UNIVERSITY OF CALIFORNIA, SAN DIEGO

GPU Accelerated Cardiac Electrophysiology

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Fred Lionetti

Committee in charge:

Professor Scott Baden, Chair
Professor Andrew McCulloch
Professor Dean Tullsen

2010

The thesis of Fred Lionetti is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____
Chair

University of California, San Diego

2010

DEDICATION

To Christina, whose love and support made this possible.

EPIGRAPH

In a few minutes a computer can make a mistake so great that it would have taken many men many months to equal it. –Anonymous

With the GPU, this may be possible in just a few seconds. –Fred

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

We would like to thank Stuart Campbell (UCSD Bioengineering), Dr. Chris Anderson (UCLA Mathematics Dept), and Dr. Xing Cai (Simula Research Laboratory) for the discussions on ODE Solvers and Dr. Jazmin Aguado-Sierra (UCSD Bioengineering) for supplying the sample electrophysiology model. We also would like acknowledge the National Biomedical Computational Resource (NIH grant P41RR08605).

ABSTRACT OF THE THESIS

GPU Accelerated Cardiac Electrophysiology

by

Fred Lionetti

Master of Science in Computer Science

University of California, San Diego, 2010

Professor Scott Baden, Chair

Numerical simulations of cellular membranes are useful for both basic science and increasingly for clinical diagnostic and therapeutic applications. A common bottleneck in such simulations arises from solving large highly complex stiff systems of ordinary differential equations (ODEs) thousands of times for numerous collocation points (representing cells) throughout a three-dimensional volume. For some electrophysiology simulations, over 98% of the time is spent solving these systems of ODEs when run in serial on a single core.

We have reduced the time to simulate a single heartbeat from 4.5 hours on a 48 core Opteron cluster (MPI implementation) to 12.7 minutes on a Desktop workstation equipped with a $500 GPU accelerator that also economizes power consumption. This improvement over cluster performance transforms the simulation workflow, and at this level of performance we can realize larger scale simulations, previously only feasible on a cluster, that are needed in a clinical setting. To achieve this same performance on our Opteron cluster would theoretically require at least a 1020 core system. Thus, the GPU has effectively miniaturized the hardware requirement to perform the simulation.

We also demonstrate 23x to 280x speedups across a wide spectrum of cardiac cell models running on the nVidia GTX-295 GPU compared with a multithread implementation on a 4-core Intel i7 processor. Our simulator employs a source-to-source translator that converts a higher level python description of a cell model into highly tuned and optimized CUDA source code, which is then compiled with the CUDA C compiler for execution. Our optimizations include automatic kernel partitioning and a software managed-memory cache. Our translator also removes numerical singularities introduced by using single precision arithmetic.

Chapter 1

Introduction

1.1 Overview

Mathematical models describing cellular membranes form the basis of whole tissue models to describe the electrical activity of entire organs, such as the heart. Researchers use these models to create numerical simulations for basic science and increasingly for clinical diagnostic and therapeutic applications. A common bottleneck arises when solving large highly complex stiff systems of ordinary differential equations (ODEs) thousands of times for numerous collocation points (representing cells) throughout a three-dimensional tissue volume. This bottleneck can easily account for 98% of the total running of the simulation when performed as a serial, single core execution.

One measure of performance in cardiac modeling is the amount of time needed to simulate a single heartbeat. We demonstrate an implementation of our ODE solver on an nVidia GTX-295 Graphic Processing Unit (GPU) that significantly reduces the computational bottleneck and reduced the time to simulate a single heartbeat from 4.5 hours on a 48 core Opteron cluster (MPI implementation) to 12.7 minutes on a Desktop workstation equipped with a $500 GPU accelerator that also economizes power consumption. This improvement over cluster performance transforms the simulation workflow, making such simulation feasible in a clinical setting. To achieve this same performance on our Opteron cluster would theoretically require at least a 1020 core system. Thus, the GPU has effectively

miniaturized the hardware requirement to perform the simulation.

When compared with a single CPU, our GPU implementation delivers a 23x-280x speedup over an OpenMP implementation on a quad-core i7 processor.

In addition to demonstrating the efficient use of GPU hardware we also show how to achieve high performance even when faced with highly complex numerical kernels which are characteristic of our problem domain. We leverage our experience to provide insights into performance optimization.

nVidia, one of the principal GPU developers, releases a complete software toolkit for developing GPU accelerated programs called *CUDA*, an acronym for Compute Unified Device Architecture. Our simulator employs a source-to-source translator that converts a higher level python description of a cell model into highly tuned and optimized CUDA source, which is then compiled with the CUDA C compiler for execution. We will discuss the translator and the optimizations it implements. All optimizations are controlled at a high level, and the user need not modify the emitted CUDA source code. Our front-end (e.g. cell model parser) could be used to target other platforms, such as ATI, OpenCL, etc. though of course, the back-end optimizations are platform dependent. The domain scientist remains unaware of the details.

A model we have used is highly complex, and contains 87 ODEs referenced within 1800 lines of CUDA source code. The CUDA compiler is challenged to manage register usage. Our optimizer splits a model into separate CUDA kernels to reduce register demand, which the CUDA compiler can then handle on its own.

A weakness of the current nVidia architecture (the GTX 200 series) is the lack of a global memory cache, which could mitigate the high latency penalty of global memory, which is two orders of magnitude higher than that of the shared memory or registers. While global device memory can be copied to registers, sufficiently complex kernels will cause the compiler to spill data back to device memory.

The nVidia GPU also has a small shared memory which is nearly as fast as the registers. However, it is a scarce resource to be divided between all threads running on the GPU. For sufficiently complex kernels, such as the Flaim model with

87 scalar state variables, it is infeasible to manage shared memory by hand. We have implemented a software managed global memory cache for complex kernels that avoids the need to manually manage cached scalars. Our algorithm is based on Belady's classic optimum MIN algorithm for page replacement and runs off line at translation time; there are virtually no branches in our code, which is characteristic of cellular models.

Another constraint of the GTX 200 series architecture is that the double precision arithmetic rate is significantly lower than that of the single precision, thus providing an incentive to use single precision. However, if unchecked, single precision can introduce unacceptable round-off errors resulting in numerical singularities. In order to tolerate the errors committed in single precision, we implemented an automatic transformation that traverses the cell model and employs a root finder to eliminate division by zero. This strategy avoids the most critical round-off errors introduced by single precision and enables us to benefit from the higher single precision computational rate of the nVidia GPU. To facilitate this transformation, as well as our optimization strategies, we built a code generator. This code generator performs automatic symbolic optimization starting with Python code and generates CUDA compliant code.

1.2 Contribution

We have developed a system by which high level cellular models can be authored by a domain scientist and automatically translated into an optimized GPU kernel which incorporates expert GPU knowledge. In doing so, we have suggested a scheme for solving stiff systems of ordinary differential equations which yields excellent performance and reasonable accuracy on the GPU. We have also demonstrated a technique to remove numerical singularities common to cellular models which allows us to use single precision arithmetic for improved performance. We have also contributed a variety of techniques for optimizing large, highly complex CUDA kernels. These include an offline automatic caching scheme based on Belady's MIN page swapping algorithm and automatic kernel partitioning scheme to

reduce register pressure. Finally, our system is able to perform electrophysiology simulations much faster than traditional cluster implementations and is indeed fast enough that it would be feasible to use in clinical setting for patient specific modeling.

1.3 GPU Computing

Graphical Processing Units or GPUs are specialized processors designed to reduce the workload on the Central Processing Units (CPU) when computing graphic or video intensive tasks. Over the last two decades, GPUs have become an integral part of a computing platforms for video games, interactive simulations, and high-end 3D rendering.

GPUs can be thought of as co-processors or accelerators. They are not designed to replace or obviate the need for a CPU. In particular, CPUs are designed to run serial applications as quickly as possible. Although in recent years CPUs have become "multicore" and are able to achieve some degree of multithreaded parallelism, they are generally optimized for running serial code, the common case. GPUs, however, are designed for massively threaded parallelism rather than for serial computation which can be run on the CPU. Thus, an application developer can employ a heterogeneous execution model to implement massively parallel parts of an application on the GPU and the serial parts on the CPU.

Over the last few years, a tremendous interest has developed in using these accelerators for more general purpose computing rather than just video and graphic acceleration [40]. This trend began with shader languages but has recently evolved into a complete set of development tools released by the major GPU chipset developers to facilitate this type of general purpose GPU computing.

The three major vendors of GPUs are nVidia, AMD (formerly ATI) and Intel. While Intel produces low-end graphics processors integrated into various mother board chipsets, AMD and nVidia compete for the high-end GPU market[1].

[1]Intel has also announced that they will launch a high-end graphics processor separate from its integrated chipsets in the future, although its first attempt, codenamed *Larrabee*, was cancelled in December of 2009.

nVidia and AMD have alternative proprietary GPU platforms each of which is compatible only with their own hardware. Specifically, nVidia's distributes the CUDA Toolkit while AMD distributes the ATI Stream SDK. A vendor independent standard, OpenCL[10], is also being developed to allow HPC applications to be developed independent of hardware. We focus our GPU computing efforts on nVidia GPUs running CUDA, although our approach could be ported to the AMD's FireStream by generating code for the *Brook+* language rather than CUDA and by modifying our specific optimizations to match AMD hardware.

1.3.1 GPU Processing Resources

The nVidia GTX 200 series GPUs consist of several *Streaming Processing Clusters* (SPC). Each SPC contains 3 *streaming multiprocessors* (SM) each of which has 8 *streaming processors* (also referred to as *cores* by nVidia) which share access to local memory. Each core contains a fused multiply-adder capable of single precision arithmetic. A core is capable of completing 3 floating point operations per cycle - a fused MADD and a MUL (see Figure 1.1).

For our work, we use the nVidia GTX 295 card. This card contains two GPUs, each with 10 clusters and thus has a peak performance of 720 floating point operations per cycle per GPU (3 flops per core per cycle $\times$ 8 cores per SM $\times$ 3 SM per cluster $\times$ 10 clusters). With a clock speed of 1.242 GHz, the GPU has a peak GFLOP rate of 894.24 per GPU. And with two GPUs, the GTX 295 can achieve a 1788.48 GFLOPs for the entire card.

In addition to single precision operations, each SM contains one 64-bit fused multiple adder for double precision operations. Thus, a SM contains 8 single precision streaming processors for every double precision unit. SMs also contain 16KB of shared memory and 16KB of registers.

1.3.2 GPU Memory Hierarchy

nVidia GPUs have several memory spaces available each with its own benefits and limitations (see Figure 1.2). Effectively understanding and appropriate

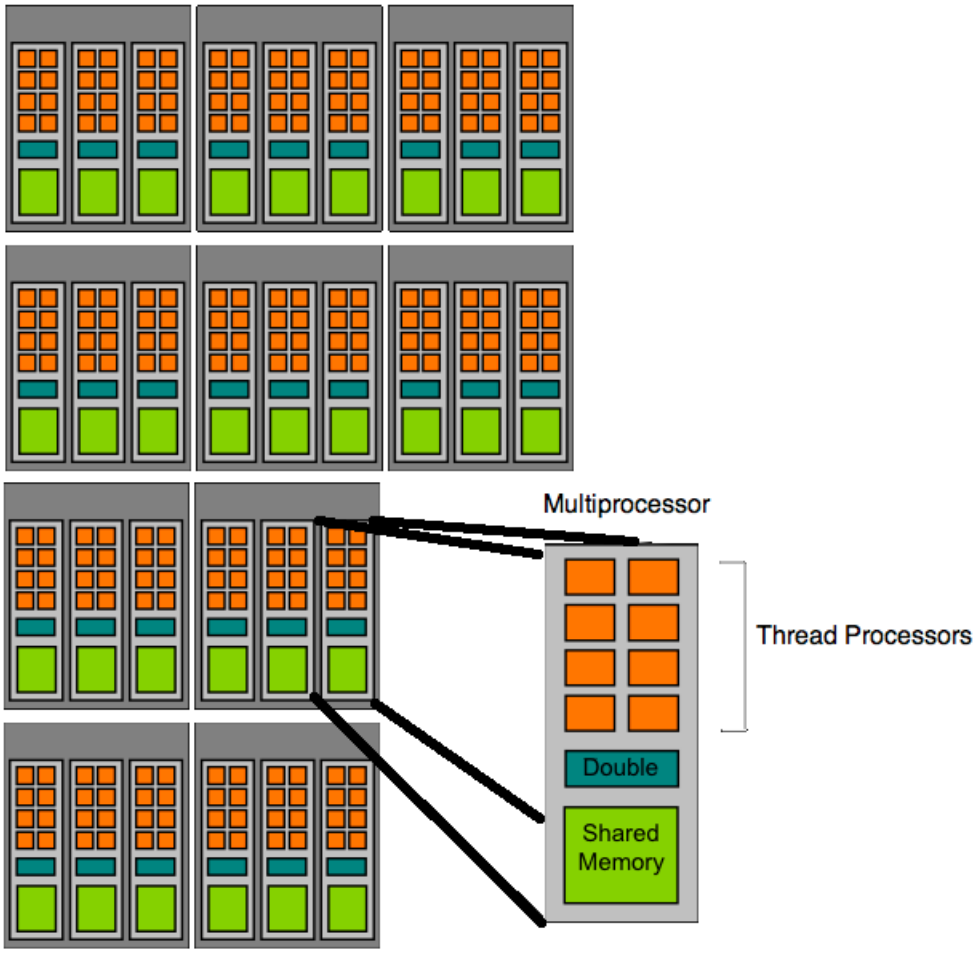

Figure 1.1: Diagram of 30 multiprocessors present in nVidia's 200 series GPU. Adapted from diagram on page 8 of [38]

.

use of these memory spaces is essential for achieving acceptable performance on the GPU.

The fastest available memory for GPU computation is *device registers*. Each streaming multiprocessor contains 16KB of registers. These registers are divided among all threads which reside simultaneously on the GPU. Thus, if CUDA kernels uses a high number of registers, the device will be unable to execute as many threads concurrently.

Each multiprocessor also has a 16KB region of *shared memory* space, which performs almost as fast as registers. Shared memory is primarily intended as a means to provide fast communication between threads, although, due to its speed, it can also be used as a programmer controlled memory cache [50].

Next GPUs have DRAM (Dynamic Random Access Memory), or *device memory*, which is available at approximately a 150x latency when compared to registers or shared memory. This device memory is logically partitioned into four regions: *global memory*, *local memory*, *texture memory*, and *constant memory*. Global memory is available to all threads and is persistent between GPU calls. Local memory is available only to individual threads and is used as a backup when the compiler is unable to fit requested data into the device's registers, in which case registers are said to *spill* to local memory. Texture memory is read-only with a small cache optimized for manipulation of textures. Constant memory, as the name implies, is also a read-only region which also has a small cache.

Finally, *host memory* (the system's main memory) is available indirectly and relatively slowly to the GPU. This memory space is only available to the GPU when copied over the PCI-Express bus to the GPU's device memory.

Although the nVidia's 200 series is capable of both single and double precision arithmetic, there are eight single precision arithmetic execution units for each double precision unit. Thus, a computational bound algorithm would be expected to execute at 1/8th the speed when using double precision. Some algorithms, though, are *memory bound*, and because they spend most of the time requesting memory rather than performing computations, might run at only a 2x penalty when double precision is required as accessing twice the memory (for

double precision data) takes twice as long.

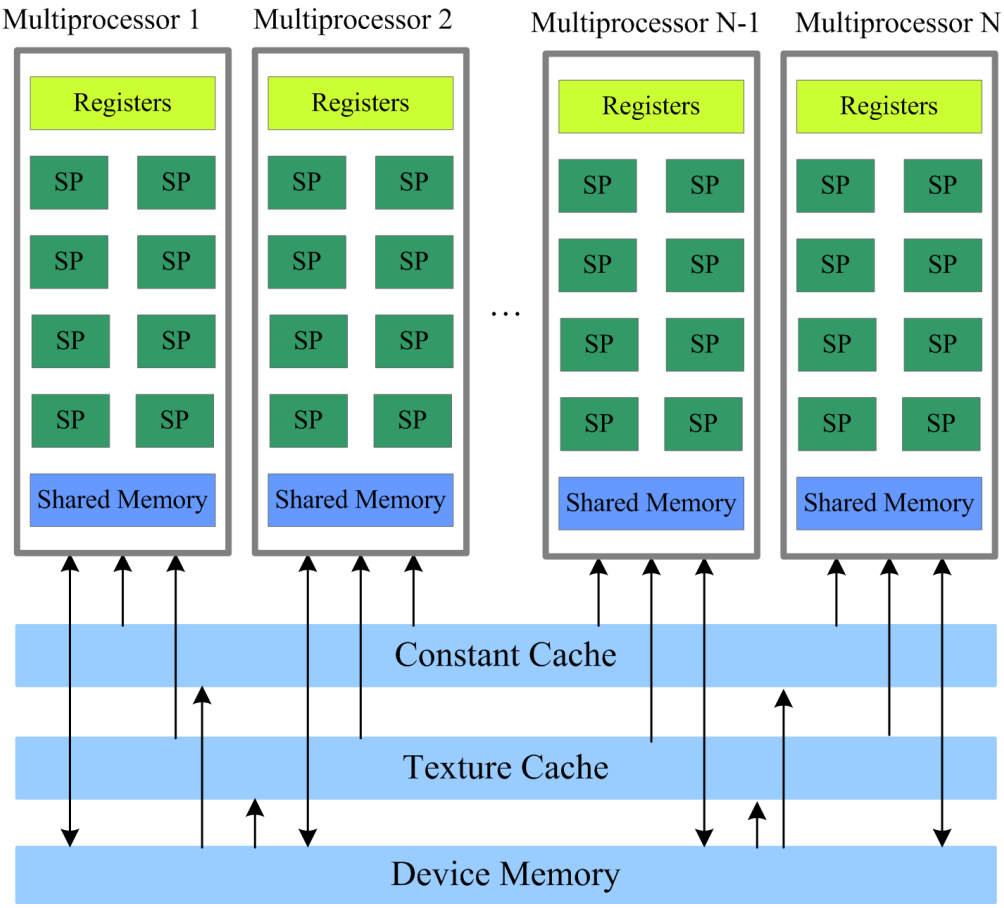

Figure 1.2: GPU Architecture. This diagram is file 3 of the supplementary data of [31].

1.3.3 CUDA

nVidia releases a complete software development toolkit for developing software for their GPUs. This tookit is freely available[2] and includes a GPU compiler (*nvcc*) as well as examples, and a kernel profiler. nVidia refers to this software development suite as *CUDA*, an acronym for Compute Unified Device Architecture. CUDA is only compatible with nVidia devices, although it is supported by all GPUs nVidia has sold for the last several years, starting with the 8000 series introduced in November of 2006.

[2] http://www.nvidia.com/object/cuda_get.html

The CUDA platform supports an extension to the C programming language with only a few additions and limitations, thus many experienced software developers are likely to be comfortable with the syntax. In order to control hardware specific to the GPU, some additional language constructs are added. For example, the `__shared__` keywords designates memory declarations to be in the GPU's shared memory. Some limitations exist, though, such as a lack of function pointers and lack of recursion, since all functions are inlined by the compiler automatically.

1.3.4 CUDA Programming Model

CUDA uses a Single Instruction Multiple Thread (SIMT) programming model, very similar to the Single Instruction Multiple Data (SIMD) model common in vector processors. Each thread running on the GPU executes the same instruction at the same time, but while operating on different data. However, unlike traditional SIMD architectures, CUDA will allow threads to diverge at a performance cost. When branches occur in the code (e.g. due to *if* statements) the divergent threads will become inactive until the conforming threads complete their separate execution. When execution merges, the threads can continue to operate in parallel. In order to take full advantage of the GPU hardware, the programmer must therefore write code in a way that minimizes thread divergence.

Thus, at the center of CUDA's programming model is the idea that there is an abundance of data which must be operated on in a uniform way by some instructions. The model requires a host (or CPU) and one or more devices (GPUs) each of which has a plethora of arithmetic execution units to perform many calculations in parallel - up to 240 units in the GTX 200 series. This model lends itself well to many applications such as image processing, physics simulations, etc. which are characterized by very large regular arrays of data which require regular processing. Each element of data can be operated on by its own very lightweight GPU thread of which thousands can execute on the device at a time.

In order to hide the latency of communicating with device memory, many threads are multiplexed over each core. When one thread requests device memory, another thread is made active until the memory request completes. From the

programmer's perspective this is hidden by virtualization - the programmer need not be concerned with which threads are running on which GPU cores. This idea of latency hiding is formalized by Little's law [30]. As applied to the context of latency hiding, Little's Law indicates that the time to hide is equal to the number of threads times the latency. In other words, as additional threads are added and multiplexed over the same hardware resources, greater latency can be hidden.

Both the GPU and CPU code exist in a single source in which GPU functions (called kernels) are identified with the `__global__` keyword. The CPU executes host code until a GPU kernel is called (see Figure 1.3). At this point, the GPU launches many - potentially thousands - of parallel threads and returns control asynchronously to the CPU. If another GPU call is made which executes another kernel or if a GPU memory transfer is initiated, the GPU will wait until the first call completes. Current GPUs, therefore, cannot execute more than one kernel at a time.

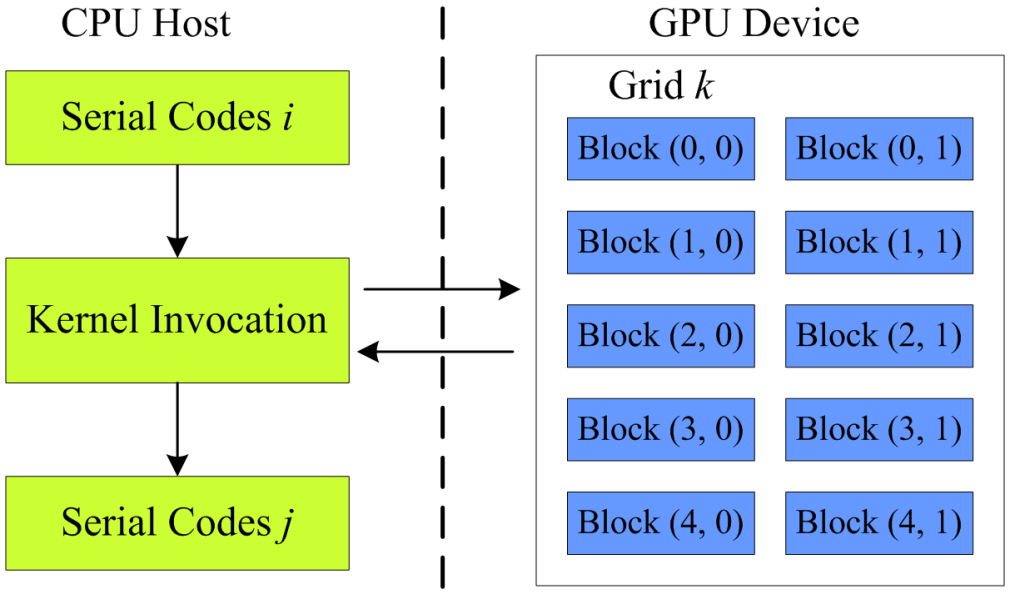

Figure 1.3: (test) Kernel execution. This diagram is file 2 of of the supplementary data of [31].

Each kernel is launched with a collection of threads, referred to as a *grid*. Grids can be indexed in 1 or 2 dimensions which can be convenient for processing a 2D image. Grids are further subdivided into *blocks* of threads, which can be

which can be indexed in 1, 2, or 3 dimensions. A block can have a maximum of 512 threads, although the programmer can decide, at run time, the exact layout of the grid and blocks to facilitate appropriate memory access. Each thread has its own *threadID* so that each thread can operate on its own data and to control its own logical program flow. CUDA provides mechanisms for the host to copy data from the CPUs DRAM to the devices DRAM, through cudaMemCpy(). The host can also allocate and free GPU memory with cudaMalloc() and cudaFree() function calls as well as manage CUDA *streams* which allow memory transfers and kernel execution to overlap.

CUDA also has a notion of thread *warps* which are collections of 32 threads which all attempt to execute the same instruction. It is at the warp level that global memory access and program flow must be carefully considered for maximum performance.

1.3.5 Device Occupancy

We define *device occupancy* as the maximum number of thread blocks that can multiplexed to run concurrently on a multiprocessor. This is influenced by the number of threads per block and the resources (registers and shared memory) required by each thread. It is desirably to have a large number of thread blocks to improve latency hiding by allowing the GPU to schedule other thread blocks while waiting for memory to become available.

nVidia GPUs are a parallel computing architecture and achieve peak performance when making the most use of available computational resources. Blocks of device threads are automatically scheduled by the GPU, which will simultaneously run as many blocks as the available hardware resources will allow. There is typically a trade off between device occupancy and device resources. For example, the more shared memory that a kernel requires, the fewer the threads that will be able to execute on a device. However, if a kernel relies excessively on global memory instead of shared memory and device registers it will likely be "memory bound" and spend time waiting for the availability of data rather than actually performing computations. This trade-off can be explored with the CUDA occupancy calcula-

tor[3] although it is often necessary to experiment with different configurations to find the optimal settings.

1.3.6 Strengths of GPU Platforms

Performance. Perhaps the most obvious benefit of GPU computing is the performance available at low cost. While the top of the line Intel CPUs come equipped with 4 cores, nVidia's GTX 200 series chipset can have up to 240 cores. nVidia's state of the art desktop card, the GTX 295, includes two GTX 200 GPUs on a single card, and thus provides 480 CUDA cores and can achieve a peak GFLOP rate of 1788.48 for single precision. The fastest desktop CPUs have a theoretical peak rate of around 70 GFLOPs. Many real world applications have seen one or two orders of magnitude speedup over optimized CPU implementations[9].

Inexpensive and Convenient. When compared to conventional supercomputing platforms, such as clusters, GPU computing has several significant advantages. Our complete desktop system with a GTX 295 was purchased for just under $2,500.00 - the GPU alone cost $500. We estimate that it would cost at least $100,000 ($1500 per 2-CPU node $\times$ 67 nodes) to build a cluster of with i7 CPU nodes to perform calculations at the same rate, very conservatively assuming that most communication costs could be hidden. Further, a GPU desktop does not require a specialized server room with additional energy and maintenance costs. Simulations must be highly efficient in clinical applications because of the time constraints involved in diagnosis and treatment, and because the simulations typically must be run many times in parameter sweeps to be useful. There is also a greater case for dedicated desktop computing in the clinical setting for reliability and privacy and security versus cluster computing on a shared resource outside of the clinic. Thus, in terms of cost, space, and convenience, GPU computing would provide many advantages over the cluster as an alternative platform for a clinical setting.

Ubiquitous. Finally, CUDA compatible hardware is already available and deployed on millions of machines, including all Apple laptops sold for the last few

[3]http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

years. Although the GPUs present in laptops and older desktops cannot deliver competitive performance with the more powerful 200 series, the ubiquity of compatible hardware does provide an opportunity by which almost anyone can learn about and experiment with the technology. In this sense, CUDA is the first readily available ubiquitous HPC platform.

1.3.7 Weaknesses of GPU Platforms

The memory bottleneck. Two memory bottlenecks are present with the CUDA platform. The first is the bottleneck of communicating between the host CPU and the device GPU. If data are needed to be frequently shuttled to and from the GPU this can be problematic. This transfer rate is limited to the peak bandwidth of the PCI-Express bus - 8 GB/s per direction for a 2.0 16x card. Thus, in the time it would take to transfer 4.4 GB to the GPU DRAM, about 1 trillion floating point operations could have been performed by the GPU or about 40 billion operations on the CPU, assuming the CPU has a peak rate of 70 GFLOPS.

Similarly, the GPU behaves best when operating on fast device registers, as opposed to device memory. The GPU operates at peak speed when it performs many operations for each memory access to hide this latency. Applications which must randomly access very large sets of data are often memory bound and will not achieve near the peak capabilities of a GPU. Of course, such applications often run slowly on CPUs, especially if the data accessed cannot fit into the CPU's caches.

Precision. The current generations of GPUs are primarily used with single precision. Although the current GPU generation supports double precision, there is a considerable performance penalty[4].

Programming Difficulty. Although GPU programming has progressed considerably over the last few years and GPU chipset developers have made significant progress in making tools for creating GPU accelerated software readily available, GPU programming is still not an easy task that is necessarily suitable for a novice programmer such as a domain scientist. As previously discussed, efficient GPU programming requires deep understanding of a new memory hierarchy and pro-

[4]also true of next generation *Fermi* chip

gramming model with new terminology and concepts, etc., and these make the learning curve for GPU programming rather steep. Competing standards, such as nVidia's CUDA, AMD's Stream, and Khronos Group's OpenCL, also increase the risk of learning a new standard as it is still not clear if a single standard will ultimately replace the others, effectively rendering the time that one has invested in any particular technology inconsequential. Rather than being locked into a single vendor, or even a single processor type, OpenCL strives to be a computing API compatible with a wide range of parallel computing devices. With OpenCL a developer can target both AMD and nVidia GPUs as well as Intel CPUs and other types of hardware. Of course in order to achieve the full potential of any specific device, OpenCL code would need to be tailored to that device's capabilities.

1.3.8 Accelerator Alternatives

AMD FireStream[5]. AMD has a competing platform for GPU development for its ATI Firestream GPUs, including a full software development kit, called the Stream Computing SDK[6]. Like CUDA, this platform is intended to allow developers to quickly create GPU accelerated applications. Rather than the C-like language used by CUDA, AMD's SDK makes use of *ATI Brook+* as an alternative C-like language. Both SDKs, though, have the same goal: to hide low-level hardware details to ease software development. Both platforms also include other tools and libraries to facilitate software development.

With AMD hardware[7], each *stream processor* has an array of *SIMD Engines*. Each engine has an array of *thread processors*, each of which has an array of *stream cores*. In nVidia parlance, an SIMD Engine can be thought of as a *streaming processing cluster*, a thread processor as a *streaming multiprocessor* and a stream core as a *streaming processor*.

Each thread processor contains 5 stream cores, one of which is capable of handling special operations (such as sine, cosine, etc.). Double precision operations can be performed by the remaining four stream cores working together. Although

[5]Formerly *ATI FireStream*
[6]http://ati.amd.com/technology/streamcomputing/sdkdwnld.html
[7]http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf

all threads within a thread engine execute the same instruction for each cycle, different thread engines can perform different operations simultaneously. Unlike nVidia GPUs, AMD GPUs support an asynchronous direct memory access (DMA) data transfer in which stream processors can exchange data with a special section of CPU RAM without CPU intervention. However, AMD GPUs do not have a shared memory for fast communication between threads.

CELL Broadband Engine. Positioned somewhere between conventional CPUs and GPUs, the CELL Broadband Engine is developed by jointly by Sony, Toshiba, and IBM for wide range of applications from video games to HDTVs. The CELL architecture includes a *Power Processing Element* (PPE) similar to a traditional single core CPU, and eight *Synergistic Processing Elements* (SPE), connected over an *Element Interconnect Bus* (EIB). While CELL's PPE can run a traditional operating system, its SPEs are specialized for SIMD and parallel use. CELL is available both as as the main system processor (such as with the Playstation 3) and as PCI-Express co-processor.

Like nVidia GPUs, CELL is optimized for single precision arithmetic, although double precision calculations are possible. CELL BE is also notoriously difficult to write software for and can require carefully tuned programs for maximum performance. This challenge has inspired researchers to explore sophisticated compilers and automatic code generation techniques that we will discuss in Chapter 5.

Chapter 2

Cardiac Modeling

Cardiac modeling, like heart research in general, is critical when we consider that heart failure is still the leading cause of death worldwide[39]. In part, this is because there is still quite a bit about the heart that we simply do not understand. Heart modeling research is also leading to clinical and therapeutic applications such as targeting ablation therapy for atrial arrhythmias, defibrillator design, and cardiac resynchronization therapy (see [35] for details and other examples). Researchers build heart models to understand some aspect of the heart's behavior which can be described by a mathematical model. This mathematical description is used to develop a simulator, based on numerical interpretations of the model.

In order to be used in a clinical setting, models would need to be able to run relatively quickly - a patient may not be able to wait several days or weeks to perform a simulation, especially if the condition is life threatening. Also, these simulations must typically be run many times in parameter sweeps in order to be useful. For example, selecting an optimum location for a pace maker lead, a surgeon might want to model several potential sites before actually implanting the device. In contrast to models used in research, clinical heart models are likely to be larger (because the human heart is larger than the mouse or rabbit or even dog), but not necessarily more complex (because we have less detailed information on human heart cells). Arguably the standards of accuracy are therefore somewhat lower too, because we can't make measurements in a patient as accurately as we can in animals. Finally, in a clinical setting a dedicated desktop workstation is

likely to have several advantages over a computing cluster. Obviously, clusters are more expensive to build and maintain, but also may raise concerns about privacy and security. Such a shared resource might not be able to provide the same level of privacy and security that a dedicated workstation likely could, especially at a comparable cost.

2.1 The Finite Element Method

While a variety of numerical methods are used for modeling cardiac behavior, the more advanced simulations tend to use the finite element method (FEM)[44],[23]. The finite element method is a numerical technique used frequently for approximating a system of partial differential equations (PDEs).

Essentially, the finite element method involves discretizing a continuous domain into a set of discrete sub-domains. The accuracy and compute time are often related by the degree of mesh finitization - the more pieces that are used to approximate the continuous domain the more accurate the solution will be, but at the expense of an increased compute time.

Finite element analysis has origins in the work done in the 1940s on structural analysis for civil and aeronautical engineering, although the method evolved during the 1950s and 60s to be much closer to what we use today [23]. More recently the approach has been applied to problems in bioengineering [44]. Although many software packages exist to perform general finite element analyses, our simulations are performed by *Continuity 6* [28], a problem-solving environment which uses the finite element method for heart modeling. *Continuity* is distributed free for research by NBCR, an NIH funded resource, which enables biomedical research by developing computational technologies. *Continuity* has been downloaded thousands of times and has over 700 registered users worldwide.

In our case we are using finite elements to simulate a time and space dependent phenomena. We split the solution into two parts: the PDEs which solve the spatially dependent part and the ODEs which solve the time dependent part.

2.2 Heart Models

Frequently, researchers are interested in using heart models to perform biomechanical and electrophysiology simulations. Biomechanics is analogous to strain and stress analysis done on buildings, bridges, and airplanes by structural engineers (see Figure 2.1). Electrophysiology involves the simulation of electrical events at the cellular level that give rise to cardiac action potentials (see Figure 2.2). When the two techniques are combined, fully coupled electromechanics simulations can be performed. Although biomechanics simulations are also useful and could benefit from the optimization techniques we present, we will focus our attention on the acceleration of electrophysiology simulations which is typically the bottleneck in heart modeling [52].

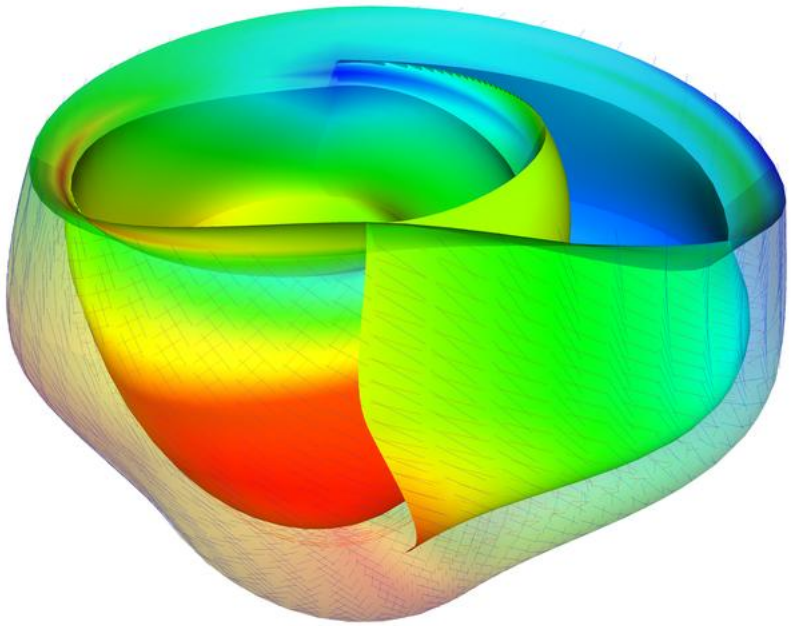

Figure 2.1: Resulting strains from a bi-ventricular canine biomechanics simulation rendered in *Continuity 6*.

With electrophysiology simulations, the generic mathematical model is a

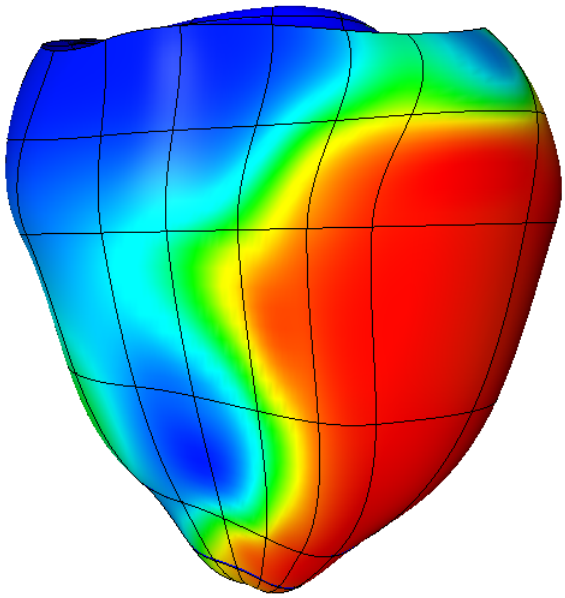

Figure 2.2: Voltage propagation in a bi-ventricular rabbit electrophysiology simulation rendered in *Continuity 6*.

reaction-diffusion system. A system of ODEs describes the kinetics of chemical reactions, and PDEs describe the spatial diffusion of the reactants. In our models, the reactions are the cellular exchanges of various ions across cell membranes during the cellular electrical impulse. The diffusion process is the flow of current through the tissue, which allows the electrical impulses to propagate. There is just one PDE of the form $\frac{dV_m}{dt} = \nabla * D\nabla V_m - \frac{I_{ion}}{C_m}$ in which I_{ion} is the sum of the ionic currents which are given by the ordinary differential equations, V_m is the voltage, and C_m is the capacitance. The chosen ODE solver advances the solution to the ODEs over time. This approach is germane to other domains besides cellular modeling, e.g. circuit simulations.

Often a population of connected cells is modeled using a *monodomain* scheme, which represents tissue as a single compartment, or a *bidomain* scheme which views tissue as two coupled, continuous domains: one for the intracellular space and another for interstitial space [21]. With either scheme, potentials and currents are defined at every point in space which yields governing equations for electric fields which can be described by a system of partial differential equations.

For our simulations, we use an operator splitting fully implicit collocation finite element method in which, alternately at each half PDE time step, we solve ODEs then PDEs. The ODEs are typically integrated with a 5 point backwards Euler adaptive time step method, although we shall shortly discuss other possibilities. The PDEs, though, result in a large sparse linear system, of static structure, which is solved at each time step.

For sufficiently complex cell models, such as the Flaim model we use here (see below), over 98% of the computational time of an electrophysiology model may be spent solving the ODE systems when executed on a single core and 80% of the time on 48 cores of an Opteron cluster. The ODEs are completely data parallel and can be efficiently solved on a parallel computer, such as a cluster, with little communication overhead. These ODEs, therefore, are the target of our GPU optimizations. A general trend in cellular modeling is that over the last few decades the models have increased in complexity. Future models have been suggested [7] that solve much larger systems of ODEs and therefore our ability to handle such systems efficiently is likely to become increasingly important.

2.3 Cell Models

Most cardiac cell models are used to solve the voltage equation which assumes the membrane acts as a capacitor allowing a charge imbalance between intra and extra cellular space [22]. Because resistance is low within the cell, the membrane integrates the flow of current leading to a single voltage difference between inside and outside of a cell. This voltage is related to the sum of ion channel currents. The ion channels in turn are described by Hodgkin-Huxley equations[22], Markov state equations[34], and Buffering and Ionic Concentration equations[24].

There exist dozens and perhaps hundreds of cell models describing electrical events [13]. Entire databases are being built to manage and curate such models [2]. Although a complete summary of the characteristics and properties of cardiac cell models is beyond the scope of this work, we briefly describe a few cell models which represent a broad spectrum in terms of complexity (see Table 2.3 for a summary).

We refer the reader to [13] for a more detailed review of the subject.

FitzHugh-Nagumo. Among the simplest and earliest of cardiac cell models, the FitzHugh-Nagumo model [14] was published in 1961 and has just two state variables. The model was derived as a simplification of the Hodgkin-Huxley equations and due to its simplicity and generality the model has been used widely. Similarly, it has been modified for specific applications, including a modified FitzHugh-Nagumo formulation (MFHN) [45] which we will use. Because the original FitzHugh Nagumo model lacks certain characteristics of cardiac cells to make it very realistic (such as separate time scales for depolarization and repolarization), it is not frequently used in modern cardiac modeling[13]. Although the modified FizHugh-Nagumo formulation is more realistic looking for cardiac cells, it is of roughly the same computational complexity as the unmodified version [45].

Panfilov. Another variation of the FitzHugh-Nagumo model, the Panfilov model [1] also uses 2 ODEs, but has more realistic looking action potentials. Although we will not explore the Panfilov model in further detail, we note that this model has very similar computational requirements to the modified FitzHugh-Nagumo model.

Beeler-Reuter. A generic ventricular model, the Beeler-Reuter model [3], published in 1977, represented a significant advancement from earlier cardiac models with its use of eight state variables, which include four ionic currents.

Puglisi-Bers. More recently, the Puglisi-Bers model [43] was suggested in 2001 to model the rabbit ventricular myocyte. This model employs 18 ODEs including 11 gating variables.

Flaim. The most recent and complex cell model we will optimize is the Flaim model [15], published in 2006. The Flaim model contains 87 variables used to model canine ventricular cells. This model is based on the Greenstein model [17], but incorporates variations to represent epicardial, endocardial, and midmyocardial cells.

Grandi-Bers. Future work will involve applying our GPU acceleration techniques to the Grandi-Bers model [16] of the human ventricle, published in 2009. Because less data is generally available for human hearts than animal hearts, the

Grandi-Bers model is somewhat less complex than recent animal models (such as Flaim) and uses only 42 ODEs. Thus, we anticipate it will be less computationally expensive than the Flaim model, but more expensive than the other models we have discussed.

Table 2.1: Cell Model Summary

Cell Model	Year	ODEs	Species
FitzHugh-Nagumo	1961	2	generic
MFHN	1994	2	generic
Panfilov	1996	2	canine
Beeler-Reuter	1977	8	canine
Puglisi-Bers	2001	18	rabbit
Flaim	2006	87	canine
Grandi-Bers	2009	42	human

2.4 ODE Solver

2.4.1 Overview

Some systems of ordinary differential equations have a characteristic commonly referred to as "stiffness". A system is said to be *stiff* if the step size we take is dictated by stability constraints more than accuracy. In other words, if we take a step size that is too large, a stiff system becomes unstable and will not yield a useful result. Complex cell models tend to be very stiff, typically because they attempt to capture events which occur on different time scales.

Stiffness can cause some solvers to suffer stability problems, including the popular 4th order explicit Runge-Kutta scheme. Much research has been done to improve the efficiency of ODE solvers by adopting more exotic schemes to reduce the computation time without sacrificing too much accuracy, especially when confronted with very stiff ODE systems. Rush Larsen [46] schemes and even 2nd order Rush Larsen variations [33, 25] have been suggested and employed to improve performance. For a review of several methods, see [51].

Frequently these schemes must balance the accuracy and stability of the solution with performance. For example, taking smaller step sizes in most schemes increases the accuracy of the solution at the cost of performance. In order to quantify this tradeoff, researchers typically compute a relative root means square error (RRMS) by comparing a method under consideration to some trusted gold standard reference solution. The RRMS is calculated as follows:

$$RRMS = \sqrt{\frac{\sum_{i=1}^{N}(V_i - V_i^{ref})^2}{\sum_{i=1}^{N}(V_i^{ref})^2}} \tag{2.1}$$

V_i is the voltage at time step i, and V_i^{ref} is the voltage for the reference computation at the same time step i. We use *radau5* [18], a 5th order Runge-Kutta backwards Euler (implicit) method with adaptive step sizes, as our reference gold standard implementation. *Radau5* has been used for all ODE solving by *Continuity 6* for several years and has demonstrated sufficient reliability even when working with very stiff ODE systems.

To integrate the ODEs in time, we used a simple single iteration backwards Euler scheme. With the Flaim model, we take a step size of .0016 which yield performance close to *radau5*. Using this small step size, our single iteration scheme produces an RRMS error of 1.48% for a 300ms simulation, a small enough error for our application. Acceptable step sizes and the resultant errors vary among the cell models. See Table 2.2 for the step sizes and resulting errors for the cellular models explored in this thesis.

We have employed the single iteration backwards Euler for several reasons. First, it is quite straightforward to implement on both the CPU and the GPU. Unlike Rush Larsen methods, it does not require separating nonlinear and quasi-linear equations which would add to the complexity of the implementation. Unlike forward Euler methods, it is unconditionally stable and will produce a valid (although potentially inaccurate) solution regardless of the stiffness of a particular cell model. Finally, unlike *radau5*, our scheme has a fixed step size, and thus we

Table 2.2: A comparison of step sizes which achieved an RRMS error of less than 1.5% when compared to the radau solver. Runtime is for a 300ms simulation of a single cell.

Cell Model	Step Size	Runtime (s)	RRMS Error
MFHN	0.001	1.29	0.91%
BR	0.00054	2.35	1.14%
Puglisi	0.0008	3.12	0.82%
Flaim	0.0016	5.19	1.48%

avoid the problem of divergent threads.

While this backwards Euler method is not the only possible method we could have used, it yields small enough errors as to be sufficiently accurate for our application, and was amenable to our automatic code generation scheme which made a GPU implementation straightforward.

The runtime of our backwards Euler implementation was dependent upon the step size which also impacts the error. We modified the step size until it yielded an acceptable RRMS error (less than 1.48%) when compared to *radau5*. At this error, our backwards Euler implementation required approximately the same runtime as *radau5*. Although *radau5* can take larger and therefore fewer time steps, it has to do much, much more work at each time step.

Although we have not implemented *radau5* on the GPU, our experience suggests that it would make a poor GPU solver anyway. In addition to adding considerable complexity (at least another 1000 lines of code) and requiring additional state space for the Jacobian and intermediate state vectors, *radau5*'s adaptive step sizes are not well suited for the SIMT architecture of the GPU. For example, at any given moment during the simulation, different collocation points might require a different number of time steps to achieve sufficient accuracy. This would lead to tremendous thread divergence and would thus likely cripple performane.

We also implemented several other single precision solvers including a second order Rush-Larsen scheme, several forward Euler variations, and backwards Euler with adaptive time steps. We implemented the Forward Euler methods for both the CPU and in CUDA for the GPU. What we found was that none of the other solvers improved accuracy and they also did not improve performance over

our backwards Euler scheme with the Flaim model. The other solvers also all required much more state space which would lead to more register pressure, more global memory references, etc., which might also lead to performance degradation in CUDA.

It may also be that the other solvers work better in double precision and that a single iteration backwards Euler method is very well suited for single precision - an topic suitable for exploration in future work. Thus, we do not claim to have the best possible method for solving systems of ODEs on the GPU, but rather a method that delivers excellent performance and accuracy when compared against *radau5*, a solver that has been in use for solving these types of systems for many years. Further, we demonstrate our method is successful even when confronted with an extremely complex and stiff cell model.

Next we briefly review various numerical methods for solving ODE systems to motivate the formulation of our simplified method. As we discuss these methods, we will use the following symbols. y_n is the vector of state variables at time step n. y_{n+1} is therefore the vector of state variables at the next time step. h is the step size which is typically fixed but can be variable if using an adaptive time stepping scheme. y'_n is the vector of the derivatives of the state variables which is computed by the user defined function $f(y_n)$, which we also refer to as the "cell model."

2.4.2 Forward Euler Method (FE)

Perhaps the simplest numerical method used to calculate the next time step in a ODE system is Euler's Method:

$$y_{n+1} = y_n + hy'_n \tag{2.2}$$

The main benefits of this scheme are its simplicity both in terms of implementation and comprehension. With this simplicity, though, comes both inaccuracy and instability making this method unsuitable for most applications.

2.4.3 Explicit Runge-Kutta Methods (ERK)

Closely related to the simple Forward Euler method, is the family of Runge-Kutta methods. Forward Euler, in fact, can be though of as a first order Runge-Kutta method. Perhaps the most popular method in this family is the 4th order explicit Runge Kutta method:

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \\ k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1) \\ k_3 &= f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{aligned} \tag{2.3}$$

Although h is typically a constant step size,variable h has also been used.

2.4.4 Backwards (Implicit) Methods

In contrast to explicit forward Euler methods, there are also implicit or backwards Euler methods. The simplest of such formulations is:

$$y_{n+1} = y_n + hy'_{n+1} \tag{2.4}$$

Like their forward counterparts, a variety of backwards Euler methods variations have been suggested.

2.4.5 Backwards Euler Method with Newton Update

As mentioned previously (see Equation 2.4), the backwards Euler Method is:

$$y_{n+1} = y_n + hy'_{n+1}$$

Because y_{n+1} is found on both sides of the equation, we can use the Newton-Raphson method to approximate a solution. Given a guess x_0, a better approximation x_1 is defined as follows:

$$x_1 = x_0 - \frac{F(x_0)}{F'(x_0)} \tag{2.5}$$

Let y_{n+1} be defined as x_0 and $\hat{y}_{n+1}$, the better approximation, as x_1. Using the Backwards Euler equation and assuming $y'_{n+1} = f(y_{n+1})$, we define $F(y_{n+1})$ as follows:

$$F(y_{n+1}) = y_{n+1} - y_n - hf(y_{n+1}) \tag{2.6}$$

$\frac{dF}{dy_{n+1}}$ or F' is therefore:

$$F'(y_{n+1}) = 1 - hf'(y_{n+1}) \tag{2.7}$$

Given our initial y_{n+1}, we can approximate a more accurate solution $\hat{y}_{n+1}$ as follows. We use F and F' with the Newton-Rhapson equations, yielding:

$$\begin{aligned} \hat{y}_{n+1} &= y_{n+1} - \frac{F(y_{n+1})}{F'(y_{n+1})} \\ \hat{y}_{n+1} &= y_{n+1} - \frac{y_{n+1} - y_n - hf(y_{n+1})}{1 - hf'(y_{n+1})} \end{aligned} \tag{2.8}$$

We can continue this process iteratively until some stopping condition (e.g. error estimation or number of iterations) is met.

2.4.6 Simplified Backwards Euler Method

If we do just a single iteration of Newton's Method and let our initial y_{n+1} guess be equal to y_n, the equations simplify as follows:

$$y_{n+1} = y_n + \frac{hf(y_n)}{1 - hf'(y_n)} \tag{2.9}$$

And if we take a small enough time step h, this scheme may achieve sufficient accuracy. Indeed we will use this scheme in our automatic code generator for both the GPU and the CPU. We judge accuracy by comparing the RRMS error of the voltages against that of *radau5* (see Section 2.4.7 for a description of *radau5* and Table 2.2 for step sizes and errors for a variety of cell models.)

2.4.7 Radau5

The *radau5* solver implements a 5th order adaptive implicit Runge Kutta with full jacobian method [19]. We use this implementation as a "gold standard" by which to judge the accuracy of other methods.

The numerical method used by *radau5* is considerably more complex than the other methods we discuss due to its adaptivity and calculation of a full Jacobian. The method is implemented as 850 lines of Fortran for its core functionality, plus an additional 650 lines to solve a linear system and calculate error estimations. The calculation of the right-hand side is most analogous with the other methods we discuss and involves computation of 3 derivatives of this form:

$$\begin{aligned} c1 &= \frac{4-\sqrt{6}}{10} \\ c2 &= \frac{4+\sqrt{6}}{10} \\ k_2 &= f(t_n + c1 \times h, y_n + z1) \\ k_3 &= f(t_n + c2 \times h, y_n + z2) \\ k_4 &= f(t_n + h, y_n + z3) \end{aligned}$$

The values of $z1$, $z2$, and $z3$ change dynamically as the solver runs based on the adaptively changing step sizes.

2.4.8 Rush Larsen Methods

These methods typically solve "quasi-linear" equations analytically and other equations using forward Euler or something fancier. The original formulation was suggested by Rush and Larsen [47] in 1978 and achieved only first order accuracy. More recently, second order accurate methods have been suggested by [33, 25]. Although these methods may achieve significant performance gains with very little loss in accuracy, they can be difficult to setup for very large ODE systems which is perhaps why they are not used as frequently as simpler, less efficient formulations.

Chapter 3

Translation

Building a highly tuned CUDA application can be a challenge, even for the expert programmer. In order that our simulator be usable by a domain scientist, it must deliver acceptable performance without requiring that the model developer have expert GPU programming knowledge and skills. Therefore, we have built a translator that accepts a higher level cell model description, automatically generates highly tuned CUDA kernels, and invokes a framework to integrate the kernels into a runnable simulator. This approach enables the model developer to focus on cell modeling without worrying about complicated implementation details, and enables the experts to provide optimizations that are customized to the hardware at a time when hardware platforms are undergoing continual change. We next describe how the translator generates code and how it deals with singularities introduced by using single precision instead of double precision arithmetic.

We also note that our approach to generating optimized source code from a high level cell model description is amenable to other platforms. For example, we can also generate OpenMP CPU code from the same high level cell model which we use for performance comparisons.

3.1 Code Generation Overview

We use *sympy* [8], a Python library for symbolic mathematics, to handle code generation and perform automatic symbolic optimizations (see Section 3.3

for additional details about the *sympy* library). Using our software environment, the model author specifies a list of state variables, parameters, and the equations which comprise the cell model (see Listing 3.1 for an example of the equations specified by a domain scientist to create a modified FitzHugh Nagumo model). This is specified in a high level python format which uses the *sympy* library. Our translator then compiles this model into C code using *sympy*. Next, we generate the appropriate function signatures, variable declarations, etc. to legitimize the C. We then use *pycparser* to build an abstract syntax tree (AST) corresponding to the C code [6]. We traverse the code's AST in order to construct a list of dependencies for each state variable. Using the dependence information and the C expressions, we generate a CUDA kernel that fuses the code for the cell model and the single iteration backwards Euler ODE solver. For the Flaim model, our translator generates a CUDA kernel 4917 lines long. The translator next performs various CUDA optimizations specific to the intricacies of nVidia hardware. Once the optimized kernels have been generated the translator calls *nvcc* to produce an executable, which is invoked indirectly via python. In the future our code generator will allow equations to be specified in an xml style format for compatibility with other cell model databases, such as cellml [9].

```
1  stim_start , stim_dur , stim_mag = params
2  ug, vg = state_vars
3  vmax = 1.0
4  vrest = 0.0
5  cm = 1.0
6  a = 0.130
7  b = 0.0130
8  c1 = 0.260
9  c2 = 0.10
10 d = 1.0
11 stim_end = stim_start + stim_dur#!end time of stimulus
12 heavi = Heaviside(t-stim_start) * Heaviside(stim_end-t)
13 i_stim = stim_mag*heavi
14 ug_norm = (ug-vrest)/(vmax-vrest)
15 dug_dt = (ug_norm * (ug_norm - a) * (1.0 - ug_norm) * c1 - c2 * vg
       * ug_norm) * (vmax - vrest) + i_stim * (1.0 / cm)
16 dvg_dt = b*ug_norm - b*d*vg
```

Listing 3.1: High level python description of model which makes use of the *sympy* library. By allowing the user to create the model as a high level description it relieves him of the burden of dealing with low level GPU implementation details

Before diving into the details of our translator, we describe a few pre-requisites including Heaviside functions, ODE Solvers, and numerical singularity removal, all of which our translator leverages for code generation.

3.2 Heaviside Functions

Conditionals in the form of *if* statements commonly arise in cell models. However, the use of *if* statements presents us with two difficulties. The first, is that because we rely on *sympy* to symbolically evaluate the cell model's equations and generate C code, we are unable to propagate conditionals into the generated code - if the user were to add *if* statements into the code they would be evaluated by *sympy* at compile time rather than at runtime when needed. A second difficulty is that the nVidia GPU penalizes if statements which result in thread divergence.

The *Heaviside step function* addresses both challenges - thread divergence and the limitations of code generation with *sympy*, without sacrificing our ability to represent a model which varies spatially and temporally. Heaviside functions work as follows: if Heaviside() is passed a value greater than zero it returns 1.0. If it is passed a value less than zero, it returns 0.0. And if it is passed a value of exactly zero, then it returns 0.5. For example, one implementation of a Heaviside function is:

Algorithm 1 Heaviside($myvar$):

```
if myvar > 0 then
    return 1.0
else if myvar < 0 then
    return 0.0
else
    return 0.5
end if
```

With an alternative Heaviside implementation, we can make use of predication and thus avoid conditionals[1]:

Algorithm 2 HeavisidePredication($myvar$):

```
return sign(myvar)*0.5
```

Thus, by providing a Heaviside() function we can obviate the need for *if* statements common in cell models. For example:

```
if time > 5 then
    stimulus = 10.0
else
    stimulus = 0.0
end if
```

Could be rearranged with a Heaviside function like this:

[1]Although CUDA does not support a sign() function, this behavior can be achieved with signbit()

stimulus = Heaviside(time-5.0)*10.0 + (1-Heaviside(time-5.0)*0.0)

or simply:

stimulus = Heaviside(time-5.0)*10.0

Although arguably conditionals may be more natural for a domain scientiest, our experience indicates that it is not difficult to convert cell models with conditions to use Heaviside functions instead. Also Heaviside functions have useful mathematical properties[2] that may make them attractive such as a well defined derivative[3].

3.2.1 Singularity removal

With the current generation of nVidia GPUs, the double precision arithmetic rate is significantly lower than that of the single precision[4]. If unchecked, roundoff errors in single precision can introduce unacceptable errors into the simulation.

For example, in the Flaim model[5], certain expressions arise that lead to singularities at magic voltages; when we use double precision arithmetic, voltages never quite reach these magic numbers, and are far enough away that the equations are benign in practice. At first glance, some of these problematic equations appear to be in the form $a = \frac{b}{e^v - 1}$ and since $\lim_{v \to 0} \frac{1}{e^v - 1} = \infty$, letting v be zero appears to be illegal, even though a zero voltage is perfectly legal in these biological systems. Upon closer inspection, though, we found that the equations with these singularities are actually of the form $a = \frac{bv}{e^{cv} - 1}$, in which case the limit does not go to infinity, but rather some constant based on b and c.

In order to tolerate the roundoff errors committed in single precision, a variety of possibilities are available. First, the presence of these singularities may suggest that the cell model author might want to reformulate his equations manually so that these singularities do not occur. Using an asymptotic expansion may

[2] http://mathworld.wolfram.com/HeavisideStepFunction.html

[3] Bracewell, R. "Heaviside's Unit Step Function, H(x)." The Fourier Transform and Its Applications, 3rd ed. New York: McGraw-Hill, pp. 61-65, 2000

[4] Older GPUs (before the 200 series) do not support double precision arithmetic.

[5] We also found that the Puglisi model [43] suffered the same difficulty.

be useful in this reformulation and is likely the preferable solution to this problem. Our tool would facilitate this process by identifying the equations with singularities near certain values.

Alternatively, we implemented an automatic transformation that traverses the equations and employs a root finder to eliminate division by zero. This strategy avoids the most critical round-off errors introduced by single precision and enables us to benefit from the higher single precision computational rate. Since we already have the AST, we can search for problematic denominators. Specifically, we employ *sympy*'s solve() method (a root finder) to identify which input values would lead to divide by zero errors. Once these illegal values are identified we add a very small offset to problematic values as they appear in arithmetic expressions. For example, the following lines would prevent the round-off problem that occurs near voltages of zero:

```
1 if (abs(voltage) < 1e-4)
2   voltage + sign(voltage) * 1e-3;
```

While this strategy introduces a small error, we found the overall effect on the simulation to be negligible. Although the conditional can cause threads to diverge, they do so only momentarily and are quickly resynchronized (see Section 5.1.1.2 of [36]). We do caution, though, that we selected $1e-4$ carefully. If this value were too small we would occasionally miss voltages close enough to the singularity to cause problems. If the value were too large it could cause the voltage to get "stuck" at zero when the voltage is changing very slowly.

Yet another approach would be to calculate the limit as voltage approaches the singularity, and replace the equation with the limit. In this case, we use *sympy*'s limit() to automatically calculate an alternative. We can then use a formulation like this:

```
1 if (abs(voltage) < 1e-4)
2   a = precomputed_limit;
3 else
4   a = f(v);
```

There are a few challenges, though, that make this approach difficult to use in practice. Some of the equations are too complex for *sympy*'s limit() function to be able to handle. Also, singularities quickly propagate into other equations and it is easy to create the situation where we need to check every equation against a set of magic values which would lead to a significant performance penalty

Since our first automatic solution (adding a small offset) approximates the limit anyway, and the error introduced is quite small, it may be sufficient to use in practice.

This approach is applicable to any equation in which a numerical singularity arises due to imprecision around a certain value which can be replaced by its limit around that value. Although we aren't aware of other domains with equations of this exact form, the equations are derived from physical phenomena which give rise to ODEs and thus are likely to occur elsewhere.

3.3 Translator Tools

In order to facilitate the code generation process, we employed several prebuilt libraries and tools which we now discuss in detail.

Python[6]. Python is a general purpose programming language useful over a broad spectrum of application domains from server side web programming to heart modeling. Python is often referred to as a *scripting language* as it is interpreted and very high level. Python is extremely dynamic and has gained a great deal of support in the high performance computing community over the last few years due to its flexility and ease of use. There are a number of scientific and numerical libraries available for python such as *SciPy* and *numpy* which also make it attractive for domain scientists. We used python version 2.5.4 for all the results given, although when we tested with python 2.6.2. it yielded no significant difference.

Because it is a dynamic interpreted language, python executes rather slowly when compared to a compiled language. However, there are straightforward tools to *extend* python with compiled C, C++, and Fortran codes so that performance bottlenecks can be reduced when necessary while still managing the overall logic of an application from a very high level scripting language.

Python is also popular to use for *metaprogramming* or writing programs to write other programs and various libraries exist to facilitate this process. We used two such tools which we will discuss next.

Pycparser[7]. *pycparser* is a python library for parsing C source code. Specifically, the library can take C source code and build an AST representation as a python data structure which is a convenient structure for code analysis and manipulation. We used version 1.04 of *pycparser*.

Sympy[8]. *sympy* is also a python library, but useful for symbolic manipulation of expressions and variables. It also has simple faculties for code generation and manipulation. The domain scientist can create their cell model equations in a *sympy* compatible format which our translator converts to other representations

[6] http://python.org
[7] http://code.google.com/p/pycparser/
[8] http://code.google.com/p/sympy/

and ultimately to CUDA source code. We used version 0.6.4 of *sympy*.

CUDA Toolkit[9]. The CUDA toolkit is a collection of tools (most notably the *nvcc* compiler) for generating nVidia GPU compatible binaries from CUDA source code. We used release 2.3, V0.2.1221 of the CUDA Toolkit, and used the flags maxrregcount=64 and ptxas-options=-v -use_fast_math with *nvcc*. nVidia also releases a separate package which it terms the CUDA SDK (software development kit) which contains example CUDA source code rather than development tools.

GCC. For compiling our C codes, we used the GNU Compiler Collection (gcc), version 4.3.3. with -fopenmp and -fPIC flags.

Decuda[10]. Finally, we make use of *decuda*, which can take a binary produced by nVidia's *nvcc* compiler and generate a ptx assembly file. Because certain stages of the nVidia compilation process are proprietary, disassembling the code is the only way to determine what is actually being executed on the GPU. We used version 0.4.1 of *decuda*. The authors of *decuda* built the tool was built by "differential analysis on the cubin files by produced by ptxas and extensive experimentation" rather than by reverse engineering of the nVidia tools. *decuda* is not associated with nVidia and might contain errors.

3.4 Translator Details

We now provide the details of the internals of our translator implementation (see Figure 3.1 for a pipeline). As evident from Listing 3.1, the user first specifies the parameters and state variables used by their model. Our translator takes this information to build the appropriate C variable declarations. Each parameter requires a variable declaration and each state variable requires two variable declarations - one for the state variable's current value and one for the state variable's derivative, which is the output of the cell model. For example, if the user specifies the state variable v we create the variables v and dv_dt.

Next the model author specifies a series of equations which define his cell model. The cell model equations define the derivative of each state variable. In

[9]http://www.nvidia.com/object/cuda_get.html
[10]http://wiki.github.com/laanwj/decuda

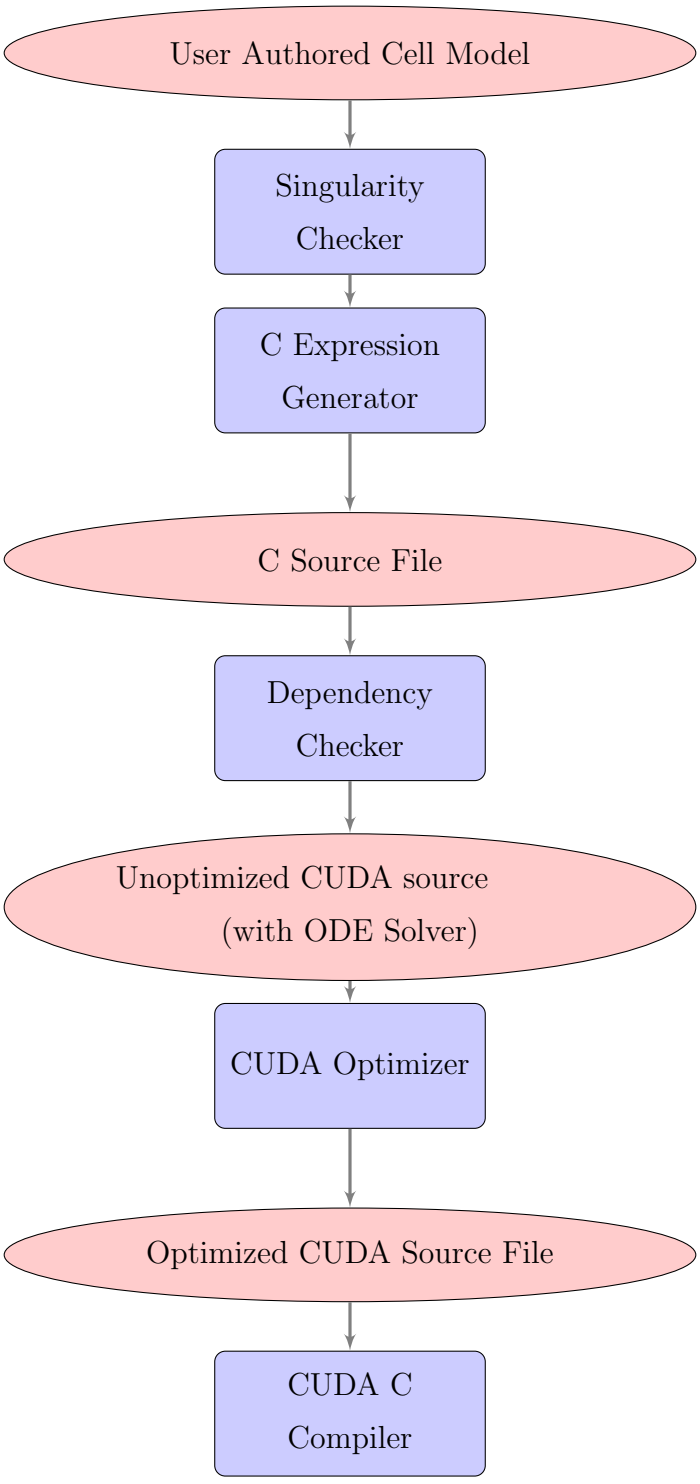

Figure 3.1: Pipeline of translator. The output of the final stage is CUDA compliant code and will be compiled by the CUDA C Compiler (*nvcc*).

other words, the input to a cell model is the current vector of state variables, and the output is the derivative for each variable. This is the standard formulation by which cell models (or other systems of ODEs) are defined.

In addition to the equations defining the derivative of each state variable, the cell model author may also specify constants or temporary variables which are used by subsequent equations.

Next, our translator takes the user specified list of equations and constants and traverses them, generating C code as it goes. If an equation can by symbolically evaluated as a constant, we do not declare a corresponding C variable, but instead simply substitute the variable for its evaluated constant whenever it is used. *sympy* is able to handle this substitution automatically. Otherwise, if the equation cannot be evaluated as a constant, we declare a C variable for the LHS of the equation, and use *sympy*'s printing.ccode() function to generate a C expression from the *sympy* equation. For example, the *sympy* equation v**2 becomes pow(v,2) in C.

As we traverse equations, we use *sympy*'s solve() function on each denominator to detect divide by zero singularities. These are reported to the user, who can then choose to optionally apply our singularity removal technique by applying a small offset. The user might choose to ignore certain singularities, such as with a voltage of -420, as the user may know that in his model the voltage may only ever be between -90 and +20.

Once we have generated the C code, we then use *pycparser*[6] to parse our C code to generate information that is useful for subsequent steps in the code generation process. Specifically, we create a python data structure with an AST representation of the C code. We can then traverse this tree to acquire whatever information we need about the C code without having to write our own C parser.

We need this AST to build a dependency graph for each variable in our cell model. For example, if an assignment is v = a * b / (c + d), we are able to recognize that v depends on a, b, c, and d. See Figure 3.2 for an example AST of this expression. We then work our way backwards to identify which a, b, c, and d are associated with this assignment; there could be multiple assignments for each variable and we must find the relevant assignment. We can make this selection

confidently as our cell models do not allow branching due to if statements. As mentioned previously the cell model author can use Heaviside() functions if he must assign a variable based on some runtime criteria, such as setting a stimulus at a particular location in the model at a particular time.

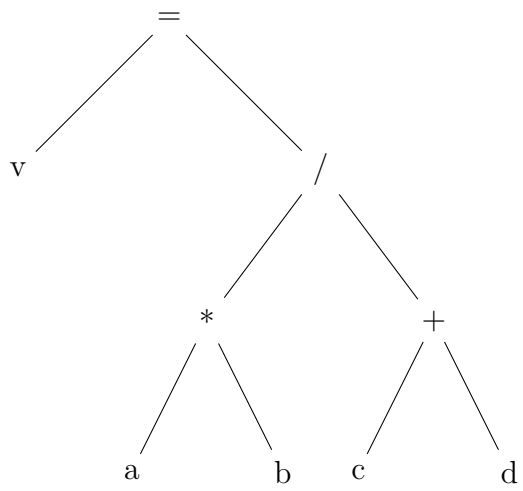

Figure 3.2: Simple AST for the expression $v = a * b/(c + d)$. A data structure with this information can be used to identify that v depends on a, b, c, and d.

After creating a mapping of each assignment and all dependencies for each assignment, we are able to generate the time stepping code for each equation in our cell model. We do this by looping over each state variable and printing its dependencies in the order in which they are required.

While we do this, we also build in the single iteration backwards Euler integration scheme in order to calculate a subsequent time step. As discussed in section 2.4.6, the backwards Euler formulation we use involves the equation:

$$y_{n+1} \quad = \quad y_n + \frac{hf(y_n)}{1 - hf'(y_n)}$$

Thus, in order to compute y_{n+1} we need y_n (the current value of y), h (the step size), and $f(y_n)$ (the derivative of y_n) and $f'(y_n)$ (the derivative of $f()$). We can estimate $f'(y_n)$ with a finite difference calculation. For example, suppose that the cell model author specifies $f = exp(param1) + y/2$ and $x = exp(param1)$. Then we would generate the following expression:

```
1 y += delta;
2 x = exp(param1)
3 f_delta = x + y/2;
4 y -= delta;
5
6 x = exp(param1)
7 f = x + y/2;
8 f_deriv = (f_delta - f)/delta;
9 y_new = y + (h*f)/(1-h*f_deriv);
```

Listing 3.2: Backwards Euler implementation.

Thus, when we calculate f' using finite differences, some small *delta* (such as $1e-4$) is added to y to compute f_delta and then removed before computing f. In other words, by dividing the difference between f_delta and f by *delta* we approximate f'. Also we note that this involves repeating the calculations for each state variable twice as our ODE scheme requires a small perturbation (e.g. a finite difference) in order to calculate the next time step. This may introduce redundant calculations (in this case x=exp(param1) was unnecessarily computed twice), but we can automatically remove any unnecessary redundancies as we discuss in Section 3.5.

After generating the source code which combines the ODE solver with the cell model, we add the appropriate CUDA specific requirements to legitimize the code. For example, a CUDA kernel must begin with the __global__ keyword so that it can be distinguished from other CPU functions intermingled in the file. We also add the appropriate function header, declarations, etc., that are required.

Finally, we perform another pass over the source code to apply additional optimizations. For example, one potential optimization is to replace global memory references with shared memory. To perform this operation, we search for variables of the form y_global[varNum*num_gauss_pts + idx] and replace them with y_smVarNum[threadIdx.x], where varNum a state variable number. Other global memory optimizations are applied in a similar manner, and the various possibilities are detailed in the following section.

3.5 Automatic Optimizations

Our code generator applies various optimizations during its different phases of operation. It applies these optimizations cumulatively. For our simpler cell models (i.e. MFHN, BR, and Puglisi), Optimization 1 and 2a, detailed below, are sufficient to lead to the maximum performance benefits we were able to obtain. For these less sophisticated cell models, we simply eliminate redundant calculations (see Opt1 below), and copy state variables from global memory to register memory. Because these models do not have excessive state space requirements (2, 8, and 18 state variables) we are able to store all the state space in registers and still have enough registers left over to work with such that we do not spill to the much slower global memory.

With the Flaim model, however, the situation is more interesting. With 87 state variables, plus more complex equations, the compiler is unable to manage the data without spilling to local memory. Thus, when confronted with an overly complex kernel requiring too much state space, we explore a variety of optimizations to improve performance. As we detail these optimizations below, we will therefore do so in the context of the Flaim model. A summary of the effect of the optimizations we applied to various cell models can be found in Tables 4.2 and 4.6.

We begin with a "naïve" implementation, which is the result of composing our cell model with the backwards Euler method for numerical integration. Composing the two modules introduces redundant calculations. For example, let's suppose that the user provided the following *sympy* code:

```
1 param1, param2, param3 = params
2 a, b = state_vars
3 x = exp(param1)/exp(param2) * pow(param3,param2)
4 da_dt = a*x*2
5 db_dt = b*x*3
```

Listing 3.3: Sample cell model input

After identifying state variable dependencies, and combining with the equations for a single iteration backwards Euler integration method, the following code

is generated:

```
1  //da_dt
2  y_global[0*num_gauss_pts + idx] += delta;
3  x = (pow(rpar_global[2*num_gauss_pts + idx],rpar_global[1*
       num_gauss_pts + idx]) * exp((rpar_global[0*num_gauss_pts + idx]
       - rpar_global[1*num_gauss_pts + idx])));
4  da_dt = a*(2 * x);
5  y_global[0*num_gauss_pts + idx] -= delta;
6  aj = da_dt;
7
8  da_dt = a*(2 * x);
9  bi_be = (aj - da_dt)/delta;
10 y_global_new[0*num_gauss_pts + idx] = y_global[0*num_gauss_pts +
       idx] - (-da_dt*dt)/(1-dt*bi_be);
11
12 //db_dt
13 y_global[1*num_gauss_pts + idx] += delta;
14 x = (pow(rpar_global[2*num_gauss_pts + idx],rpar_global[1*
       num_gauss_pts + idx]) * exp((rpar_global[0*num_gauss_pts + idx]
       - rpar_global[1*num_gauss_pts + idx])));
15 db_dt = b*(3 * x);
16 y_global[1*num_gauss_pts + idx] -= delta;
17 aj = db_dt;
18
19 db_dt = b*(3 * x);
20 bi_be = (aj - db_dt)/delta;
21 y_global_new[1*num_gauss_pts + idx] = y_global[1*num_gauss_pts +
       idx] - (-db_dt*dt)/(1-dt*bi_be);
```

Listing 3.4: Sample cell model output with redundant calculation of temporary variable x

When our code generator built a list of dependencies for *da_dt* and *db_dt*, it found that x was required for both state variables. Naïvely, our code generator duplicates the calculation for x, which in this case is quite expensive! However, because both x computations are the same and since $param1$, $param2$, and $param3$ (the variables on which the calculation of x depends) do not change between cal-

culations, we can safely eliminate the second assignment of x. This leads us to our first optimization.

Optimization 1 automatically removes redundant calculations by searching and modifying the AST. The *decuda* disassembly tool [53] establishes that, prior to this optimization, the generated Flaim model kernel comprises 19,747 assembly operations. The optimization reduces the number of assembly operations by a factor of 3, to 6,685. We use this variant as the baseline for performance. We note that this choice could be considered overly restrictive, since we don't how successful a programmer would be in recognizing opportunities for removing redundant computations. An assignment statement is redundant only if none of the variables on the RHS have changed since the last assignment, and this would be difficult to verify in 6,685 lines of entangled assignment statements.

The next optimizations (2a-2c) are mutually exclusive. *Optimization 2a* copies global variables into local variables. We observed that with simpler cell models having fewer variables, the compiler can allocate the global variables to fast registers. With larger models, though, the compiler warns us that it is spilling variables to *local memory* which, physically located in device memory, still suffers the same 150x latency penalty as global memory. However, our tests indicate that the CUDA compiler spills variables very effectively and even when confronted with many more state variables, the local memory usage did not increase dramatically (see Table 4.2). We also suspect that because local variables are simpler to reference (e.g. they do not require a thread index) register pressure may have decreased, which may have allowed more local variables without much more local memory spillage.

Optimization 2b: (MFR) As an alternative to optimization 2a, each thread, at run time, copies its most frequently accessed global variables into shared memory. Because our kernels required more than 40 registers, our device occupancy was already limited to 5 thread blocks of 64 threads. Thus, we had storage for 11 variables in each thread without further reducing our occupancy. This strategy uses an offline algorithm - there are virtually no branches in our code, a characteristic of cellular models. In general an offline caching algorithm is not feasible

because at compile time it is impossible to determine which branches will be taken and thus which data are needed in the cache. However, in code without branches cache placement can be determined at compile time, thus obviating the need for a runtime caching mechanism.

Optimization 2c: (MIN) Instead of statically allocating the most frequent accessed globals to shared memory, we use a dynamic cache replacement strategy, based on Belady's MIN page swapping algorithm [4]. When the shared memory cache is full, we evict the variable whose next reference is the furthest in the future, because all other cached variables will be needed sooner. Like optimization 2b, this strategy is done off-line and we had sufficient storage for 11 variables at a time. With an 11 variable shared memory cache, we are able to achieve a hit rate of 88%; not far from the rate that would be achieved by an infinite sized cache, since the first global reference for each variable is always a miss. See Appendix A for an example of the application of Optimization 2c applied to the FitzHugh-Nagumo cell model.

Optimization 3: Kernel partitioning. This optimization is combined with one of 2a, 2b, or 2c. Our model is highly complex, and contains 87 state variables referenced within 1800 lines of CUDA code. The CUDA compiler, *nvcc* is challenged to manage register allocation. If we attempt to compile our single monolithic kernel, numerous variables will be spilled from the fast registers to the slow local memory. Our optimizer splits a model's code into separate CUDA kernels to reduce register demand, which the CUDA compiler can then manage on its own. The best kernel size appears to be right after the kernel has just begun to spill to local memory. This optimization corresponds to loop fission, a strategy used in vectorizing compilers dating back two decades [27].

We partition kernels at the boundary between state variables updates, corresponding to the ODEs specified in the higher level user input. Each kernel saves the new value for each state variable that it is responsible for computing into global memory. We determine the dependencies for each state variable from the AST, so we can treat each state variable as its own independent unit and not worry about saving any intermediate calculations between kernel calls. Saving these variables

would require writing them to global memory, since global memory is the only writable memory space persistent after a kernel's execution. See the appendix for an example of how MFHN might be partitioned into two pieces with this technique.

Optimization 4: Dual GPUs. With this optimization we make use of both GPUs present in the GTX 295 video card. CUDA requires that each GPU be managed by a separate CPU host thread. We used the pthreads library to create a separate CPU thread to manage and control the second GPU.

Chapter 4

Results

4.1 Simulation Description

To benchmark the bottlenecks of an electrophysiology system, we use *Continuity 6* to model a left ventricular electrophysiology simulation using a variety of representative cell models and a total of ∼44,000 degrees of freedom for the transmembrane potential (see Figure 4.1 for a *Continuity 6* rendering of the model). This model involves 5280 finite elements and therefore 42,240 collocation points, since each element has 8 collocation points. These collocation points exist at the level of parallelism we wish to exploit: at each ODE time step, each point must solve its own independent ODE system.

In addition to the ODEs, we also must solve a system of PDEs much less frequently. Solving the PDEs requires solving a linear system of equations in which the LHS matrix is very sparse. We use a direct solver, SuperLU [11], which also executes on the CPU, but can take advantage of many cores to achieve superior performance.

For the performance results we present in Figure 4.3, we only consider the speedup achieved with the ODEs and discuss PDE acceleration in Section 4.5.

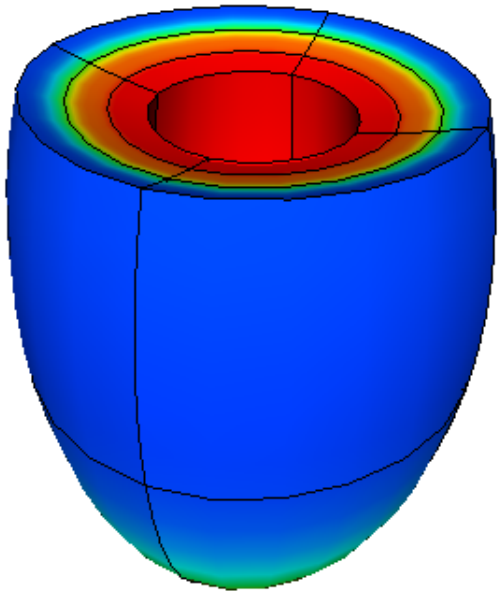

Figure 4.1: Single ventricle electrophysiology simulation rendered in *Continuity 6* using the Flaim canine cellular model.

4.2 Hardware Testbed

MPI cluster. Assembled in 2007, the cluster consists of 68 Dell PowerEdge SC6590/SC1435 enterprise servers equipped with the AMD Opteron 2216 dual core processors, running at 2.4 GHz clock, 1MB Cache and 1 GHz HyperTransport. Each server has 4 cores, 4 GB of RAM. The entire cluster has a theoretical peak performance of 1.3 TFLOP And runs 64 bit Rocks 4.2.1 (CentOS). A 10 Gbps (4X) InfiniBand network is managed by a Cisco SFS 7009 server switch, with 1.92 Tbps full bisectional bandwidth[1]. We used gcc version 3.4.6, with -fPIC, -O2 and -finline-functions flags when compiling C codes.

Desktop. The desktop comprises both a multicore processor and a GPU. The multicore processor is a dual socket Intel Quad Core i7 CPU 940 (2.93GHz), with 12GB RAM. The GPU is an nVidia GTX 295, a single card with two GPUs. Each GPU has 240 single precision units running at 1242 MHz with a theoretical peak performance of 894 GFLOPS. Our device is 1.3 capable.

[1]Wilfred Li, NBCR, Private Communication, 2007

4.3 ODE

4.3.1 CPU Performance and Optimizations

Before employing a GPU implementation of our ODE solver, we review more traditional high performance computing platforms: multithreaded CPU and multi-CPU cluster. The results are summarized in Figure 4.2. Put simply, both OpenMP CPU implementation and MPI Cluster implementations result in dramatic speedups over single thread implementations, although the cluster suffers from diminishing returns as we use additional CPUs (as the communications costs between CPUs start to dominate the computations). It is also worth noting that a quad-core i7 processor achieved speedups beyond 4 OpenMP threads, likely due to the effects of hyperthreading. See Table 4.1.

We also explored the potential of using SSE instructions to vectorize instructions to improve performance by using gcc's auto-vectorization (-ftree-vectorize -msse2). While a small amount of automatic vectorization was possible by the compiler, it was largely unable to vectorize the cell models for several reasons. First, the mere complexity of our code made auto-vectorization of large loops difficult. We employ strided memory references which also proved challenging. Finally, our loops possess rare but essential conditional statements which also create uncertainties from the compiler's perspective which cause autovectorization to fail. While these conditional statements cause momentary thread divergence on the CUDA platform, the thread divergence is only temporary (see Section 3.2.1 for additional details).

Table 4.1: The speedup achieved by adding additional OpenMP threads. Times measure the running time to simulate 20ms of a heart beat for a mesh with 42,240 collocation points.

Procs	Time (s)	Speedup
1	6.7553	1.0
2	3.4061	2.0
4	1.7099	3.9
8	1.0658	6.3

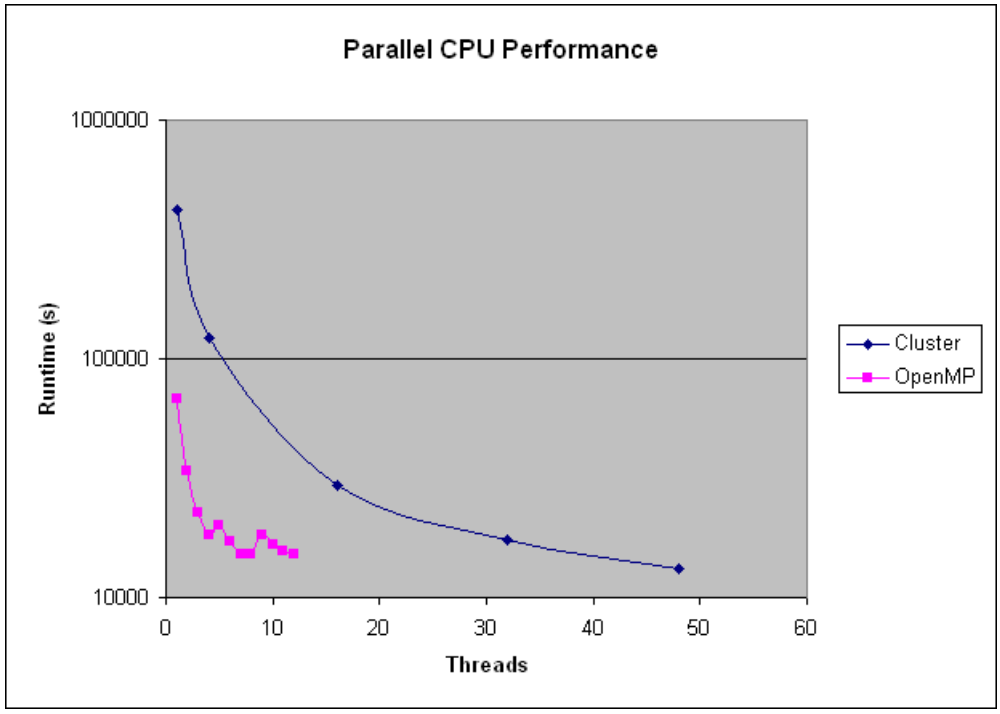

Figure 4.2: OpenMP (on a workstation) vs MPI (on a cluster)

4.3.2 GPU Performance and Optimizations

In Table 4.2 we see the effects of applying various optimizations to the Flaim model. To understand these optimizations and their effects we must be careful to distinguish between *local memory* and *local variables*. Local memory exists in the GPU's DRAM and, like global memory, suffers approximately a 150x latency penalty over register or shared memory access. Local variables are simply variables which are local to a CUDA C function. The compiler will attempt to place local variables into the fast registers whenever possible, although overuse of local variables will cause the compiler to spill to the slower local memory.

We were able to ascertain the number of references to local and global memory and the number of operations based on the output of the *decuda*, a third party tool to disassemble the binary produced by nVidia's *nvcc* compiler. This provides a static analysis and does not account for runtime behavior such as loops or conditionals. Optimization 2a (copying global data to local variables) adds a loop which the compiler does not unroll with the Flaim model. Manual inspection

Table 4.2: The impact of optimizations on the running time of the ODE solver for the Flaim model. Times measure the running time to simulate 20ms of a heart beat for a mesh with 42,240 collocation points. The first two entries in the table were run on the CPU and the remaining entries on the GPU. Optimizations were applied cumulatively, in the order listed in curly braces. *CPUx* and *GPUx* are the speedups achieved over the reference implementation. *Ops* are the number of ptx assembly operations for the kernel as revealed by *decuda*. The *Gbl* column contains the number of times global memory is referenced in the ptx assembly and the *Loc* column contains the number of times local memory was referenced. The *Tot* column contains the sum both global and local memory references.

Method	Time(s)	CPUx	GPUx	Ops	Gbl	Loc	Tot
CPU:serial	6754.06						
CPU:OpenMP	1068.82	1.0					
Naïve GPU	52.69	20.28		19747	1415	68	1483
Opt{1}	31.46	33.97	1.0	6685	1009	214	1223
Opt{1,2a}	16.44	65.03	1.91	5317	178	304	700
Opt{1,2a,3}	41.47	25.77	0.76	7557	614	743	1357
Opt{1,2a,4}	8.12	131.58	3.87	5317	178	304	700
Opt{1,2b}	25.86	41.34	1.22	6493	820	179	999
Opt{1,2c}	18.27	58.49	1.72	6039	278	264	542
Opt{1,3}	26.11	40.94	1.21	8205	1058	0	1058
Opt{1,2c,3}	13.81	77.38	2.28	7557	331	0	331
Opt{1,2c,3,4}	7.94	134.65	3.96	7557	331	0	331

of the generated assembly indicates that to account for this loop we needed to add 86 global memory references, 86 local references and 602 (86×7) operations to the static count of the ptx assembly generated by *decuda*, which we have included in Table 4.2.

The running time of the naïve GPU implementation was 52.69 seconds to simulate 20.0 milliseconds of a heartbeat. By applying various optimizations, we were able to improve performance by a factor of 6.6, or 134.6 times faster than the OpenMP implementation running on a quad core i7 processor and 850.5 times faster than a single core. We ran with 64 threads per block and were able to maintain a device occupancy of 5 concurrent thread blocks.

When comparing the speedups of the various GPU optimizations, we do not include the redundancy expression removal (Optimization 1). As mentioned previously, the naïvely generated code is bulky, and it isn't known to what extent the programmer could remove redundant expressions by hand. Thus, when we use Optimization 1 as our base GPU implementation, our GPU speedup was 3.9 rather than 6.6.

decuda reveals that Optimization 1 reduces the number of assembly instructions by a factor of 3.0 and global and local memory references decreased by a factor of 1.2. The actual running time drops by a factor of 1.7 indicating that, while the running time is influenced by the number of instructions, the global and local memory references have a larger impact in the performance - we will quantify this relationship in greater detail shortly. As a result of this improvement we always applied Optimization 1 regardless of what other optimizations were applied subsequently.

The three caching optimizations 2a, 2b, and 2c resulted in improvements of 1.91x, 1.22x, and 1.72x respectively. Prior to Optimization 3, Optimization 2a (copying global variables into local automatic storage) was the most effective. Although the ptx assembly generated by *decuda* reveals that this data is simply copied into local memory from global memory, the compiler is able to effectively place the local data into registers when they are accessed frequently. Further, we found that number of instructions referencing the global memory dropped from

1009 to 396 references while the local memory references increased only from 214 to 304. These results indicate that the compiler can do a very good job of managing registers and local memory when it is forced to spill. Because the compiler is capable of handling registers and local memory so effectively, we suspect that if it could also spill to shared memory (as we have done manually with code generation), it would generate even more efficient code.

Inspection of the ptx assembly revealed that Optimization 2a also results in a decrease in total operations performed. Although the exact behavior of the compiler is proprietary, we speculate that this is because data in global variables is more difficult to reference than data in registers. For example,"y_global[12 * num_gauss_pts + idx]" becomes simply "y_reg[12]". This adds additional instructions for each global memory reference. We also suspect that the compiler has additional optimization opportunities when dealing with local variables as opposed to global data. For example, more aggressive common-subexpression elimination (CSE), etc., might be possible on local data and the compiler knows it is *safe* to assume that local data stored in a register hasn't changed from one instruction to another and thus does not need to be refetched from global memory each time it is used.

Unlike 2a, Optimizations 2b and 2c provide a cache for global memory, using statically determined replacement (at translation time) without adding any conditionals to the generated code. The cache lives in the multiprocessor's shared memory. Because we do not need to use shared memory for interthread communication, and since shared memory is available almost as fast as registers, it is ideally suited for use as a global memory cache. However, these caching strategies incur a penalty over Optimization 2a. For example, *decuda* tells us that Optimization 2c results in approximately 13.5% more instructions executed with Optimization 2a.

Because Optimization 2b merely caches the most frequently used global variables in shared data, the technique, while beneficial, is not very effective with large kernels with many global memory references. Optimization 2c, using Belady's MIN page replacement policy, always outperformed this simpler caching mechanism.

We also see that 2c references more global variables than 2a, by a factor of 1.5, due to the finite size of the cache (11 words). Indeed 2c's 278 global variables references can be accounted for as follows. In our C code there are 187 cache misses for loading global data, 87 references for storing global data, 5 variable "parameters" which are stored in global memory and not cached, and 1 additional global store for handling the numerical singularity (see Section 3.2.1). This results in 280 variables total, so 2 references were apparently eliminated by the compiler. Theoretically, a large shared memory cache would yield better performance indicated by the results in Table 4.3. Although we could not test these various cache sizes due to the limits of shared memory size in hardware, we can simulate the potential hit rates which would be obtained by other cache sizes. By the time a cache size of 64 words is used, the maximum hit rate is obtained. We selected a cache size of 11 words, because it is the largest size we could use and fit 5 thread blocks of 64 threads at a time and leave some space for passing parameters which also relies on shared memory (16 KB / (4 bytes $\times$ 5 blocks $\times$ 64 threads)). Indeed we observed that increasing the cache size beyond 11 resulted in a performance penalty.

Table 4.3: Cache hit rates for Flaim model for different size shared memory caches using optimizations 1,2c. and 1,2c,3

Cache Size	Hit Rate (1,2c)	Hit Rate (1,2c,3)
1	47.76%	46.70%
2	61.59%	60.09%
4	73.21%	71.51%
8	84.57%	82.30%
9	85.70%	83.38%
10	86.71%	84.20%
11	87.51%	84.84%
12	88.18%	85.34%
16	89.91%	86.04%
24	92.05%	86.23%
32	93.45%	86.23%
64	94.19%	86.23%
100	94.19%	86.23%

The best running time resulted from combining Optimizations 1, 2c and 3 (kernel partitioning). Optimization 3 is much more effective when combined with

2c: performance nearly doubles. Without 2c (the shared memory cache), Optimization 3 impairs performance by a factor of 1.9. Although the optimization effectively eliminates all local memory references, it introduces additional references to global storage. This is a surface to volume effect; splitting kernels corresponds to partitioning the underlying dependence graph into subgraphs. Since kernels run to completion, they must communicate via global memory or redundantly recalculate intermediate variables common to multiple kernels. Caching reduces the impact and nearly doubled performance.

To better understand why kernel partitioning is effective when combined with a shared memory cache, it is important to consider the effect of these optimizations on register pressure. As indicated in Table 4.4, after applying kernel splitting, the largest kernel referenced 67 different global variables and 128 different local variables as compared to 175 global variables and 267 local variables without the partitioning. With less than half the variables to manage, register pressure is less sever allowing the compiler to store local variables in *registers* without spilling them to *local memory*.

Table 4.4: The number of DIFFERENT global and local variable names that were referenced after each optimization. When optimizations include kernel splitting (i.e. Optimization 3) the maximum number of variables referenced in any of the kernels is used.

opt	globals	locals
1,2a	175	354
1,2c	175	267
1,3	67	128
1,2c,3	67	128

In Table 4.5 we see that Optimization 2a the CUDA code referenced global variables 175 times, yet referenced its 354 local variables 1585 times. Although the compiler clearly manages these 1585 references effectively by using registers when possible, it is unable to eliminate 304 local memory references due to the register pressure demand. By eliminating local memory spilling with kernel partitioning, the only remaining access to the GPU's DRAM are global memory references - exactly what our caching scheme can significantly reduce.

Table 4.5: For optimization 2a, the number of references of the form "y_global [var * num_gauss_pts + idx]" and "y_reg[var]".

y_global	y_reg
175	1585

These results indicate that at least three factors were making large contributions to our observed runtime: the number of operations, the number of global references, and the number of local references. A simple equation might be:

$$t \approx operations \times t_{op} + globals \times t_{gbl} + locals \times t_{loc} \tag{4.1}$$

Where t is the observed runtime, *operations* is the number of operations, *globals* is the number of global variables, *locals* is the number of local variables, t_{op} is the time per operation, t_{gbl} is the time per global reference, and t_{loc} is the time per local reference. A least squares fit of our results obtained from the optimization sets {1}, {1,2a}, {1,2b}, {1,2c}, and {1,2c,3} indicated that $t_{gbl} \approx 0.026, t_{loc} \approx 0.0186, t_{op} \approx 0.001$. Thus, global and local memory access required approximately 20x more time than other operations - consistent with the latency results Volkov and Demmel describe in [54]. Volkov and Demmel also describe an 3-7 μs overhead for launching kernels. Although for Optimization 3 this might account for up to .35s of the runtime ($\frac{20.0s}{0.0016s}$ $\times$ 6 kernels $\times$ 7 μs), it does not significantly alter our analysis. Also, since local memory references always coalesce (see the CUDA Guide 2.1 section 5.1.2.2 [36]) and local and global memory required roughly the same time, our efforts to align global memory references to coalesce appears to have been successful. The application of the fitted values for t_{loc}, t_{gbl}, t_{op} to our remaining optimization sets yielded predicted runtimes within a few precent of the actual runtime.

We also attempted to apply Optimization 3 (function partitioning) to our CPU implementation. We found that this led to a performance degradation. We suspect that there are several factors that may have contributed to the performance penalty. First, because this optimization creates additional functions, it adds overhead associated with each additional function call. Also, when we break up a function into smaller pieces the compiler is likely less capable of performing

certain optimizations (such common subexpression elimination, etc.) across function calls which may add additional inefficiencies. Our main motivation in splitting functions was to relieve the compiler of the register pressure - it's not clear that the CPU has the same restrictions. When the CPU, for example, is forced to spill to its DRAM, it still has L1 and L2 caches to fall back on. These caches are not available on the GPU. In other words, while the GPU pays a 150x latency penalty for running of of registers, the CPUs penalty is far less. Finally, the size of our partitioned kernels were carefully tuned for the GPU. We partitioned a kernel just before it grew too complex for the compiler to manage and started spilling to local memory. There is no reason to assume that this same kernel size would be optimum for our CPU, so a sweep of different function sizes for the CPU might lead to some improvement. This optimization remains as future work.

As a final optimization, we use both GPUs present in our GTX 295 (Opt4). As mentioned previously, the two GPUs must be managed by separate CPU threads, even though both GPUs are on the same graphics card. Working together, the GPUs are able to achieve an additional 1.7x speedup over our fastest implementation, consistent with 67% speedup Volkov and Demmel achieved with 2 GPUs in [54]. Figure 4.3 summarizes the impact of the various optimizations on the observed runtime.

In order to better understand how the GPU was spending its time, we employed the CUDA profiler (see Figure 4.4). This helps demonstrate why the GPU has been so successful at reducing the ODE runtime - very little time is spent shuttling data back and forth between the CPU and the GPU. Instead the vast majority of time is spent exactly where we want it: in the GPU compute kernels. We were able to achieve this efficiency by reducing the amount of data updated at each PDE time step. Of the 87 state variables updated at each time step, only one variable (the voltage) must be sent from the host to device at each time step while only an updated voltage and its derivative must be returned to the host. Thus, only two words of data for each collocation point are returned from the device to the host.

In Table 4.6 we see the results for the remaining cell models. For these

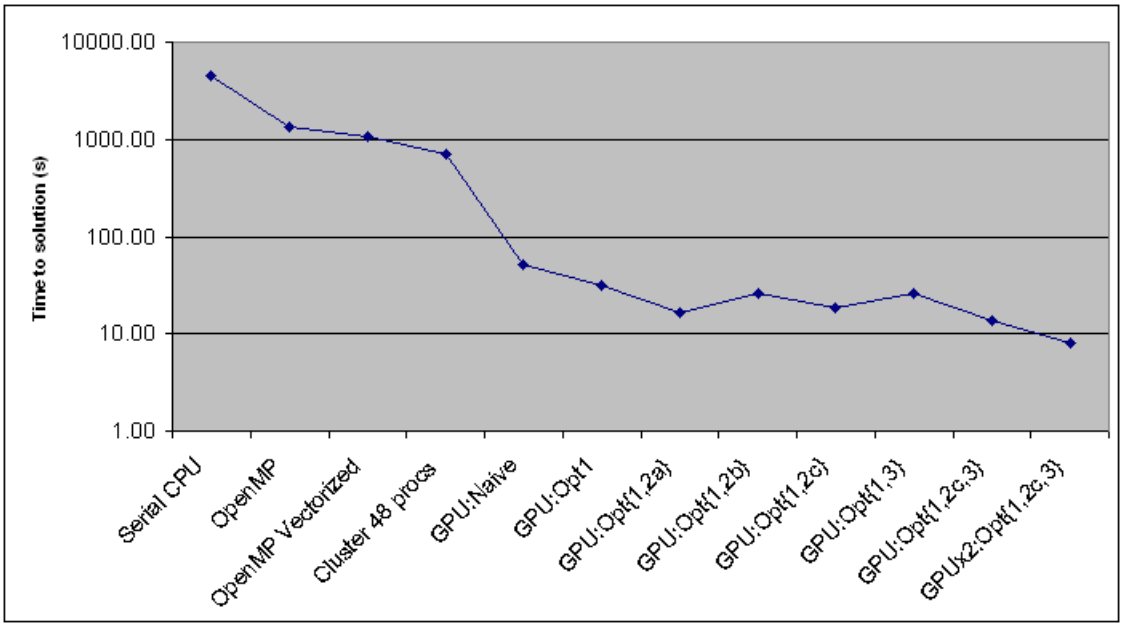

Figure 4.3: Summary of performance showing the impact of optimizations

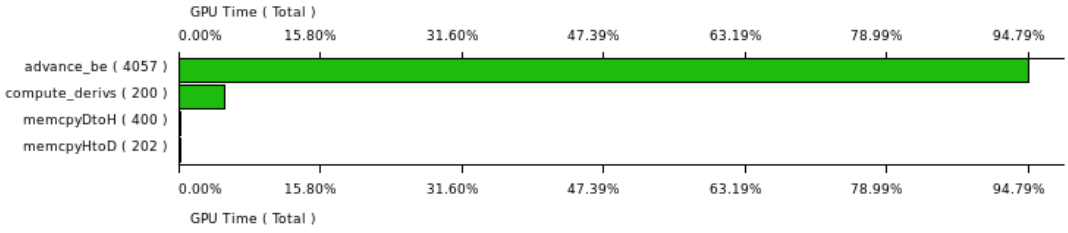

Figure 4.4: Profile of the time spent on the GPU with the Flaim model. *advance_be* is the time spent calculating the next time step of the cell model using the backwards Euler method. *compute_derivs* simply computes the derivatives for each state variable prior to calculating the next PDE time step. The *memcpy* functions transfer data between the CPU and GPU and occur infrequently relative to the time spent with other computations.

models, the situation is a bit different than the more complex Flaim model - we were able to fit all the state variables and intermediate local variables into registers without spilling to local memory. Thus, the need for kernel partitioning (Optimization 3) was obviated by the reduction in register pressure demand.

For MFHN and Beeler-Reuter, our 11 variables shared memory cache was more than enough space to hold all the state variables without any replacement. Optimization 2c was therefore able to perform almost as well as Optimization 2a which relies on only registers to cache the global data. That the register file achieves better performance than shared memory does not come as a surprise and is consistent with Volkov and Demmels findings [54]. With Puglisi, although all 18 state variables could not fit in our 11 variable shared memory cache, performance was only 12% slower than copying to registers.

In each of these cases, caching to shared memory or registers was better than directly referencing global memory, although simply copying to registers (2a) was the most effective approach.

However, as previously mentioned, recent trends in cell model formulation suggest that even more state variables are likely to be needed, requiring more creative optimizations, such as those we have suggested with the Flaim model, for best performance.

Finally, combining 2a with 4 (dual GPUs) added a 40-80% improvement consistent with the speedup observed with the Flaim model.

It could be argued that our assessment of performance only demonstrates improvements against our own code base. However, the original code, part of *Continuity 6*, is a publicly available resource, with years of experience by hundreds of users and continual enhancement by dedicated programmer support. We have been using automated translation to generate single core python+Fortran code for years. Although hand coded simulators may be faster than automatically generated ones, we feel that the required effort is unjustified. Our Flaim model starts with 525 lines of high level equations, and results in 2500 lines of CUDA C. Coding this by hand would be impractical for the domain scientist, and there are several other cell models of interest by our diverse user community. The dramatic speedups we

Table 4.6: The impact of optimizations on the running time of the ODE solver for the modified FitzHugh-Nagumo, Beeler-Reuter, and Puglisi model. Times measure the running time to simulate 20ms of a heart beat for a mesh with 42,240 collocation points. Optimizations were applied cumulatively, in the order listed in curly braces. *CPUx* and *GPUx* are the speedups achieved over the reference implementation. *Ops* are the number of ptx assembly operations for the kernel as revealed by *decuda*. *Gbl* are the number of times global memory is referenced in the ptx assembly.

Method	Time(s)	CPUx	GPUx	Ops	Gbl
MFHN					
CPU:serial	37.79				
CPU:serial:SSE	37.67				
CPU:OpenMP:SSE	7.34	1.0			
GPU:Naïve	0.96	7.68		116	19
GPU:Opt{1}	0.81	9.05	1.0	87	14
GPU:Opt{1,2a}	0.56	13.15	1.45	79	8
GPU:Opt{1,2a,4}	0.4	18.57	2.05	79	8
GPU:Opt{1,2b}	0.81	9.09	1.0	87	14
GPU:Opt{1,2c}	0.57	12.83	1.42	86	8
Beeler-Reuter					
CPU:serial	2326.73				
CPU:serial:SSE	2280.06				
CPU:OpenMP:SSE	419.29	1.0			
GPU:Naïve	5.22	80.26		780	71
GPU:Opt{1}	4.28	97.94	1.0	627	55
GPU:Opt{1,2a}	2.5	167.72	1.71	455	20
GPU:Opt{1,2a,4}	1.5	278.78	2.85	455	20
GPU:Opt{1,2b}	2.97	141.08	1.44	543	32
GPU:Opt{1,2c}	2.77	151.64	1.55	488	20
Puglisi					
CPU:serial	3386.05				
CPU:serial:SSE	3381.87				
CPU:OpenMP:SSE	622.4	1.0			
GPU:Naïve	19.21	32.4		4422	267
GPU:Opt{1}	8.6	72.4	1.0	1628	129
GPU:Opt{1,2a}	5.98	104.03	1.44	1471	49
GPU:Opt{1,2a,4}	3.31	188.15	2.6	1471	49
GPU:Opt{1,2b}	7.67	81.17	1.12	1663	93
GPU:Opt{1,2c}	6.69	93.08	1.29	1617	53

report have transformed the way we engage in scientific discovery with *Continuity*, and will soon be put into practice. A constant factor in performance increase (if there was one lurking) would not significantly impact the discovery process, though of course we are always looking to improve performance.

We also originally attempted to use a gigaflop rate to assess performance, however, as revealed by decuda, our application makes significant use of the special function unit (pow(), exp(), etc.), so a gigaflop rate is not meaningful.

4.3.3 Accuracy

As discussed previously in Section 1.3.2, there is a significant performance penalty for using double precision arithmetic[5]. This gives us an incentive to use single precision. It is important to clarify what this means in terms of accuracy for cellular modeling.

We use RRMS error, a standard measure used in the literature when discussing solvers. We computed solutions over a 250ms simulation, roughly the duration of an action potential in the Flaim model. For the Flaim model, the difference between double precision (DP) and single precision (SP) on the CPU was quite small, about 0.85%. The error between SP on the GPU and SP on CPU was even smaller: only 0.0018%.

A greater error was introduced by our solver selection. There was a 1.55% error when comparing DP radau5 with DP backwards Euler (both on CPU). However, to see if our errors were accumulating, we compared our SP backwards Euler GPU solver against the DP radau5 CPU solver and found that our RRMS error was less than 1.48%.

Of course, the more serious divide by zero errors caused by round-off errors in single precision were eliminated by our precision checker (see section 4.2).

We deem this error acceptable in light of the fact that much larger uncertainties are involved in measuring the empirical parameters used in the cell models. Since these parameter errors aren't large enough to be distinguished by experimentation (see [15]), the ones we observe certainly are not. Thus, our experience indicates that single precision arithmetic was sufficiently accurate for our

needs, although, this result is heavily application dependent[2].

4.4 Implications for Fermi

nVidia's next generation GPU architecture, Fermi, includes several innovations that will impact performance [37, 41]. These innovations include: main memory caches, a new instruction set, faster DRAM and more cores per Streaming Multiprocessor (SM). Although Fermi is not available at the time of writing, we are able to make some performance projections based what is known about the processor publicly.

We are interested in the impact of the new on-chip memory architecture. Traditionally, GPUs have avoided using space for caching global memory with the hope that the latency will be partially hidden through thread concurrency. However, in practice, even with many threads, there is still a need to reduce the latency penalty. Our experience has indicated that reducing traffic to the device's DRAM is critical to achieving high performance, which we addressed with the careful use of shared memory as a global memory cache. Indeed, Fermi's caches confirm the need to reduce the penalty to reference global memory.

Fermi has 16 streaming multiprocessors (SM), each with 32 cores that share 128KB of registers and a 64KB on-chip memory. All SMs share a 768 KB L2 cache. The on-chip memory is software configurable-it can be split into 48 KB cache and a 16 KB shared memory space, or the other way around. Thus, the amount of shared memory per core is the same as in the 200 series, but the number of registers per core has been cut in half. Presumably the L2 serves as a victim cache for L1 since it is the same size. With the GTX 200 series, each streaming multiprocessor (SM) has 8 cores, 16 KB of shared memory, and 64KB of registers.

As mentioned previously, after Opt1 has been applied, each CUDA kernel reads 88 different globals (it also writes 87, but there is no advantage to caching them). On the GTX-295, with 64 thread blocks and 4 active thread blocks, Opt{1+3+2c} enables the processor to store most of the required state

[2]Xing Cai, Simula Laboratory, Private Communication, 2009.

(90112 bytes/SM) on-chip (80KB). To maintain the same level of occupancy on Fermi, we require 16 active thread blocks for a total of 352KB. However, the loss of half the registers will lead to a dramatic increase in the number of slow device memory accesses. We have only 192KB of fast on-chip memory and are 50% short. By comparison, we were only 10% short on the GTX-295. (L2 is shared by all SMs, and cannot help us out of this dilemma.) This is where optimizations 2a,3, and perhaps 2c come in, as they are designed to help us utilize the registers and the remaining shared memory. Otherwise, we are actually short by 304KB.

Additionally, we simulated what kind of hit rate would be obtained with *least recently used* (LRU), *first in first out* (FIFO), and *random* cache replacement policies and both *direct mapped* and *fully associative* caches when compared to running on the GTX-295 with the same occupancy[3] in which each thread would have about room for about 8 cached values - 25% less space per thread than was possible with our shared memory cache in the 200 series GPU. We assume the cache size to be a single word - also unknown at the time of writing. In Table 4.7 we explore the impact these various cache policies might have on performance.

For our first set of simulations, we used Opt{1} as a starting point: 6685 operations, 214 local memory operations, and a single (i.e. non-partitioned) kernel. We applied our fitted weights 4.1for operations, global references (cache misses), and local references to obtain the estimated runtime which we report. We observed that a fully associative cache with a LRU replacement policy would yield the lowest miss rate on our data, and thus achieve the highest performance, although directly mapped cache and FIFO replacement policy trailed by less than 5%. We observed that a RANDOM replacement policy (not likely to be used by Fermi anyway) yielded the worst results.

Compared to the 33.46s we measured with Opt{1}, these projections indicate that the Fermi cache has the potential to reduce runtime to 18.6s, roughly a 1.8x speedup. It also has the potential to be competitive with our offline cache, such as Opt{1,2c} which required 18.27s. The simulated cache is competitive

[3]For a more direct comparison with the GTX 200 series, we assumed that the registers available per thread were the same in our cache simulations, although Fermi will actually include half the registers.

due to the overhead introduced by maintaining our own caching. Local memory references increased from 214 to 264 with a shared memory cache as discussed previously and displayed in Table 4.2. We suspect that this local memory overhead is unnecessary, as it should be no more difficult to reference shared memory than global memory. And perhaps with better compiler support this overhead might be eliminated completely.

Table 4.7: Cache simulator results. Cache replacement policy found in the *replacement* column, Cache Associativity (e.g. fully associative or directly mapped) found in the *Assoc.* column, and estimated runtime found in the *ETA* column. These results are for a single kernel (Optimization 3 has not been applied).

Policy	Assoc.	Hits	Misses	Miss Rate	ETA (s)
LRU	FULL	864	307	26.22%	18.65
FIFO	FULL	829	342	29.21%	19.56
RANDOM	FULL	765	406	34.67%	21.22
LRU	DIRECT	831	340	29.04%	19.51

Alternatively, register pressure can be reduced and local memory spilling can be eliminated by kernel partitioning, as previously discussed. Applying our cache simulator to a kernel split into several pieces, such as with Opt{1,3}, the estimated runtime was reduced to 17.44s (see Table 4.8 for various cache configurations). Thus, we anticipate our shared memory cache, which required 13.81s, to be at least 20% faster than Fermi's cache. In our simulation, we assumed the L1 cache to be maintained across kernel calls, which may not be the case.

Table 4.8: Cache simulator results. Cache replacement policy found in the *replacement* column, Cache Associativity (e.g. fully associative or directly mapped) found in the *Assoc.* column, and estimated runtime found in the *ETA* column. These results are for a partitioned kernel (Optimization 3 has been applied).

Policy	Assoc.	Hits	Misses	Miss Rate	ETA (s)
LRU	FULL	894	355	28.42%	17.44
FIFO	FULL	856	393	31.47%	18.42
RANDOM	FULL	772	477	38.19%	20.61
LRU	DIRECT	854	395	31.63%	18.48

Alternatively, the 48 KB could be allocated for shared memory instead of as an L1 cache. In this case, the shared memory available has increased by a

factor of 3 while the processor resources have increased by a factor of 4, effectively decreasing the shared memory available per thread by 25%. Each thread will therefore only be able to cache 8 or 9 single precision floating point values if the same device occupancy is to be maintained.

Decreasing the shared memory cache size to 8 single precision floats from 11 will increase the global variable references from 278 to 322 and increase local memory references from 264 to 290 and operations from 6039 to 6081 for Opt{1,2c}. Based on our fitted weights, this might increase the run time from 18.3s to 19.6s or about an 8% penalty. Observed performance with our GTX 200 series card while using 25% less shared memory was 19.9s - about a 9% penalty.

Our conclusion is that the partitioned cache will provide no advantage compared with our far less costly approach based on source-to-source translation. Although Fermi's cache may be easier for some programmers to work with, we suspect that it may actually yield a performance penalty over simply using all the space for a programmer (or compiler) controlled shared memory cache. Further, we suspect that there is a missed opportunity to take advantage of offline clairvoyant algorithms, such as our use of Belady's MIN.

Because CUDA uses a SIMT programming model, performance conscious software developers are already accustomed to writing code which minimizes or eliminates branching. And such codes are ideally suited for offline analysis and programmer controlled caches which do not need to troubled by unpredictable runtime behavior. Even if a programmer controlled cache performed roughly the same as a hardware cache, the software solution is simpler because it can be tuned to the application, as we have done. Other applications may benefit from the technique, which don't do much branching (after all that's the expected case).

Further, nVidia's scheme is not very flexible and extremely coarse grained. The advantage of our approach is its flexibility. We create exactly as much cache as there is space for, no more and no less, and therefore have greater control over the distribution of shared and cached state. Perhaps Fermi's cache will help speed these up. it is straight forward to cache global memory accesses (reads) using static analysis memories aren't likely to be as useful as they have been with traditional

CPUs.

4.5 Complete Electrophsyiology Simulation

After optimizing the ODEs with CUDA, the time spent calculating the PDEs become more significant. The PDE calculation involves solving a sparse system of linear equations. Such systems can either be solved *directly* or *iteratively*. We used SuperLU[11], which employs a direct method.

With a direct method, there are two phases: *factorization* and series of *back substitutions* which are performed at each PDE time step. With electrophysiology simulations, the factorization must be performed only once while back substitutions occur at each PDE time step. However, we give special attention to the time spent factorizing because, although our electrophysiology simulations require only a single factorization, it can be a very computational intensive operation.

In our initial simulations, we employed the serial implementation of SuperLU which required 59.95s for factorization and 1590.78s all 3000 back-subsititions (e.g. 300ms at 0.1ms time steps). We found that both factorization and back substitution phases benefited from the use of distributed SuperLU[29] (a parallel implementation) by a factor of 3.45x and 2.52x respectively reducing computation time to 17.38s and 632.63s on a quad-core i7 processor or 650.0s total. When we compare this the 119.3s spent solving the Flaim model on the GPU, we see that the PDEs are still the bottleneck.

We now summarize the speedups obtained for the entire electrophysiology simulation for various cell models. In our original serial CPU implementation with the Flaim cell model, the 300ms simulation required 103,530.7s total - almost 29 hours; with 101,889s (98.4%) spent solving ODEs and 1,641.7s (1.6%) solving PDEs. By optimizing the ODE solve with OpenMP we were able to reduce the ODE time to 16,015.8s. Our best dual GPU implementation reduced the time spent solving ODEs to 119.29s. And by solving the PDEs with distributed SuperLU the PDE time is reduced to 650.0s or 84.5% of the 759.3s total. In other words, the entire Electrophysiology simulation was sped up by 136.3x from the original serial

implementation or 21.2x over our best multicore CPU implementation.

Future work will involve optimizing PDEs on the GPU as well. Because solving sparse linear systems is such a common problem applicable to many domains, several projects are already underway to explore this possibility, including a solver being developed by nVidia[4]. A sparse matrix multiplication routine on CUDA achieved an average of 10.85x speedup over a CPU implementation [5]. If this same speedup could be achieved for the factorization and solve, we speculate that the PDEs could be solved in 65s, reducing the total run time to 184s and once again making the ODE solve the computational bottleneck and responsible for about 65% of the running time.

For the remaining cell models, the PDE times remain unchanged; only the ODEs time vary. See Table 4.9 for a summary of the best ODE times.

Table 4.9: A summary of the time spent computing ODEs for 300ms for a mesh with 42,240 collocation points. Because MFHN, BR, and Puglisi are relatively small and do not spill registers to local memory, our most aggressive optimizations (shared memory cache and kernel splitting) were unnecessary.

Model	Method	Time (s)	Ops	% of Sim
MFHN	GPU:Opt{1,2a,4}	5.91	79	.9%
BR	GPU:Opt{1,2a,4}	22.64	455	3.3%
Puglisi	GPU:Opt{1,2a,4}	49.87	1471	7.1%
Flaim	GPU:Opt{1,2c,3,4}	119.29	7557	15.5%

[4]Nathan Bell, nVidia, Private Communication, 2009.

Chapter 5

Contribution and Related work

Electrophysiology modeling differs from other time dependent problems (such as the classic N-body problem) in that the cost of solving the Partial Differential Equation (PDE) is relatively inexpensive compared with that of solving the resultant systems of Ordinary Differential Equations (ODEs). Recently, the GPU has been employed in electrophysiology modeling [48] with 8 state variables, a simple model by modern standards. The forward Euler integration method was used to solve the ODEs, which is sufficient for non-stiff ODE systems. However, forward Euler is not appropriate for the current generation of cell models with very stiff ODEs. Lastly, there is no discussion about the impact of single precision on accuracy. In our case, accuracy is a significant concern, especially when considering numerical singularities present in our complex cell models.

Constructing a well tuned application in CUDA can be a challenge, owing complex behavior of interactions among a set of optimizations, and the uncertainty in modeling performance of the hardware and the *nvcc* compiler. Domain scientists prefer to remain aloof of the process. Liu et al. [32] implemented a source-to-source translator to automatically optimize a kernel against program inputs, and they use statistical learning techniques to search the optimization space obtained by running the application repeatedly for different inputs and optimization parameters. They also demonstrate that for different inputs different optimizations yield the best results, which we observed from experimenting with the Flaim, Beeler-Reuter, and other cell models. Our optimization search space was generally small enough

as to remain tractable, but clearly an automatic approach might facilitate the identification of the most beneficial optimizations.

Silberstein et al. [50] demonstrate a *user-managed cache* which, like our cache, resides in shared memory. For their application, caching to shared memory was more dramatic - up to a factor of 20 - which likely indicates that their kernels were far more memory bound than any which we encountered in our domain. Also, the cache replacement policy described by Silberstein et al. is implemented in software and thus cannot make use of Belady's MIN algorithm which generates the optimum replacement policy offline. They also do not explore the possibility simply copying global data to local registers which may have yielded similar performance improvements.

The CELL Broadband Engine (CELL) [26] includes a Power Processing Element (PPE) similar to a traditional single core CPU, and eight Synergistic processing elements (SPE). Cell's 8 SPE are designed to enable mulithreaded programming, but with far fewer-coarser grained threads than those available on an nVidia GPU. The SPEs are also designed for single instruction multiple data (SIMD) use, analogous to the single instruction multiple thread (SIMT) programming model of the GPU. Cell uses Direct Memory Access (DMA) transfers to 256KB local memory on the SPEs. Although each thread has access to a 256KB local store whereas dozens of CUDA threads share just 16KB, this 256KB resource is still a limited resource which must be managed careful to achieve high performance. In [12] Eichenberger et al. discuss techniques to provide source-to-source translation to optimize scientific codes on the CELL through function partitioning, code generation, etc. This work involves abstracting the hardware specific details from the user to enhance programability without sacrificing all the hardware specific benefits available to a more sophisticated CELL programmer. For example, their compiler uses automatic code partitioning in order to fit code and data into the SPE that might not otherwise be possible. We also use partitioning to improve the use of very limited resources (registers), although we do so by breaking up very large functions into smaller pieces and use global memory to retain information between kernel calls. In other words, granularity is the key difference between

our partitioner and IBMs. Another comparison can be made with the use of a compiler controlled software cache which identifies data references not optimized using DMA transfers and caches them with a 4 way associative cache. We are also interested in caching data, although in our case it is from the global device memory which is available at a much slower rate than shared memory and registers.

An alternative approach to optimizing cell models is to simplify the model itself. This idea is explored in [20] which uses a 2 ODE system rather than a more detailed ionic model characteristic of recent developments in the field. Our experience with FitzHugh-Nagumo indicates that even simpler cell models are also amenable to GPU acceleration for additional speedups, although the speedups achieved may not be as significant as those we observed for the more complex cell models (see Table 4.6).

We have developed a system by which high level cellular models can be authored by a domain scientist and automatically translated into an optimized GPU kernel which incorporates expert GPU knowledge. In doing so, we have suggested a simple scheme for solving stiff systems of ordinary differential equations which yields excellent performance and reasonable accuracy on the GPU. We have also demonstrated a technique to remove numerical singularities common to cellular models which allows us to use single precision arithmetic for improved performance. We have also contributed a variety of techniques for optimizing very large, highly complex CUDA kernels. Our offline automatic caching scheme based on Belady's MIN page swapping algorithm yielded a 1.7x speedup over referencing global memory. We also developed an automatic kernel partitioning scheme to reduce register pressure and eliminate spilling to local memory yielding an additional 1.3x speedup. And by making use of a multi-gpu implementation we achieved an additional 1.7x speedup.

In summary, we have sped up the ODE solve by 4x over our base GPU implementation, 134x over a multicore i7 CPU implementation, 850x over a serial implementation, and 111x over a 48 core MPI cluster implementation. Indeed our implementation is fast enough that it would be feasible to use in clinical setting for patient specific modeling without the use of a cluster.

Chapter 6

Discussion and Conclusion

6.1 Applicability to other domains

Our approach to automatic optimization should have a broad appeal for other domains. In particular, we have demonstrated that when faced with a highly complex kernel, as exemplified by the Flaim model, the use kernel partitioning and an automatic shared memory cache can lead to significant performance improvements. Additionally, the use of a compile time automatic caching should be better exploited by the compiler. Since CUDA kernels are often designed as non-branching (as is typically required for maximum efficiency) our use of Belady's MIN algorithm may be ideally suited to the CUDA architecture. Further, we would hope that a future version of the nVidia compiler should be able to automatically use shared memory as a compile time cache to reduce spilling to global memory. Forcing the user to manage this cache creates an additional burden on the application writer which could be alleviated with compiler support. Perhaps to address this concern, Fermi [37] will indeed include a hardware cache for global memory.

6.2 Suggestions for GPU hardware developers

6.2.1 Increase transparency

Perhaps one of the most challenging aspects of fine tuning CUDA kernels for performance is the lack of transparency with automatic compiler optimizations. Frequently, a developer attempts a novel, or even a well-known optimization, only to find that the change unexpectedly hurts performance or makes absolutely no difference. Most developers resort to a "trial and error" strategy in which the effectiveness of optimizations can only be determined by measuring actual performance. We recommend that GPU compiler authors consider making these automatic compiler optimizations more transparent to facilitate performance tuning.

Currently, the compiler can be configured to output an intermediate "ptx" assembly file and some general diagnostics about register, local, and shared memory usage. However, at a later stage in the compilation process, this intermediate ptx file is assembled to binary while performing additional optimizations completely hidden from the developer. In order to gain some insight into this last phase of compilation, a developer must resort to using a third party disassembler (*decuda*) and wade though potentially thousands of lines of uncommented assembly. We suggest the GPU chipset developers open up this currently closed proprietary information so that a developers are better able to understand what optimizations will work in certain circumstances. Additionally, we recommend that additional output be made available to the user in terms of global memory usage, estimated frequency of device memory usage, etc. Information from the CUDA occupancy calculator could also be incorporated into the compiler to provide suggestions for alternative configurations that should improve performance, such as reducing register pressure.

6.2.2 Automatic Performance Tuning

In additional to improving transparency, GPU chipset developers might consider incorporating the large body of techniques for *automatic performance tuning* into the compilers themselves. As discussed in Section 5, Liu et al. [32]

implemented a source-to-source translator for CUDA to automatically optimize a kernel against program inputs, and they use statistical learning techniques to search the optimization space obtained by running the application repeatedly for different inputs and optimization parameters. Automatic performance tuning has been used for years to optimize CPU code as well [56, 10], and frequently use *Automated Empirical Optimization of Software* (AEOS) to determine the best optimizations based on empirically measured timings. For example, Automatically Tuned Linear Algebra Software (ATLAS)[42] uses such timings to automatically identify the best optimizations for a linear algebra solver on a particular target architecture. With the rapidly changing landscape of GPU architectures these techniques might provide a mechanism for automatically tuning GPU kernels to keep pace with the changing architectures. Another notable example is the work done by the Berkeley Benchmarking and Optimization Group (BePop)[1] such as the Optimized Sparse Kernel Interface (oski)[55], which includes both offline and online tools to build matrix solvers well suited for a given platform. These empirical techniques may be applicable to the GPU and compiler designers might consider adding facilitates to automatically discover the *best* optimizations to automatically apply for a given compute kernel.

6.2.3 Spill to shared memory

Our experience indicates that shared memory can provide an excellent cache for global memory. However, simply allowing the compiler to spill to local memory (effectively using registers as a device memory cache) was also effective, even more so in many situations. We suggest compiler support that would combine both strategies automatically. If a kernel is not using shared memory for communicating between CUDA cores, the compiler should be able to automatically use shared memory as an additional register resource. This could reduce register pressure, obviate the need to spill to local memory, etc., and improve performance. This would cost nothing in terms of hardware - it would just require additional compiler support.

[1] http://bebop.cs.berkeley.edu/

6.2.4 Exploit Belady's algorithm

As we have also demonstrated, CUDA can be an ideal platform for Belady's optimum page swapping algorithm. Since some kernels involve non-branching code (or nearly non-branching in our case) the compiler may be able to determine caching strategies at compile time more effectively than a general purpose compiler which can make no such assumptions.

6.2.5 Improve Multi-GPU support

Optimizing algorithms across multiple GPUs proved very challenging. Indeed, multiple CPU threads are required to control multiple GPUs which required additional synchronization primitives as well as experience with pthreads. We suggest improved compiler support to automatically and transparently make use of the multiple GPUs available on a system. Ideally the compiler should be able to handle a heterogenous collection of GPUs and perform some degree of load balancing so that a GPU with greater computation power is assigned a greater portion of the work. For example, if a system has a GTX 120 with 32 cores and a GTX 285 with 240 cores, the compiler would ideally be be able to allocate an appropriate amount of work for each system.

6.2.6 Split large kernels

Our most complex kernels with many equations and many variables proved very challenging for the CUDA compiler. After breaking up a kernel into smaller and more manageable pieces the compiler was able to do its job more effectively. Although we relied on the application specific knowledge that state variables could be calculated independently, it may be possible for the compiler to automatically partition kernels to reduce register pressure to improve performance.

6.3 Conclusion

We have demonstrated an effective technique that encapsulates expert knowledge in a translator, allowing the domain scientists to work at the level of the cell model, without becoming entangled in low level implementation details. We have shown that a very complex cell model with 87 state variables, and hundreds of equations evaluated inside of a backwards Euler scheme, is amenable to GPU computing. Further we have shown that by using the shared memory as an offline global memory cache and by partitioning complex kernels we can improve performance even more.

In addition to the obvious benefits of faster cardiac simulations (e.g. faster research, larger models, etc.) our implementation scheme enables our accelerated heart simulator to be a transformational tool in the clinical setting; it will be a key step to enabling the use of these models for patient-specific modeling and diagnosis. Simulations must be highly efficient in clinical applications because of the time constraints involved in diagnosis and treatment, and because the simulations typically must be run many times in parameter sweeps to be useful. There is also a greater case for dedicated desktop computing in the clinical setting for reliability and privacy and security versus cluster computing on a shared resource outside of the clinic.

Having reduced the ODE bottleneck in electrophysiology simulations, we find that new bottleneck becomes apparent: the PDEs. Previously we have mostly ignored the PDE performance as they had such a limited impact on the overall performance of the simulation. However, now that over 50% of the simulation time is spent working with PDEs, we are investigating the opportunity to executing the PDEs on the GPU as well. This amounts to solving a sparse system of linear equations on the GPU, which has been the subject of other work [49] While we currently use serial and distributed SuperLU sparse linear, direct solvers, we are also looking at iterative solvers.

We are also interested in further optimizing the ODE performance by using a cluster of GPU enabled nodes and by exploring the applicability of more exotic ODE solvers to the GPU. Finally, we plan to further optimize our GPU perfor-

mance by introducing additional locality optimizations into our code generator.

Appendix A

MFHN Example

We now take a more detailed look at the output generated by our translator. First, we'll look at a FitzHugh Nagumo model as it might be specified as input to the translator. This is the model that the domain scientist would author. By allowing the user to create the model as a high level description it relieves him of the burden of dealing with low level GPU implementation details which our translator provides.

```
1  stim_start , stim_dur , stim_mag = params
2  ug, vg = state_vars
3  vmax = cm = d = 1.0
4  vrest = 0.0
5  a = b = 0.130
6  c1 = 0.260
7  c2 = 0.10
8  stim_end = stim_start + stim_dur#!end time of stimulus
9  heavi = Heaviside(t-stim_start) * Heaviside(stim_end-t)
10 i_stim = stim_mag*heavi
11 ug_norm = (ug-vrest)/(vmax-vrest)
12 dug_dt = (ug_norm * (ug_norm - a) * (1.0 - ug_norm) * c1 - c2 * vg
       * ug_norm) * (vmax - vrest) + i_stim * (1.0 / cm)
13 dvg_dt = b*ug_norm - b*d*vg
```

Listing A.1: *sympy* modified FitzHugh Nagumo model

Using the *sympy* library, our translator creates a symbol for each equation, and generates corresponding C expressions. *sympy* also replaces any variables with constants if it a constant can be determined at compile time.

Our translator takes this C code and identifies the dependencies for each state variable. Combining the dependency information with the single iteration backwards Euler method, our translator generates the following CUDA kernel. This kernel corresponds to Optimization1 - any variables which were redundantly calculated have been removed.

```
1  __global__ void y_next(REAL t, REAL t_end, REAL *y_global, REAL *
       y_global_temp, REAL *rpar_global, int num_gauss_pts){
2    REAL bi,yi, bi_be, aj, het_gradient, ug_norm,dvg_dt, dug_dt,
         heavi,stim_end,i_stim, dt = t_end-t;
3    int i, j,idx = blockIdx.x * blockDim.x + threadIdx.x;
4    y_global[0*num_gauss_pts + idx] += delta;
5    stim_end = (rpar_global[0*num_gauss_pts + idx] + rpar_global[1*
         num_gauss_pts + idx]);
6    heavi = (HeavisideEq((t - rpar_global[0*num_gauss_pts + idx])) *
         HeavisideEq((stim_end - t)));
7    i_stim = (heavi * rpar_global[2*num_gauss_pts + idx]);
8    ug_norm = y_global[0*num_gauss_pts + idx];
9    dug_dt = ((i_stim - ((0.1 * ug_norm) * y_global[1*num_gauss_pts +
         idx])) - (((0.26 * ug_norm) * (1.0 - ug_norm)) * (0.13 -
         ug_norm)));
10   y_global[0*num_gauss_pts + idx] -= delta;
11   aj = dug_dt;
12   dug_dt = ((i_stim - ((0.1 * ug_norm) * y_global[1*num_gauss_pts +
         idx])) - (((0.26 * ug_norm) * (1.0 - ug_norm)) * (0.13 -
         ug_norm)));
13   bi_be = (aj - dug_dt)/delta;
14   y_global_temp[0*num_gauss_pts + idx] = y_global[0*num_gauss_pts +
         idx] - (-dug_dt*dt)/(1-dt*bi_be);
15   y_global[1*num_gauss_pts + idx] += delta;
16   ug_norm = y_global[0*num_gauss_pts + idx];
17   dvg_dt = ((0.013 * ug_norm) - (0.013 * y_global[1*num_gauss_pts +
         idx]));
18   y_global[1*num_gauss_pts + idx] -= delta;
19   aj = dvg_dt;
20   dvg_dt = ((0.013 * ug_norm) - (0.013 * y_global[1*num_gauss_pts +
         idx]));
21   bi_be = (aj - dvg_dt)/delta;
22   y_global_temp[1*num_gauss_pts + idx] = y_global[1*num_gauss_pts +
         idx] - (-dvg_dt*dt)/(1-dt*bi_be);
23 }
```

Listing A.2: CUDA source for MFHN model with many global memory references

Although this code will execute without trouble on the GPU (and at a tremendous speedup) it accesses the devices DRAM more frequently than necessary. To reduce the number of global memory references, we can store global memory in a shared memory cache. We use Belady's algorithm at translation time to decide which cache entries to evict when the shared memory is full. However, MFHN is so simple that no memory eviction is required.

```
1  __global__ void y_next_optimized(REAL t, REAL t_end, REAL *y_global
       , REAL *y_global_temp, REAL *rpar_global, int num_gauss_pts)
2  {
3    __shared__ REAL y_sm0[64], y_sm1[64];
4    REAL bi, yi, bi_be, aj, ug_norm,dvg_dt,dug_dt,heavi,stim_end,
         i_stim, dt = t_end-t;
5    int i, j,idx = blockIdx.x * blockDim.x + threadIdx.x;
6    y_sm0[threadIdx.x] = y_global[0*num_gauss_pts + idx];
7    y_sm0[threadIdx.x] += delta;
8    stim_end = (rpar_global[0*num_gauss_pts + idx] + rpar_global[1*
         num_gauss_pts + idx]);
9    heavi = (HeavisideEq((t - rpar_global[0*num_gauss_pts + idx])) *
         HeavisideEq((stim_end - t)));
10   i_stim = (heavi * rpar_global[2*num_gauss_pts + idx]);
11   ug_norm = y_sm0[threadIdx.x];
12   y_sm1[threadIdx.x] = y_global[1*num_gauss_pts + idx];//prepended
13   dug_dt = ((i_stim - ((0.1 * ug_norm) * y_sm1[threadIdx.x])) -
         (((0.26 * ug_norm) * (1.0 - ug_norm)) * (0.13 - ug_norm)));
14   y_sm0[threadIdx.x] -= delta;
15   aj = dug_dt;
16   dug_dt = ((i_stim - ((0.1 * ug_norm) * y_sm1[threadIdx.x])) -
         (((0.26 * ug_norm) * (1.0 - ug_norm)) * (0.13 - ug_norm)));
17   bi_be = (aj - dug_dt)/delta;
18   y_global_temp[0*num_gauss_pts + idx] = y_sm0[threadIdx.x] - (-
         dug_dt*dt)/(1-dt*bi_be);
19   y_sm1[threadIdx.x] += delta;
20   ug_norm = y_sm0[threadIdx.x];
21   dvg_dt = ((0.013 * ug_norm) - (0.013 * y_sm1[threadIdx.x]));
22   y_sm1[threadIdx.x] -= delta;
23   aj = dvg_dt;
24   dvg_dt = ((0.013 * ug_norm) - (0.013 * y_sm1[threadIdx.x]));
25   bi_be = (aj - dvg_dt)/delta;
26   y_global_temp[1*num_gauss_pts + idx] = y_sm1[threadIdx.x] - (-
         dvg_dt*dt)/(1-dt*bi_be);
27 }
```

Listing A.3: CUDA source for MFHN model using shared memory cache

Finally, we could split the kernel up into smaller pieces. Since MFHN is already very small and there is no risk of spilling registers to the device's DRAM, this optimization would provide no benefit and would actually result in a performance penalty due to additional global memory references. However, with more complex cell models, such as the Flaim, splitting up the kernel reduces register pressure and can result in a performance speedup.

```
1  __global__ void y_next_optimized_split1(REAL t, REAL t_end, REAL *
       y_global, REAL *y_global_temp, REAL *rpar_global, int
       num_gauss_pts)
2  {
3    __shared__ REAL y_sm0[64];  __shared__ REAL y_sm1[64];
         __shared__ REAL y_sm2[64];  __shared__ REAL y_sm3[64];
         __shared__ REAL y_sm4[64];  __shared__ REAL y_sm5[64];
         __shared__ REAL y_sm6[64];
4
5    REAL bi, yi, bi_be, aj,ug_norm,dvg_dt,dug_dt,heavi,stim_end,
         i_stim, dt = t_end-t;
6    int i, j,idx = blockIdx.x * blockDim.x + threadIdx.x;
7
8    //dug_dt
9    y_sm0[threadIdx.x] = y_global[0*num_gauss_pts + idx];
10   y_sm0[threadIdx.x] += delta;
11   stim_end = (rpar_global[0*num_gauss_pts + idx] + rpar_global[1*
         num_gauss_pts + idx]);
12   heavi = (HeavisideEq((t - rpar_global[0*num_gauss_pts + idx])) *
         HeavisideEq((stim_end - t)));
13   i_stim = (heavi * rpar_global[2*num_gauss_pts + idx]);
14   ug_norm = y_sm0[threadIdx.x];
15   y_sm1[threadIdx.x] = y_global[1*num_gauss_pts + idx];
16   dug_dt = ((i_stim - ((0.1 * ug_norm) * y_sm1[threadIdx.x])) -
         (((0.26 * ug_norm) * (1.0 - ug_norm)) * (0.13 - ug_norm)));
17   y_sm0[threadIdx.x] -= delta;
18   aj = dug_dt;
19   dug_dt = ((i_stim - ((0.1 * ug_norm) * y_sm1[threadIdx.x])) -
         (((0.26 * ug_norm) * (1.0 - ug_norm)) * (0.13 - ug_norm)));
20   bi_be = (aj - dug_dt)/delta;
21   y_global_temp[0*num_gauss_pts + idx] = y_sm0[threadIdx.x] - (-
         dug_dt*dt)/(1-dt*bi_be);
22 }
```

Listing A.4: CUDA source for MFHN model using shared memory cache split into 2 kernels (kernel 1)

```
1  __global__ void y_next_optimized_split2(REAL t, REAL t_end, REAL *
       y_global, REAL *y_global_temp, REAL *rpar_global, int
       num_gauss_pts)
2  {
3    //kernel 1 needs 2 vars
4    __shared__ REAL y_sm0[64];  __shared__ REAL y_sm1[64];
         __shared__ REAL y_sm2[64];  __shared__ REAL y_sm3[64];
         __shared__ REAL y_sm4[64];  __shared__ REAL y_sm5[64];
         __shared__ REAL y_sm6[64];
5
6    REAL bi;
7    REAL dt = t_end-t;
8    int i, j;
9    int idx = blockIdx.x * blockDim.x + threadIdx.x;
10
11   REAL yi, bi_be, aj, ug_norm, dvg_dt,dug_dt,heavi,stim_end,i_stim;
12
13   //dvg_dt
14   y_sm1[threadIdx.x] = y_global[1*num_gauss_pts + idx];
15   y_sm1[threadIdx.x] += delta;
16   y_sm0[threadIdx.x] = y_global[0*num_gauss_pts + idx];
17   ug_norm = y_sm0[threadIdx.x];
18   dvg_dt = ((0.013 * ug_norm) - (0.013 * y_sm1[threadIdx.x]));
19   y_sm1[threadIdx.x] -= delta;
20   aj = dvg_dt;
21   dvg_dt = ((0.013 * ug_norm) - (0.013 * y_sm1[threadIdx.x]));
22   bi_be = (aj - dvg_dt)/delta;
23   y_global_temp[1*num_gauss_pts + idx] = y_sm1[threadIdx.x] - (-
         dvg_dt*dt)/(1-dt*bi_be);
24 }
```

Listing A.5: CUDA source for MFHN model using shared memory cache split into 2 kernels (kernel 2)

Bibliography

[1] R. R. Aliev and A. V. Panfilov. A simple two-variable model of cardiac excitation. *Chaos, Solitons and fractals*, 7(3):293–301, 1996.

[2] Auckland. Overview - cellml. http://www.cellml.org/, 2009.

[3] G.W. Beeler and H. Reuter. Reconstruction of the action potential of ventricular myocardial fibres. *The Journal of physiology*, 268(1):177–210, 1977.

[4] L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. 2008.

[6] E. Bendersky. pycparser: http://code.google.com/p/pycparser/, 2009.

[7] S. G. Campbell, F. V. Lionetti, K. S. Campbell, A. D., and McCulloch. Coupling of adjacent tropomyosins enchances crossbridge-mediated cooperative activation in a markov model of the cardiac thin filament. *Biophysical Journal*. In Press.

[8] O. Certik. Sympy python library for symbolic mathematics. 2008.

[9] A. A. Cuellar, C. M. Lloyd, P. F. Nielsen, D. P. Bullivant, D. P. Nickerson, and P. J. Hunter. An overview of cellml 1.1, a biological model description language. *Simulation*, 79(12):740, 2003.

[10] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[11] J. W. Demmel, J. R. Gilbert, and X. S. Li. Superlu users guide. *University of California at Berkeley, Berkeley, CA*, 1997.

[12] A. E. Eichenberger, JK OBrien, KM OBrien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, and Z. Sura. Using advanced compiler

technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[13] F. H Fenton and E. M. Cherry. Models of cardiac cell. *Scholarpedia*, 3(8):1868, 2008.

[14] R. Fitzhugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961.

[15] S. N. Flaim, W. R. Giles, and A. D. McCulloch. Contributions of sustained ina and ikv43 to transmural heterogeneity of early repolarization and arrhythmogenesis in canine left ventricular myocytes. *American Journal of Physiology-Heart and Circulatory Physiology*, 291(6):H2617, 2006.

[16] E. Grandi, F. S. Pasqualini, J. L. Puglisi, and D. M. Bers. A novel computational model of the human ventricular action potential and ca transient. *Biophysical journal*, 96(3S1):664–665, 2009. Accepted.

[17] J. L. Greenstein and R. L. Winslow. An integrative model of the cardiac ventricular myocyte incorporating local control of ca2 release. *Biophysical journal*, 83(6):2918–2945, 2002.

[18] E. Hairer, S. P. Noersett, and G. Wanner. *Solving ordinary differential equations*. Springer, 1993.

[19] E. Hairer and G. Wanner. Solving ordinary differential equations ii. stiff and differential-algebraic problems. *Springer Series in Computational Mathematics*, 14, 1991.

[20] Monica Hanslien, Robert Artebrant, Aslak Tveito, Glenn Terje Lines, and Xing Cai. Stability of two time-integrators for the aliev-panfilov system. *journal for publication*, 2009.

[21] C.S. Henriquez. Simulating the electrical behavior of cardiac tissue using the bidomain model. *Critical Reviews in Biomedical Engineering*, 21(1):1–77, 1993.

[22] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–554, 1952.

[23] T. J. R. Hughes. *The finite element method*. Prentice-Hall Englewood Cliffs, NJ, 1987.

[24] L. A. Irvine, M. Saleet Jafri, and R. L. Winslow. Cardiac sodium channel markov model with temperature dependence and recovery from inactivation. *Biophysical journal*, 76(4):1868–1885, 1999.

[25] O. Skavhaug J. Sundnes, R. Artebrant and A. Tveito. A second order algorithm for solving dynamic cell membrane equations. *In Progress*, 2009.

[26] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[27] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001.

[28] R.C.P. Kerckhoffs, S.N. Healy, T.P. Usyk, and A.D. McCulloch. Computational methods for cardiac electromechanics. *Proceedings of the IEEE*, 94(4):769–783, 2006.

[29] X. S. Li and J. W. Demmel. Superlu dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, 2003.

[30] J. D. C. Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.

[31] Y. Liu, D. Maskell, and B. Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009. M3: 10.1186/1756-0500-2-73.

[32] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. volume 0, pages 1–10, 2009.

[33] R.J. Spiteri M.C. Maclachlan, J. Sundnes. A comparison of non-standard solvers for odes describing cellular reactions in the heart. *Computer Methods in Biomechanics and Biomedical Engineering*, 10(5), 2007.

[34] A. Michailova and A. McCulloch. Model study of atp and adp buffering, transport of ca2 and mg2 , and regulation of ion pumps in ventricular myocyte. *Biophysical journal*, 81(2):614–629, 2001.

[35] M. L. Neal and R. Kerckhoffs. Current progress in patient-specific modeling. *Briefings in Bioinformatics*, 2009.

[36] nVidia. Programming guide 2.1: http://developer.download.nvidia.com /compute/cuda/2_1/toolkit/docs/nvidia_cuda_programming_guide_2.1.pdf. *NVIDIA CUDA Programming Guide*, 2, 2008.

[37] nVidia. Nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[38] nVidia. Optimizing cuda. http://www.sdsc.edu/us/training/assets/docs/NVIDIA-04-OptimizingCUDA.pdf, 2009.

[39] World Health Organization. The top 10 causes of death. http://www.who.int/mediacentre/factsheets/fs310/en/index.html, 2004.

[40] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Citeseer, 2007.

[41] David Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf, 2009.

[42] A. Petitet, RC Whaley, and JJ Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1-2):3–25, 2001.

[43] J. L. Puglisi and D. M. Bers. Labheart: an interactive computer model of rabbit ventricular myocyte ion channels and ca transport. *American Journal of Physiology- Cell Physiology*, 281(6):2049–2060, 2001.

[44] J. M. Rogers, M. Courtemanche, and A. D. McCulloch. *Finite Element Methods for modeling impulse propagation in the heart*, chapter 7, pages 1–428. Computational Biology of the Heart. Sussex: John Wiley and Sons, Ltd., 1996.

[45] J.M. Rogers and A.D. McCulloch. A collocation-galerkin finite element model of cardiac action potential propagation. *IEEE Transactions on Biomedical Engineering*, 41(8):743–757, 1994.

[46] S. Rush and H. Larsen. A practical algorithm for solving dynamic membrane equations. *IEEE Transactions on Biomedical Engineering*, pages 389–392, 1978.

[47] S. Rush and H. Larsen. A practical algorithm for solving dynamic membrane equations. *IEEE Trans. Biomed. Eng.*, BME-25(4):389–392, 1978.

[48] D. Sato, Y. Xie, J. N. Weiss, Z. Qu, A. Garfinkel, and A. R. Sanderson. Acceleration of cardiac tissue simulation with graphic processing units. *Medical and Biological Engineering and Computing*, 47(9):1011–1015, 2009.

[49] O. Schenk, M. Christen, and H. Burkhart. Algorithmic performance studies on graphics processing units. *Journal of Parallel and Distributed Computing*, 68(10):1360–1369, 2008.

[50] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318. ACM New York, NY, USA, 2008.

[51] R.J. Spiteri and R.C. Dean. On the performance on an implicit-explicit runge-kutta method in models of cardiac electrical activity. *IEEE Transactions on Biomedical Engineering*, 55(5), 2008.

[52] T. P. Usyk and A. D. McCulloch. Computational methods for soft tissue biomechanics. *Biomechanics of Soft Tissue in Cardiovascular Systems*, 441:273342, 2003.

[53] W. J. van der Laan. Decuda and cudasm, the cubin utilities package: http://wiki.github.com/laanwj/decuda, 2009.

[54] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra.

[55] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, pages 521–530. Institute of Physics Publishing, 2005.

[56] R. Vuduc, E. J. Im, J. Demmel, A. Gyulassy, C. Hsu, and S. Kamil. Automatic performance tuning of sparse matrix kernels. *matrix*, 35(40):45–50.