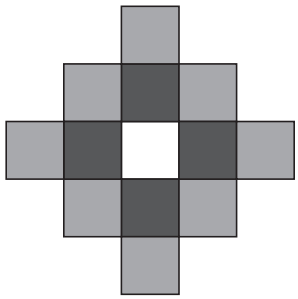


Figure 6.3: Illustration of all the cells that are accessed during the computation of incoming volume for the center cell. Lightly shaded cells respresent those which must be accessed in-order to compute the incoming water for the center cell.

6.5 Final GPU Implementation

Because the revised model is a standard CA, implementing the various performance improvements is a straightforward process that does not lead to error-prone or unmaintainable code. The final implementation includes many of the techniques described in Section 4. A complete list is presented here:

- Basic global memory

- Shared memory
- Aligned shared memory, with halo
- Aligned rectangular, with halo
- Two-Generation

A bar graph, depicting the total time for each method is shown in Figure 6.4. At a high-level, these results are expected: rectangular regions perform the best, followed by aligned memory, shared memory, and global memory. Since each of these techniques builds on the previous one, these results are expected. The two-generation kernel result is surprising, insofar as it is much worse than expected. From previous results, the two-generation kernel is expected to be slower due to a high instruction count, but 7x slower is significantly worse than seen in previous experiments. The largest performance increase is due to the introduction of shared memory: a 21% performance improvement over global memory. The other successive techniques, aligned and rectangular, garner .35 and .28 second improvements, respectively. The process kernel consumes the largest amount of time, as is expected, since this kernel implements the complicated CA rules and contains more arithmetic operations than the add and remove kernels, which both entail a single add or subtract instruction. The add and remove kernels benefit only from the addition of aligned memory as they do not complete any neighbor oriented tasks.

Table 6.2: Revised Surface Water Implementations

	Global	Shared	Aligned	Rect.	2 Gen.
Total time	5.5995	4.4027	4.0500	3.7669	25.6975
Add time	1.2948	1.1211	0.7374	0.7046	0.8410
Remove time	1.0488	1.0488	0.5991	0.5778	0.6723
Process time	3.2047	2.2308	2.7106	2.4820	24.1815

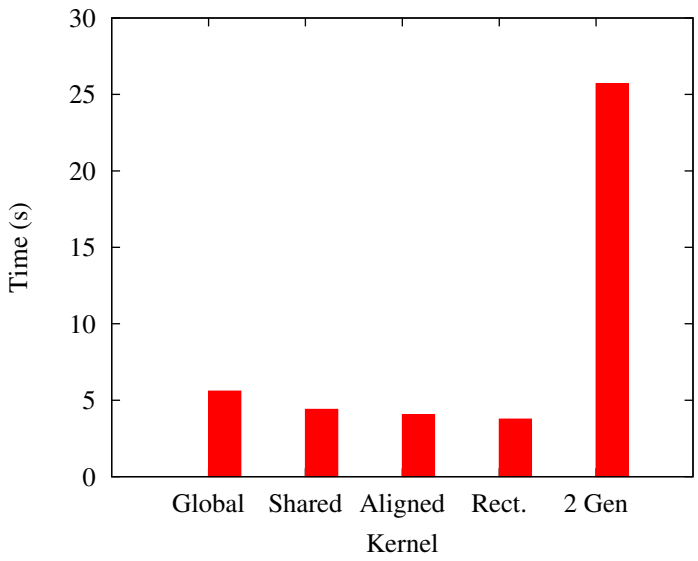


Figure 6.4: Surface Water Total Time

Further investigation is required to determine the cause of the two-generation performance. Table 6.3 contains output from the CUDA Visual Profiler. The most telling statistic for the two-generation performance problem is instruction count: the two-generation kernel executes over 3x as many instructions as the rectangular kernel. Code inspection reveals a considerable number of index calculations to be the cause of such a high instruction count. While this is a significant amount, it cannot account for all of the performance degradation. One may suspect that large number of branches contributes to the problem, however only a small fraction of the branches cause divergence. The occupancy of the two-generation kernel, 0.167, is exactly half of all the other kernels. Twice as many threadblocks for the rectangular kernel can fit on an SM. When a threadblock for the two-generation kernel is waiting on a memory request, the number of waiting and ready threadblocks that can be context switched is much lower, causing the SMs to be idle. The occupancy of the two-generation kernel is lower because it uses 54 registers per thread; registers are allocated in the same physical space as shared memory, which is very limited. The more registers a

thread uses, the fewer threads can be allocated to an SM. The reason the register count is so large stems from the extra index calculations. The lower occupancy and high instruction count are the reasons the two-generation kernel performs poorly.

Table 6.3: Comparison of Visual Profiler results for surface water kernels.

	Global	Shared	Aligned	Rect.	2 Gen.
Instructions	719,888	1,081,273	1,086,864	1,145,703	3,866,277
Coalesced Loads	9,166	2,792	53,440	55,680	69,368
Uncoalesced Loads	431,616	431,616	63,408	66,096	63,072
Branches	108,757	273,657	272,270	290,542	992,429
Divergent	11,209	22,924	23,093	27,812	37,903
Occupancy	0.333	0.333	0.333	0.333	0.167
Registers Per Thread	20	23	21	20	54
Shared Mem. Per Block	56	6,456	6,460	8,764	7,868

6.6 Surface Water Flow: Final Thoughts

The case study of a surface water flow problem provides some useful insights. Chief among these insights is that using a properly defined cellular automata model is important. If the next generation is dependent upon more than the current state and neighbor states, or if during the course of processing a single cell the state values of another cell are modified, complications arise which are difficult and tedious to handle. It was shown that a GPU implementation of poorly defined model could, indeed, achieve performance gains. However, a revised implementation built upon a strict CA model realizes a much larger improvement. Specifically, the initial GPU version is approximately 3.9x faster than the serial version while the revised implementation is approximately 30x faster.

Another insight concerns the multi-generational kernels. The multi-generational surface water kernel is similar to the Game of Life kernels in that less threads are

used to load and subsequently manipulate more values. In both situations, a high instruction count an low occupancy cause performance problems. Future research in the area of multi-generational kernels should attempt to use a paradigm where there is a thread for each required piece of data, but not all those threads are used for calculations.

Lastly, due to the large amount of data required to process each generation, techniques that use additional shared memory, such as the two-element approach, prove to be infeasible. However, several further improvements could still be made: packing the various input data into native vector types, such as `float2`, `float3`, and `float4`, would reduce the number of global memory transactions, while constant memory, described in Section 2.4.2, could be used for static input data, such as the elevation of each cell. The amount of shared memory required does not, however, affect the implementation of memory alignment, which achieves a substantial gain. The data shows that these techniques, do in fact, realize a notable performance improvement, even in the precedence of complicating factors.

Chapter 7

Discussion

This chapter gives an overview of the salient points that have been learned from the research presented herein. Section 7.1 enumerates each of the performance techniques while providing some comments about their implementation, feasibility, and overall usefulness. Section 7.2 discusses the multi-generation kernels and also indicates areas of future work. Next, Section 7.3 presents some of the compromises encountered while implementing various techniques. Finally Section 7.4 discusses more subjective topics such as ease of use and return-on-investment.

7.1 Improvement Overview

Due to the parallel nature of many-core processors such as GPUs, a first-pass parallel implementation of a serial, CPU-based application is likely to achieve acceptable results; in the case of our surface water application, a 4x speedup. Section 2.4.1 discusses this phenomenon in more detail. However, to fully realize the potential of GPGPU computing, additional techniques must be applied.

7.1.1 Shared Memory

The technique typically exhibiting the largest performance improvement over a baseline implementation is the use of shared memory. Section 4.2.1 details the workings and reasons for the performance improvement resulting in the use of shared memory. In the Game of Life environment, the introduction of shared memory results in a 30% speed increase. For applications that take weeks or months to run, a performance increase of this size is dramatic. In the surface water application, the introduction of shared memory results in a 21% speed increase. Since cellular automata are, by definition, neighbor based models, the employment of shared memory is critical. However, the surface water case study shows that an application may require more space than is available on the current hardware. In such cases, it may be possible to trade the use of one technique for another. For example, one could forgo the use of shared memory altogether, but instead, implement a rectangular two-element kernel, thereby significantly reducing instruction overhead. Future work should compare the effects of each technique individually.

7.1.2 Memory Alignment

The hardware implementation of global memory directly affects the performance of GPU kernels, mostly because this memory is not cached. Improper memory access patterns seriously inhibit performance and application throughput. Memory alignment is critical for minimizing memory latency. Section 4.2.2 explains the details of memory alignment. For Game of Life, memory alignment increases performance by approximately 8% over unaligned memory. While 8% is not quite as impressive as the 30% achieved by shared memory, it is still worth implementing. The surface water application also sees an 8% speed increase from using aligned memory. In the

case of surface water, the increase is application wide: all GPU kernels benefit. An ancillary benefit of memory alignment is that GPU kernels do not need to change in order to enjoy the benefit. Host CPU code is responsible for correct alignment. One downside of using aligned memory is that it may require some initial processing to set the alignment and add any required padding. Some software libraries, such as OpenCV [4], complete this step, but others may not.

7.1.3 Halos

Adding a memory halo to a set of data removes the need for conditional logic while processing cells that lie on the edges of the memory region, as these edge cells do not have certain neighbors. In a GPU implementation, the presence of a halo simplifies logic, by removing conditional instructions, and in doing so, reduces the total instruction count of a kernel significantly. Section 5.2.1 shows that using a halo increases speed by 18% for Game Of Life. It is also shown that this increase is due to the reduced number of instructions: the halo-based kernels actually make more requests to global memory. Adding a memory halo is trivial, but unlike memory alignment, the kernel must be implemented in such a way as to take advantage; that is, remove conditional logic for memory boundary cells.

7.1.4 Rectangular Memory regions

The use of a memory halo causes additional global memory requests to read the extra halo data. Not only is this extra data essentially unused, only a portion of these memory requests are aligned. To mitigate these consequences, it is possible to structure threadblocks in such a way that significantly reduce the amount of unused or wasted data that is transferred across the bus. As stated in Section 4.2.4 each left-side

and right-side halo load request incurs a throughput penalty, but the top and bottom halo cells do not. The results in Section 5.2.2 indicate that structuring threadblocks that are much wider than they are high, increases performance by 16% for The Game of Life. Section 6.5 demonstrates that the use of a rectangular threadblock increases performance by approximately 7%. Using a rectangular threadblock is not difficult; in fact, it is possible to implement kernels in such a way that the shape of the threadblock does not matter, essentially realizing this performance improvement for free.

7.1.5 Two Elements Per Thread

It has been stated in Section 5.2.2 and again in Section 6.5 that index calculations constitute a large portion of the overhead for a GPU kernel. Once an index has been calculated, deriving more indices from it is relatively inexpensive. Processing two elements in one thread re-uses the initial index calculation to obtain a new index, relative to the first. The end result is a substantial reduction in instructions. For the Game of Life, introduction of this technique resulted in a 44% improvement over kernels processing only one element per thread. The convolution kernel achieved a 37% performance improvement from processing two-elements per thread. Clearly this technique is powerful, but it does not come without cost. Twice as much data is processed which means twice as much memory is used. Because shared memory is limited, it may not be possible to implement a two-element kernel without an unacceptable decrease in occupancy. See Section 7.3 for more discussion on this point.

7.1.6 Data Packing and Interleaving

Section 7.1.2 gives an overview of why memory alignment is important. Many kernels operate on floating point data, a 4-byte type, which results in coalesced memory transactions of 64 bytes for each half-warp. However the current hardware can support 128-byte memory transactions. Packing the data into native 8-byte types can further reduce the number of memory transactions by half. In the CUDA environment this is done by using the `float2` native type. Initial results from Section 5.3.2 indicate this to be a promising technique, however, more investigation is required. The surface water application would benefit from this technique since multiple sources of data are needed: elevation, volume, etc. The data elements could be packed into the native vector types, such as `float2`, to reduce the number of total memory accesses.

7.2 Multi-Generational Kernels

In theory, computing multiple CA generations per kernel increases the arithmetic intensity and improves throughput. However, in practice, no substantial performance improvement is made. Investigation in Sections 5.2.3 and 6.5 indicate that a substantial amount of overhead is required when computing multiple generations. This overhead manifests itself in the form index calculations. The multiple generation kernels implemented herein use a paradigm of one thread for each effective cell; this results in a situation where many threads not only load more than one element, but they also process multiple elements during the computation of the first generation. This paradigm was chosen because, initially, it was thought that keeping all the threads busy would result in better performance than a situation where many threads were idle. The results show that using fewer threads to load and processes more data incurs too much overhead to be effective. Future work in this area should investigate

the alternative solution where each thread loads a value (including the halo values), but only those threads representing effective data actually execute CA computations.

7.3 Compromises

As is the case with every software project, certain decisions must be made to accommodate application specific parameters. In the surface water case study, a large quantity of data is required to compute each generation, thus significantly reducing the occupancy of the kernels. In this case, the impact of a reduced occupancy is lower memory throughput. However, for another application that is bound by instructions, a lower occupancy may have little effect. Many such situations can arise, and a clear understanding of the problem at hand can help in deciding which techniques to apply and which to forgo. For example, implementing both shared memory in conjunction with a two-element kernel may not be possible due to the limited amount of shared memory. For a memory intensive application, the performance gained from repeatedly accessing values in shared memory far outweighs the reduction of instructions a two element kernel achieves. However, the opposite is true for an application that requires little in the way of memory bandwidth. The results of this work do not indicate that any one technique is more valuable than another, but instead, illuminate situations where each technique realizes the largest benefit.

7.4 Observations and Intangible Results

During the course of the research presented in this work, many observations were made not directly related to the performance data. Results clearly show that GPGPU computing has the capacity to significantly improve performance for many cellular

automata applications. However, for these improvements to be of use to the research community as a whole, they must be relatively easy to attain; that is, the introduction of GPGPU computing should not make the lives of researchers harder. As such, the bulk of the techniques presented here are not difficult to implement and should be attainable by others. Multi-generation kernels may be an exception, however. Implementation of multi-generational kernels is tedious, error-prone, and time consuming. Even if a multi-generational solution produced better improvements, the work required to build, debug, and verify such a kernel may far outweigh the performance benefit.

This work presents techniques and patterns that are specifically applicable to problems modeled by cellular automata theory. The main benefit of using a CA model over more complicated mathematical models is simplicity. An important observation made while completing this research is that the simplicity achieved via the use of CA models is real. Therefore, it is our belief that CA theory should be applied to more areas and problems, at least initially, because model implementation and simulation results can be achieved quickly. However, it is important to maintain strict adherence to CA rules and definitions, lest the simplifications inherent in CA models be lost.

7.5 Conclusion

This work presents a series of performance enhancements that apply to implementations of cellular automata models using GPGPUs. These improvements include: the use of shared memory, the addition of memory halos, aligning data to maximize memory coalescence, processing multiple elements per thread, and modifying the shape of memory regions to improve bus utilization. Performance improvements are implemented in the contexts of Game of Life, convolution operations, difference of Gaussian

computations, local extrema detection, and finally a surface water flow model case study. Experimental results are shown, comparing the improvements made by each technique while giving a detailed analysis of their applications. An in-depth look at a real world problem is presented as a case study, highlighting many of the proposed techniques while discussing application specific details. Lastly, an overview of the techniques is given, emphasizing the importance of each. Future work is also described, indicating further areas of research to be conducted. The results indicate the improvements described herein are not only beneficial, but also applicable to a wide range of areas, making the subject of cellular automata models and GPGPU computing a worthwhile endeavor.

Bibliography

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45:56–61, November 2002.

[2] D. Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25:724–734, July 2006.

[3] D. Blythe. The direct3d 10 system. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 724–734, New York, NY, USA, 2006. ACM.

[4] G. Bradski and A. Kaehler. *Learning openCV: computer vision with the openCV library; electronic version.* O'Reilly, Sebastopol, CA, 2008.

[5] D. R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[6] J. Conway. The game of life. *Scientific American*, 1970.

[7] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science and Engineering*, 5:46–55, 1998.

[8] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In M. D. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 131–140. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-28642-4-2.

[9] B. Dreier, M. Zahn, and T. Ungerer. Parallel and distributed programming with pthreads and rthreads. In *High-Level Parallel Programming Models and Supportive Environments, 1998. Proceedings. Third International Workshop on*, pages 34 –40, Mar. 1998.

[10] B. G. Ermentrout and L. Edelstein-Keshet. Cellular Automata Approaches to Biological Modeling. *Journal of Theoretical Biology*, 160(1):97–133, January 1993.

[11] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[12] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21:513–520, 2006. 10.1007/s11390-006-0513-y.

[13] J. Fung and S. Mann. Openvidia: parallel gpu computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia*, MULTIMEDIA '05, pages 849–852, New York, NY, USA, 2005. ACM.

[14] H. L. Garner. The residue number system. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, IRE-AIEE-ACM '59 (Western), pages 146–153, New York, NY, USA, 1959. ACM.

[15] J. Gómez and G. Cantor. A population scheme using cellular automata, cambrian explosions and massive extinctions. In *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 1849–1850, New York, NY, USA, 2009. ACM.

[16] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 61–72, New York, NY, USA, 1994. ACM.

[17] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. *High Performance Computing HiPC 2007*, 4873(LNCS):197–208, 2007.

[18] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[19] G. Koch. Discovering multi-core: Extending the benefits of moore's law. *Technology*, 2005.

[20] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau. Real-time eye blink detection with gpu-based sift tracking. In *Computer and Robot Vision, 2007. CRV '07. Fourth Canadian Conference on*, pages 481 –487, May 2007.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 –55, march-april 2008.

[22] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance comparison of mpi implementations over infiniband, myrinet and quadrics. In *IN PROCEEDINGS OF INT'L CONFERENCE ON SUPERCOMPUTING, (SC'03*, 2003.

[23] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004. 10.1023/B:VISI.0000029664.99615.94.

[24] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[25] S. Nandi, B. K. Kar, and P. P. Chaudhuri. Theory and applications of cellular automata in cryptography. *IEEE Trans. Comput.*, 43:1346–1357, December 1994.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008.

[27] Nvidia. nvidia cuda c programming guide, version 3.1, 2010.

[28] Nvidia. nvidia cuda visual profiler, version 3.1, 2010.

[29] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃ$\frac{1}{4}$ger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[30] J. A. Parsons. A COMPUTATIONAL CELLULAR AUTOMATON FOR MODELING SURFACE WATER FLOW IN THE ROCKY MOUNTAIN NATIONAL PARK AND THE WALNUT GULCH EXPERIMENTAL WATERSHED. *unkown*, 2004.

[31] J. A. Parsons and M. A. Fonstad. A cellular automata model of surface water flow. *Hydrological Processes*, 9999(9999):n/a+, 2007.

[32] J. Preston, K., M. Duff, S. Levialdi, P. Norgren, and J. Toriwaki. Basics of cellular logic with some applications in medical image processing. *Proceedings of the IEEE*, 67(5):826 – 856, may 1979.

[33] C. Reynolds. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, pages 113–121, New York, NY, USA, 2006. ACM.

[34] G. M. A. M. M. J. Rutherford and K. P. Valavanis, 2011.

[35] S. Rybacki, J. Himmelspach, and A. Uhrmacher. Experiments with single core, multi-core, and gpu based computation of cellular automata. In *Advances in System Simulation, 2009. SIMUL '09. First International Conference on*, pages 62 –67, september 2009.

[36] R. R. Schaller. Moore's law: past, present, and future. *IEEE Spectr.*, 34:52–59, June 1997.

[37] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[38] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[39] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 22:207–217, 2011. 10.1007/s00138-007-0105-z.

[40] R. Szerwinski and T. GÃ$\frac{1}{4}$neysu. Exploiting the power of gpus for asymmetric cryptography. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and*

Embedded Systems - CHES 2008, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85053-3-6.

[41] A. C. Thompson, C. J. Fluke, D. G. Barnes, and B. R. Barsdell. Teraflop per second gravitational lensing ray-shooting using graphics processing units. *New Astronomy*, 15(1):16 – 23, 2010.

[42] T. Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D: Nonlinear Phenomena*, 10(1-2):117 – 127, 1984.

[43] G. Y. Vichniac. Simulating physics with cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):96 – 116, 1984.

[44] J. von Neumann. The general and logical theory of automata. In *Cerebral Mechanisms in Behavior*, pages 1–41. pub-WILEY, pub-WILEY:adr, 1941.

[45] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657 – 673, 1994. Message Passing Interfaces.

[46] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, January 1962.

[47] Wikipedia. Floating point operations per second, Feb. 2011.

[48] S. Wolfram. *A New Kind of Science*. Wolfram Media, May 2002.

[49] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.