January 2013

An Application Developed for Simulation of Electrical Excitation and Conduction in a 3D Human Heart

Di Yu
University of South Florida, diyu@mail.usf.edu

An Application Developed for Simulation of Electrical Excitation and Conduction in a 3D Human Heart

by

Di Yu

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yicheng Tu, Ph.D.
Hui Yang, Ph.D.
Swaroop Ghosh, Ph.D.

Date of Approval:
March 21, 2013

Keywords: GPGPU, Parallel Algorithm,
CUDA, SIMD, Cellular Automata

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis first reviews the history of General Purpose computing Graphic Processing Unit (GPGPU) and then introduces the fundamental problems that are suitable for GPGPU algorithm. The architecture of GPGPU is compared against modern CPU architecture, and the fundamental difference is outlined. The programming challenges faced by GPGPU and the techniques utilized to overcome these issues are evaluated and discussed.

The second part of the thesis presents an application developed with GPGPU technology to simulate the electrical excitation and conduction in a 3D human heart model based on cellular automata model. The algorithm and implementation are discussed in detail and the performance of GPU is compared against CPU.

CHAPTER 1: INTRODUCTION AND EVOLUTION OF GPUS

In recent years, industries and academics are witnessing an increasingly strong trend of renaissance of parallel computing. This is mainly because the microprocessors of modern computers are getting closer to the transistor density limit [1]. Moore's law is no longer sustainable because of the uncontrollable leak voltage of nanometer sized transistor gate feature of modern IC chips and the unmanageable power density problem on the chip [2]. A major shift towards processors with much higher parallel processing power had been the characteristic movement of the post Moore's law era ever since 2000, which promoted the development of multi-core and many-core CPUs.

At the same time, many other parallel computing schema has been proposed and studied, among which GPGPU is one major commercially successful effort that targeted desktop system [3]. To understand the fundamental differences between GPGPU and conventional CPU, it's beneficial to take a closer look at: (1) the evolution of GPU, (2) the problem domains that are suitable for GPUs, and hence (3) the architecture and design of the modern GPGPUs under the reference of current multicore CPUs. This review organizes the sections in this order accordingly. The intention of this review is to establish a general understanding of GPGPU technology, discuss the capabilities and limitations of the SIMD schema as the foundation of GPGPU, and demonstrate many potential applications that can be resolved more effectively with GPGPU than SISD approach applied in conventional CPUs.

1.1 Evolution of GPUs

1.1.1 Origin of GPU

GPU was first developed around 1980s to early 1990s to produce realtime graphics on computer screen as fast as possible [4], at which time graphic user interface was replacing the conventional character based console interface and became the ubiquitous method by which the mass consumers use computers. CPU developed at that time was not powerful enough to process the graphical commands and therefore not efficient at producing fast graphics on computers. Graphic generation requires the abilities to process large chunks of data with relatively simple arithmetic computing steps. CPU can perform complicated computation tasks, however it turns out to be a waste of processing power on such graphic computation. On the other hand, CPU does not have the throughput/memory bandwidth to handle large amounts of data quickly. Therefore, the computer industry decided to offload such tasks to a dedicated processor, which is called graphic process unit or GPU as comparison to central processing unit or CPU [7].

1.1.2 Development of GPU

GPU was first developed for 2D graphics acceleration for desktops to generate fluent graphical user interface (GUI) for the contemporary Apple or Microsoft operating systems. The main functionalities of these early GPUs are on 2D computer UI accelerations, display resolution and signal optimizations, and multimedia file encoding/decoding tasks. When the first generation of 3D computer games, such as Doom and later titles like Quake from ID software, obtained overwhelming successes on the market, computer industry recognized the strong demand on the processing power of GPUs, especially on the 3D graphic processing capabilities. GPUs received its first golden era of technology advancement [6]. Great investment and R&D efforts have been poured into 3D graphic GPUs and pushed forward the development of 3D capable GPUs. The computer game industry has become the main driving force for GPUs with higher processing capabilities, and catalyzed the formation of several major GPU processor manufacturers, such as S3, 3dfx interactive, and later ATI, Nvidia, and also joined by the behemoth microprocessor manufacturer Intel. The movement also generated a rich group of 3D Graphics APIs, such as 3dfx Glide, OpenGL, and DirectX. Under these platforms, many advanced 3D graphic processing techniques were invented to improve the efficiency and visual effect of rendering a 3D sense. New generations of games will test these 3D processing techniques in the market. Once widely accepted these techniques will be implemented by the GPU manufacturers with hardware support and made part of the GPU processing pipeline to generate better performance. Among them, the vertex processors, texturing processors and pixel shaders become ubiquitous. They were first incorporated into the GPU pipeline hardware and later brought into separated functional units for better scalability. At this stage, these functional units were still fixed and non-programmable in nature. GPU continues to optimize its memory architecture to achieve high bandwidth for data transferring.

1.1.3 Emerging of GPGPU

In the early time of computer science of 1970s, there had already been research efforts in utilizing graphic hardware to perform general purpose computation. England [8] described in his 1978 paper a system consisted of computer graphics processor and stereoscopic display to generate a spinal surface that models an isoparametric contours. England generated the left-eye and right-eye view of the surface, by computing with a unique display processor designed with a raster scan evaluation technology. The display is processed by multiple non-pipelined micro programmable modules at the same time.

In 1989, Potmesil and Hoffert at AT&T Bell Laboratories proposed a computer utilizing asynchronous MIMD nodes that has parallel access to large frame of memory. The intended application of such a parallel machine is to provide fast geometry and image –computing capability [9]. For a given image of dimension of mXn, a total of mXn pixel nodes will be launched to process the image. Each pixel node directly accesses the m-th pixel of every n-th scanning line. The node is physically implemented with a high-speed, floating-point programmable processor computer. Because of the flexibility of such node, any arbitrary algorithm for each pixel calculation can be implemented in software. Rhoades, et al [10] describe a software system called pixel plane graphics. It allows user-defined engine to specify many advanced computer graphical features, such as anti-aliasing procedures, user texture creation, adjustable diffuse and specular colors, sharp specular highlights, transparency and an object surface normal. To smooth processing for real-time texture, Rhoades introduced some of the techniques used in texture antialiasing program, which is capable of performing real time flashing animation of the texture of the water, environment mapping, and normal disturbances such as collision mapping.

Even though these early machines were originally designed to compute programs related to computer images and graphics, they utilized general purpose pipelining and provided programmability in such pipe lines, and therefore were categorized as the earliest research efforts in GPGPU area. Because of the overwhelming success of CPU combined with fixed functional unit GPUs at that time and also their own limitation in applications, these early researches achievements did not receive wide acceptance and adoption. However, the influence of their design philosophy can be found everywhere in modern commercially available GPGPUs.

At early 2000s, a new trend was showing in computer industry. The density of transistors in modern microprocessor design is approaching the limitation and the power wall prevents further complicate design of a single CPU chip. The entire industry switched the focus from higher chip frequency and more transistors in a chip to exploring the parallel computing capabilities with multi-core and many-core CPUs. Under such context, early researchers in the area of GPU recognized certain similar nature of some computational tasks between many scientific problems such as large matrix calculation, and 3D graphic processing. The pioneers in this field attempted to tweak their problem to map into the vertex processors, texturing processors and pixel shaders of fixed function GPUs, in order to harness the massive parallel processing power from

GPUs. Even at that time, the GPUs on the market did not offer general programming capabilities. Researchers found that many complicated mathematical calculation could be realized by combining the GPU hardware supported arithmetic operations. The problems solved with these techniques were no longer limited to computer graphic and image processing applications.

Heidrich [12] described a method to use fixed function GPU hardware available at that time to implement a wider variety of methods, such as multi-pass methods and flexible parabolic parameterization of environment map to model light reflection off an object. McCool [13] proposed a method combining some basic operations of GPU hardware to calculate a flexible per vertex lighting model. Min´e and Neyret [14] mapped the Perlin noise function to synthesize procedural textures with graphics hardware using OpenGL. Hoff et al. [15] utilized the high bandwidth of the GPU memory to calculate Voronoi diagrams, which was a novel attempt to explore the computational capability of GPU for non-graphic related application at that time. Lengyel et al. [16] performed robot motion kinetics in realtime by utilizing rasterizing hardware of GPU. Bohn [17] did computation on a Kohonen feature map by using a four dimensional vector function to represent a pixel of rectangle. An excellent survey paper of Trendall and Steward [11] summarized the research efforts of this period.

However, there are still many data intensive problems that could not be mapped into those fixed functional units, and therefore could not utilize the processing power of GPUs. On the other hand, traditional 3D graphic programmers were requesting more flexibility in their programs so that they were able to implement more customized effect for visual displaying. Under these driving forces in the market, GPU manufacturers such as Nvidia and ATI made a very significant strategic decision of developing programmable GPUs with APIs to support these market demands.

In 2001, GeForce 3 (NV20) was launched by Nvidia as the first GPU with programmable shaders, which marked the beginning of the era of consumer GPGPU. From then on, GPGPU experienced great expansion in scientific computation, machine learning, data mining, and many other areas that require massive parallel computing power to process large amount of data. At the same time, as traditional CPU technology is approaching the limitation of its growth, industries and academics start shifting their attention towards GPGPU architecture to explore the new territories of parallel programing and data intensive computation. Dedicated programming APIs are developed in this period for the GPGPUs. The most well developed ones include CUDA developed and promoted by Nvidia, OpenCL adapted by ATI. Many conventional software development platforms and IDE such as Visual Studio and Eclipse start to provide support for GPGPU program development. The industry is embracing the capabilities released by GPGPU and gearing up to harness its full power.

CHAPTER 2: PROBLEMS SUITABLE FOR GPUS

2.1 Problems Fundamentally Suitable for GPU

To understand the design considerations of GPU, we need to first study the type of problems GPUs try to solve. GPU was first made to solve computer graphical problems and accelerate displaying of generated computer graphics. These calculations typically involve solving expensive arithmetic computation for large amounts of data. A simple example can show how it scales. A picture of dimension 640x480 contains 307,200 points, while a picture of dimension 1024 x 1024 contains 1,048,576 pixels. An increase of dimension of 2 times of an image will increase the points by 4 times; and to be even more dramatic, an increase of dimension of 2 times of an 3D geometry object will increase the vertex by 8 times, should the same level of details to be preserved. Suppose for every single one of the pixels, we need to computation certain quantity with arithmetic operations involve floating point precision. Such scaling behavior will make calculate exponentially expensive on a CPU that only carries out series computational steps. On the other hand, the calculation for each pixel is independent and don’t have to be performed in any particular sequence. Therefore, GPUs with the underlining SIMD architecture is the ideal model to parallelize such calculation and accelerate the solution of the entire problem. On a more abstract level, GPU is designed to solve problems with following characteristics:

1. The problem consists of many small problems of identical operation with different input data.
2. The operation of each small problem does not depend on each other and can be carried out asynchronously.
3. The operation is arithmetic intensive yet logically simple (minimum branching conditions).

The above concludes the major characters of a typical problem GPGPU will face. It is obvious that parallel processing is the most efficient way to perform these types of tasks. For the last 10 years, ever since GPGPU became readily available in the market and programming for GPGPU becomes significantly easier and systematic, problems in vast fields such as engineering, science and financial have been tackled with powerful GPGPU algorithms. The following sections will introduce several typical categories that best suited for GPGPU acceleration.

2.2 N-Body Simulation

N-body simulation generally involves forces and physical interactions among many physical entities; such as electromagnetic forces between electrons, ions and atoms; molecule interaction between small molecules or large bio-molecules; gravitational force between planets, stars and galaxies. The algorithms need to calculate the net forces applied to each individual particle by all the other particles in the system and therefore, often has an O(n^2) time complexity. Applying algorithm that implementing GPGPU technique can drastically reduce the time needed for such calculation because every particle can be calculated independently. GPGPU algorithms are also developed for many N-body simulation related pre/post processing steps, such as density map calculation. For any given particle i, to determine the force it is subjected to, we need to sum up the forces between particle i and any other particles from 1 to n. To determine force of each particle, we need to run O(n^2) algorithm. GPGPU allows calculation of the force of each particle simultaneously, and therefore speed up the calculation. Here particles can be atoms, molecule, astronomic bodies; and the number n can approach hundreds of thousands to millions.

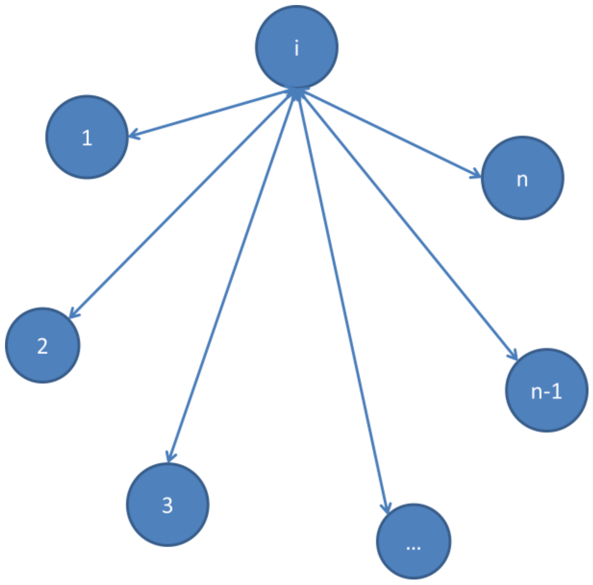


Figure 1. N-body simulation calculates pairwise interactions.

Koji Yasuda [18] carried out quantum chemistry calculations with GPGPU techniques. In one of this group’s work, they achieved more than 10 fold acceleration. Rodrigues and Hardy [19] used GPU to accelerate the pair potential for molecular simulation, which gain 5X to 13X performance gain over classic CPU calculation. They also developed a cluster algorithm that based on multiple connected GPU, which maintains same level of performance on scaled up problems of same nature. Rio Yokota et al [20] developed a multipole method for n body simulation with cuda and achieved at least one order of magnitude performance gain. It had become a common practice in the study of molecular dynamic simulations, that data parallel process implemented with GPGPU techniques is used to perform efficient computation of such models.

2.3 Finite Element Analysis (FEA)

FEA was widely utilized in computational engineering applications to solve ordinary differential equation (ODE) and partial differential equations (PDE). The nature of FEA method is to divide the object under simulation into many smaller pieces and solve the ODE or PDE on each small piece with a numerical approximation. The final solution is a numerical combination of solutions along every small divided piece. The nature of FEA fits perfectly within the reign of GPGPU computational capability, which intrigued large attention and effort of moving FEA methods onto GPGPU platform.

Fluid dynamics simulations are typical FEA simulation that can benefit from GPGPU computation technique. Mark Harris [21] developed a fast algorithm that utilizes GPU to perform FEA simulation of fluid dynamic. This work becomes the ground work followed by many later researchers in fluid dynamic simulation areas. Researchers in the field of chemical engineering, electronics, and earth physics that relies on FEA enjoyed speed up with GPGPU. Dimitri etc. [22] carried out FEA simulation of seismic wave. They were able to accelerate the high order FEA simulation with GPU to perform significantly better than CPU algorithm.

2.4 Data Mining and Machine Learning with Big Data

Bid data is attracting great attention lately because the academy and industry believe information and knowledge buried in bid data is the key to boost profit for tomorrow’s business and technology world. It is always a challenge to process large dataset and extracting valuable information in a timely manner. Very often such information is time sensitive and therefore any measure that speeds up the process is of great interests. Naturally data mining and machine learning through big data found their share of promise in GPGPU architecture.

Weinman [23] etc. developed a machine learning algorithm that runs 30 times faster than the CPU algorithm on a same machine. They attribute the speed up simply to the higher number of GPGPU cores over the CPU cores. Fang et al [24] performed data mining algorithm on graphics processor and observed 10 to 5 times speed up with large sets of data. Map reduce is a popular algorithm to process large online data set, and is adopted by Google and Facebook etc. to process their huge volume of user data on the Internet. Map reduce can be carried out by clustered computers or GPGPUs. Stuart et al [25] showed that their GPGPU cluster based map reduce can be 8 times faster than the best performance of CPU map reduce package.

Among many areas that utilize data mining technique for data analysis, financial industry has its own unique character and requirement. Financial industry critically depends on fast and accurate financial data processing and calculation. Many of their applications need to be real time with stringent latency tolerance. Again, GPGPU found its important role in solving challenging financial data processing / analyzing tasks. Preis and Tobias [28] used GPU to perform autocorrelation function to analyze the fluctuations observed in financial market and achieved 40 times acceleration with their model. Such significant improvement has utter importance in today’s competitive financial market.

2.5 Image Processing and Computer Vision

GPGPU is natively constructed for computer graphics processing. However, there are a large category of image processing applications that do not belong to the conventional computer graphics task. This type of image processing involves pixel by pixel modification by certain algorithm and normally the input is some forms of 2D images. Typical applications include but are not limited to: picture filter effect, image enhancement, medical image processing and reconstruction. On top of these elementary operations, artificial intelligence and computer vision techniques can be applied to realize various innovative functionalities. Researchers in these areas found that their algorithms and programs contain series of similar features that can be accelerated nicely with GPGPU programming paradigm. Any computation that requires pixel by pixel work can be considered for parallelized GPGPU processing, to reduce the time complexity from O(n^2) to O(n) by processing each pixel simultaneously instead of one by one.

Jeong et al [26] adapted GPGPU to implement ITK image filter algorithm for medical image processing tasks to obtain close to real-time performance.

James [27] implemented their computer vision algorithm on GPU and achieved 21X speed up. They introduced two open source computer vision projects that based on GPGPU, OpenVidia and GPUCV, which streamline the tasks of video acquisition, image processing and computer vision into a group of self-contained packages.

2.6 Numerical Algorithms, Parallel Algorithms and Data Structure

There are a group of researchers that placed their focuses not on particular sets of application-oriented problems, but rather on more fundamental problems of GPGPU programming paradigm adaptation for elementary algorithms and data structures. Some of the algorithms and data structures are natively compatible with parallel programming paradigm while others are not. It is difficult to find a universal pattern for different algorithms and data structures to adapt parallel programming paradigm, and some of them might never obtain as much performance gain as others. However, these fundamental researches are constantly going on and any breakthrough in these areas will cause profound change in the landscape of computer science.

One typical and outstanding work in this area is Kim's work [29] on tree search algorithm on GPU and CPU. Kim's work focused on optimizing the tree search algorithm both on CPU and GPU. A cache aware algorithm is adapted for both CPU and GPU. Tree search by itself is not easily parallelizable because where the next level of tree to search always depends on the outcome of search results of previous level. Such data dependency kills possibility of searching every tree levels at the same time. Kim's algorithm, instead of trying to improve individual search query execution time, focused on increasing the throughput of queries. With GPGPU, large amounts of queries can be carried out simultaneously and therefore performance improvement is gained when big data needs to be searched. Similar to GPGPU algorithm, Kim's algorithm on CPU utilized the SIMD pipeline as well, such as Intel's Streaming SIMD Extensions (SSE) and AMD's 3D now pipeline. The algorithm paid special attention to data transfer to make sure the L1 cache, L2 cache and memory buffers of the system can be maximally utilized. Similar consideration was addressed in their GPGPU algorithm as well. Because of the much simpler cache structure, the GPGPU algorithm structure is less complicated than it is in CPU.

The sub-tree size is carefully selected so that the node data of sub-tree can fit into a shared memory block on GPU and minimum memory access is needed to process query within the sub-tree. The GPGPU implementation performed at approximately 2X better than CPU version. Even though the speed up may not be as impressive as many other applications, considering this FAST framework went out of their way to optimize the code for CPU and tree search do not natively support parallelism very well, it is fair to say that GPGPU again greatly outperformed CPU.

Modern GPGPU design went great extent to optimize the architecture for problems of these categories, and the consideration of these optimizations are significantly different from conventional considerations in CPU architectures. The following sections will illustrate several most important aspects of the architecture design. We will first review a typical modern CPU architecture as a reference point to provide benchmark when GPGPU architecture is presented.

CHAPTER 3: GPGPU ARCHITECTURE

3.1 CPU Architecture

Each core of the CPU is designed to perform single process of complicated instruction sets, including integer arithmetic (ALU), floating point operations (FPU), memory accessing and various control logic switching instructions. Many of the optimization techniques focus on the following areas listed below.

1. Out of order execution: the instructions are reordered when they are executed to maximize the utilization of the execution units while preserving the correctness of the computing.
2. Branch prediction: CPU tries to intelligently guess a branching condition result and prepares in advance the instructions and data for the branched operation. This way, CPU can partially reduce the branching cost.
3. Cache hierarchy: Data in the memory is pre-fetched and stored in fast and close by cache memories based on temporal and special locality principles. Such cache system can be very complicated multi-layer structures (as much as three levels of cache) and the performance of the cache (hit rate, miss rate, miss penalty) will have great impact to the performance of CPU.

4. Virtual memory: Virtual memory was implemented first to extend the physical memory and allow utilization of hard drive for memory storage. Later when memory price dropped significantly, virtual memory found another important functionality, to divide the physical memory into logical address spaces so that programs can access their memory address under a uniform interface that is managed by OS and hardware (TLB). This allowed one CPU core to execute multiple programs.

In the following section, we intend to compare GPU architecture against each aspect of CPU architecture and thus demonstrate the reasons of their vastly different design.

3.2 GPU Core Structure

GPU is designed to process large amount of data with great parallelism. Therefore, GPU is equipped with large amounts of cores to achieve maximum degree of parallelism. On the other hand, each core of GPU is substantially simpler than a CPU core because each GPU core is not intended to perform logically complicated computations. Since every core of GPU only executes a fixed set of instructions, it becomes redundant to let each core to have its own instruction fetch and decode module. Therefore, all the cores in one GPGPU multiprocessor unit share one instruction fetch and decode module, which further simplifies the structure of GPGPU. This architecture simplification is widely known as SIMD, which stands for Single Instruction Multiple Data processing.

Generally one GPU device contains several to tens of stream multiprocessor on board. Every GPU SM only contains the most essential parts of a CPU core, which are instruction fetching and decoding, instruction execution, and execution context. GPU core do not have out of order instruction execution or branch predictor because the instructions it executes are not supposed to have complicated logical branching conditions and they should mostly contains arithmetic computations that can be quickly processed by the functional units of the execution module. For this particular case, when no branching has happened and all 16 units executed the same instructions, there will be 16 times performance speed up. However, if the instruction contains branching condition, it is possible that some of the core need to execute instructions that are different from the other cores; depending on the branching condition's results. The more branching in the code, the less parallelism the system will be able to enjoy. Branching in the program for GPGPU should be avoided due to the execution pattern GPGPU will carry out with branching conditions. By design, many GPU cores share one fetch and decode unit, therefore every GPU core that share the same fetch and decode unit have to perform the same computational logic. If branching happens to a portion of the cores, the fetch and decode unit need to fetch the branched code for the affected cores and therefore the unaffected cores have to be stopped and wait for the termination of the branching code on the affected cores. This is actually stopping the parallelism and executing the code in a series manner, which will unavoidably introduce performance penalty to GPU program. Han et al [30] proposed a work around for branching conditions. By coding the intended branching condition in a particular pattern, certain branching situation can be delayed or distributed so that reduced penalty will occur. They observed 30% to 80% performance improvement against the naïve branching algorithm in a GPGPU program. Nevertheless, to maximize the utilization of GPGPU cores, branching condition should be avoided at all cost.

A more detailed illustration can be found in Hou [31]. The paper described the detailed inter configuration of a streaming multiprocessor, which is the typical configuration of Nvidia Fermi architecture. Bo Zhang [33] showed a figure illustrating the organization of SMs in an Nvidia Fermi GPGPU device, which indicates every 4 SMs is grouped into a Graphic Processor Cluster (GPC) or Thread Processor Cluster (TPC). One GPGPU device can have 4 GPCs and therefore 16 SMs and 512 cores in total. The GPGPU device contains 4 GPCs and in each of the GPC resides 4 streaming multiprocessors (SMs).

3.3 GPU Memory Hierarchy

Like CPU architecture, GPGPU cores also have sophisticated cache systems. To some extent, the memory hierarchy system of GPGPU might be even more complicated than a CPU memory system, due to the history of supporting computer graphics operation, which requires texture management etc. GPGPU also has a virtual memory system. The lowest structure in the memory hierarchy of a GPGPU system is the global memory that can be accessed by every GPU core. This global memory is normally limited to the on board DRAM video memory we found on a video card. GPGPU can only access data located in the global memory, which have to be transferred through CPU DRAM. GPGPU does not have a mechanism to access I/O or hard drive, which limits its performance for some applications.

The exact memory hierarchies of GPGPUs were never clearly revealed by the GPU manufactures. Researchers probe into the GPGPU by benchmarking the device performance under carefully designed tasks and deduct the hierarchy from the bench test results.

Wong et al [32] performed micro-benchmark to detect the internal organization of the multi-core and the memory hierarchy within the GPGPU device. According to their results, the Nvidia GT200 card has 16KB shared memory with latency of 36 cycles for each SM unit. This shared memory should not be considered as part of the cache because it is not cached. Rather, when L1, L2 and L3 caches are discussed, they are normally referred to as components of the texture memory and constant memory. For instance, in GT200, texture L1 cache is 5KB with latency of 261 cycles and texture L2 cache has 256KB size. Both texture cache is part of the texture memory and is visible globally. The global memory of the device is visible to every thread and SM, the latency of global memory is around 440 cycles. Global memory is not cached. The constant memory is used to store user specified data and also part of it is reserved for compiler generated constants. 64KB of it can be accessed by user. Constant memory is cached, with L1 2KB private to each SM, L2 8KB shared among SMs on a same Thread Processing Cluster (TPC), and L3 32KB shared globally.

Ueng etc. [34] proposed the simplified CUDA programming model which shows a simplified memory accessing hierarchy of NVIDA GPU device. Many more comprehensive figures that indicate the structure can be found easily in open literatures and NVidia published technical documents.

3.4 GPU Memory Access

Since all the data need to be brought to the core from memory for computation, GPGPU should have suffered much significant memory latency problem than CPU, due to the common large volume of data GPGPU need to deal with. However, GPGPU utilizes a method called streaming to allow a way to hide the memory accessing latency. The concurrency of large amount of instruction streams combined with fast context switching can be used to mask memory latency. When one instruction stream is stalled due to unavailability of data, another instruction stream is then switched to the stalled core and continues execution. The utilization of the core can be maximized in such an execution manner. The switching between instruction streams is very fast and efficient with minimum overhead, generally only takes a few clock cycles. Normally the contents in the register are not switched out during the instruction stream switching. These contents are kept in the register memories and when the previous instruction stream switched back, they can be picked up instantly. A figure in NVidia CUDA programming guide illustrates the memory latency hiding mechanism.

Kayvon et al [35] provided an example to demonstrate memory latency hiding mechanism in a GPGPU device. It is concluded that for the modern GPGPU design with typically 16 processing cores, the GPGPU needs 8192 parallel processed data fragments to keep the GPGPU cores running at peak occupancy. Therefore, the more of the parallel computation work, the better the processer cores can be utilized.

Volkov et al [36] studied the memory latency of GPGPU, and designed a matrix multiplication algorithm that specifically targeted at maximizing the memory latency hiding technique. The algorithm adaptively changes the computation unit size, which is called pivotal point, to allow optimized memory accessing and cache utilization. On one hand, memory accessing time is reduced; on the other, the data available rate or cache hit rate is improved. The overall performance of their memory latency hiding technique aware algorithm is 13 to 21 times faster than naïve GPGPU matrix multiplication algorithms.

3.5 GPU Memory Types

GPGPU inherited several different types of memories from graphic processing pipeline. These memories have different physical locations and API accessing interface. They are optimized for specific read/write patterns. They include texture memory, constant memory, shared memory and global memory. Only the shared memory and global memory will be explored in this section, since in most GPGPU programs these two types of memory are the focus of most of the optimization works.

Shared memory is the one built on chip and can be accessed by all the cores on the same SIMD unit, which is on the same Streaming Multiprocessor SM. Shared memory is often considered as a manually manageable cache. It has much smaller size than global memory but because the vicinity toward GPGPU core, the accessing speed (1+ TB/s) is order of magnitude faster than global memory. The typical latency of shared memory is in the order of tens of clock cycles. When manually declaring shared memory, one needs to be aware that the execution contexts of the same SIMD unit also utilize the same physical memory and when certain amounts of share memory is declared, the amount of threads that can be launched for the multi cores will be reduced or limited. On the other hand, global memory of GPGPU is much larger and is the only gateway that the outside world can send data to GPGPU for processing. However, global memory bandwidth is in the order of 150GB/s, which is the bottle neck for many GPGPU programs. The latency of global memory often falls into several hundreds of clock cycles.

When executing a program, GPGPU launch multiple threads onto each streaming multiprocessor. Each streaming multiprocessor execute 32 threads at a time given each streaming multiprocessor contains 32 cores. Such a batch of threads are called warp. Even though warp was not directly addressed in API of GPGPU programming guide, a programmer should be aware of it and design the algorithm accordingly because threads within a warp access the memory in a specially manner.

When a warp of threads access the global memory, the data access can be achieved by one transaction if no offset is present in the location the warp need to access. Such access is called coalesced memory access. To utilize global memory efficiently, un-coalescing memory access should be avoided because in that case, each thread within the same warp needs to take their own time to access data in global memory. To achieve coalescing memory access, programmers need to plan carefully of how to store the data needed for computation and how to launch threads to access these data. Later GPGPU API improved on the un-coalesced memory accessing performance and reduced penalty caused by un-coalesced memory access.

Shared memory has a different behavior when accessed by threads. Shared memory is divided into smaller modules called banks. Accessing different banks can be carried out simultaneously and therefore fully utilize the memory bandwidth and low latency. However, if multiple threads need to access the same bank, their requests are serialized, and therefore causes waiting in the process. To maximize the memory bandwidth, it is important to arrange data in such a manner that all the threads will access their data from different banks. Since shared memory is 100x faster than global memory, such effect is normally considered secondary to the global coalescing effect. A figure in [39] shows the mechanism of un-coalesced memory access and the way to avoid it.

Vast effort had been made in the area of optimize memory access in GPGPU programming. Harris et al [37] described strategies to avoid or reduce the un-coalesced data access both in global memory and shared memory. Boyer et al [38] demonstrated an automatic software tool to detect bank conflict together with many other GPGPU critical performance metrics.

3.6 Conclusion

The problems GPGPU architecture was designed to tackle with are generally the ones with large data set and operation to the data can be carried out by independent threads.

Given correct data intense program, the GPGPU approach is capable of achieving 10 to 100 times performance gain compare to CPU approach. However, to fully harness the power of GPGPU, one must have insights to the characteristic feature of architecture of GPGPU. Avoid branching conditions to allow maximum parallelism in the algorithm. Pay attention to memory latency hiding techniques and prepare the data in a cache friendly manner helps improve memory accessing latency. Arrange the data accessing pattern to fully utilize simultaneous data transaction helps program to achieve peak performance. The deeper one understands the architecture of the GPGPU, the better performance he/she can achieve.

CHAPTER 4: GPGPU SIMULATION OF A 3D HEART

A correctly beating heart is important to ensure adequate circulation of blood throughout the body. Normal heart rhythm is produced by the orchestrated conduction of electrical signals throughout the heart. Cardiac electrical dynamics is the resulted function of a series of complex biochemical-mechanical reactions, which involves transportation and bio-distribution of ionic flows through a variety of biological ion channels. Cardiac arrhythmias are caused by the direct alteration of ion channel activity that results in changes in the AP waveform. In this work, a whole-heart simulation model is developed with the use of massive parallel computing with GPGPU and OpenGL. The simulation algorithm was implemented under several different versions for the purpose of comparisons, including one conventional CPU version and several GPU versions based on Nvidia CUDA platform.

OpenGL was utilized for the visualization / interaction platform because it is open source, light weight and universally supported by various operating systems. The experimental results show that the GPU-based simulation outperforms the conventional CPU-based approaches and significantly improves the speed of simulation. By adopting modern computer architecture, this investigation enables real-time simulation and visualization of electrical excitation and conduction in the large and complicated 3D geometry of a real-world human heart.

4.1 Introduction

Computer simulation of human heart is receiving increasing attention because it empowers scientists to advance cardiac research and battle against heart disease. The heart beat is the result of a series of complex biochemical-mechanical reactions, which involves transportation and concentration distribution of a dozen of ionic and molecular species in heart tissue through a various different biological channels. These biochemical movements of cardiac ions (i.e., Na+, Ca2+, K+) create a local electrical potential variation, which in turn causes the contraction of heart cells. The 4 phases of electrical potential of cardiac myocytes (i.e., action potential – AP) are as shown in the Figure below, which can be found readily in many open literatures and text books.

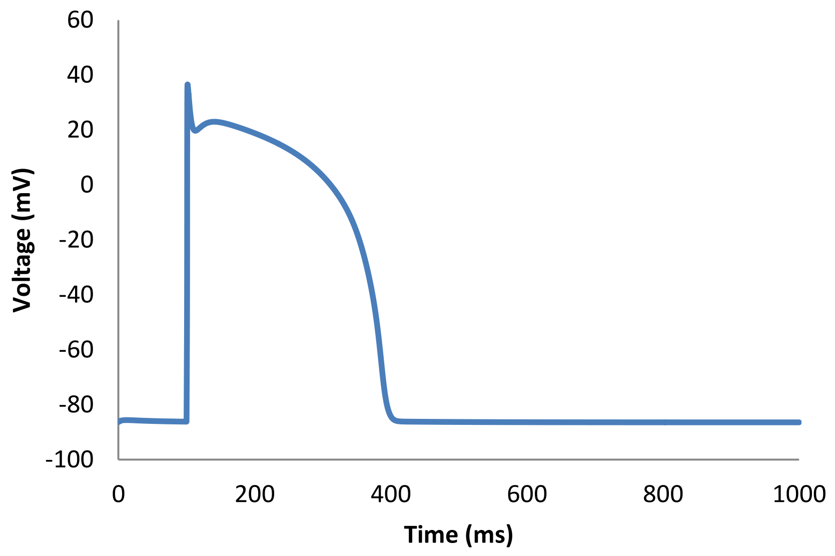


Figure 2: Heart muscle cell electrical potential variation cycle.

This electrical potential fluctuation will propagate through every cell of the entire heart to complete one cycle of heart beat. An ideal comprehensive simulation model will take into consideration of every aspect of this complicated biochemical and mechanical interaction and simulate the process on a full scale of 3D model. However, because of the complexity of cardiac system, the unknown/unclear aspects of many details involved in this biological process, and the limitation of modern computational power; currently available simulation models usually simplify certain aspects of the computation and generate an approximate result of the real-world cardiac process. If the bio-chemical interaction is carefully monitored and various ionic and molecular species are simulated, the model will only be able to simulate the activities of 2D tissue. With currently available (2012) computational power, several models were developed under such scheme and are all only capable of performing calculation for simplified geometry such as 1D or 2D tissues, under a non-realtime basis [41].

Our approach follows a different methodology, which focuses on the macro-scale behavior of the entire heart. To do such a large-scale simulation, we greatly simplify the micro-scale simulation by only computing the propagation of the electrical potential and ignore interactions of any other bio-chemical species involved in this biological process. This approach allows us to observe and analyze the excitation pattern of a whole heart in realtime condition, which has not been well established before. To facilitate this demanding task, a massive parallel computation model is developed based on the modern GPGPU (i.e., General-purpose computing on graphics processing units) architecture.

The organization of this chapter is as follows: Section 4.2 presents research methodology of the model. The algorithm and implementation is described in Section 4.3 in detail. In Section 4.4, we compare the performance of this algorithm under conventional single CPU architecture and under modern GPGPU architecture. Section 4.5 concludes this present study.

4.2 Research Methodology

4.2.1 Cellular Automata Based Model

Two major methodologies coexist in the area of heart excitation research. The first is based on a macro/minimal approach, such as cellular automata models, FitzHugh-Nagumo model[42], and Barkley Model [43]. The second approach is based on micro/maximal models, e.g., Hodgkin-Huxley model, Luo-Rudy model, Rasmusson model and Nygren-Lindblad model [44][45]. The macro/minimal models simplify simulation of ion physics, but they are easier to be incorporated into a large-scale system and they can be computed much faster. This present study is based on the first type of the simulation approach, assuming heart tissue cells are uniformly distributed excitable media and the excitation and propagation of cardiac electrical potentials are governed by the potential of the directly adjacent neighboring cells. The cell will stay in excited condition for an experimentally predetermined period, and then drop back to resting condition and waiting for the next excitation. Based on clinical experimental data of heart tissue and only considering the measured potential of the cells, we free ourselves from the time-consuming calculation of the underlying bio-chemical process, and avoid the uncertainties and inaccuracy introduced by unreliable and /or unavailable constants and parameters in the biological transportation model of ions and molecular species in various channels of heart tissue. With the modern GPGPU computer architecture, a fast algorithm is developed that can handle large and complicated 3D geometry of a real human heart.

4.2.2 GPGPU Acceleration

Our model can be categorized as one type of cellular automata models. Cellular automata models have been widely applied in computational simulation research on biological processes, excitable media modeling, disease and disaster propagation simulation, macroeconomic fluctuations in human society, and game of life. To achieve real-time simulation, modern GPGPU computing architecture is employed to accelerate the massive parallel processing of the simulation. The cellular automata model is developed in a parallel computing framework. Each unit in such a system is only determined by the status of its neighboring units. To determine the overall status of the system, each unit's status can be calculated independently. Such a problem is perfect for harnessing the computational capability of GPGPU processor.

4.2.3 Algorithm

Since only the potential of the cell is consider and the potential can be determined by time alone in our simplified model, we can model the excitation of a single cell if we keep track of the duration, for which the cell has entered excitation status. For example, if a cell is excited at time zero (t=0), it will stay in excited status for 200 ms (t=200) and then drops back to resting status. A cell can only be excited when there is a neighboring cell is in excited status. It would be ideal if each cell in the heart can be represented by one point in the 3D heart model; however that will generate an overwhelmingly huge 3D model that cannot be handled by current computers. Therefore, a 3D heart model with limited resolution has to be used and the distance between two neighboring points will cover a space between many heart cells. This introduced a deviation of our model from conventional cellular automata models. Our model needs to calculate the distance between any two neighboring points and estimate how much time it will take for the excitation to propagate from one point to the other. The whole algorithm is provided in the section below. The 3D heart model consists of two files:

1. A list of all the points representing mass of a heart. The information for each point includes the index of this point, and xyz coordinates of this point.
2. A list of all the tetrahedral formed by any 4 conjunction points. Each tetrahedral is represented by index of 4 points.

The data preparation step:

1. For each point, find out the neighboring points of this point and generate a linked list of these neighboring points for this point. This is done by traversing the tetrahedral file and adding a link to the linked list of each point if this point is found in current tetrahedral data.
2. Calculate the distance from each point to every neighbor that this point has a conjunction relationship.

After the data preparation step, we then have an array of all the points and each element of the array is a linked list recording every neighboring point and distance to it. This data structure can be represented in the section below.

Start point represented is such a format: the index of the point→neighboring point 1 (distance), neighboring point 2 (distance), neighboring point 3 (distance), etc. An example is shown is the following section.

0→1(8),3(8),4(6.9282),5(6.9282),7(8),8(6.9282),249(6.9282),2692(7.64532),2695(6.9282),2696(7.7316),2698(7.32724),2713(8.57508),2717(5.65685),2718(6.63325),2721(6.9282),2740(5.65685),2741(6.63325),2742(6.63325),2759(4),2760(6.63325),2762(5.65685),2764(5.65685),2767(6.9282),2769(8.30697),2771(5.65685),2772(8.34392),2775(6.9282),2803(6.63325),2808(4),2809(5.65685),2810(5.65685),2824(6.63325),2825(4),2826(5.65685),2830(5.65685),7789(5.65685)total neighbor36.

1→0(8),2(8),4(6.9282),5(6.9282),15(8),17(8),18(6.9282),19(6.9282),27(6.9282),37(6.9282),2720(6.63325),2721(6.9282),2762(5.65685),2767(6.9282),2806(6.63325),2810(5.65685),2863(6.63325),2865(5.65685),2922(6.63325),2926(4),2927(5.65685),2928(5.65685),7745(6.63325),7747(5.65685),7748(6.63325),7789(5.65685),7790(6.63325),7792(4),7793(6.63325),7797(5.65685),7801(5.65685)total neighbor31.

2→1(8),3(8),4(6.9282),5(6.9282),9(8),11(6.9282),12(6.9282),26(8),27(6.9282),36(6.9282),37(6.9282),38(6.9282),2811(6.63325),2817(6.63325),2821(5.65685),2925(6.63325),2927(5.65685),2931(6.63325),2934(4),2935(5.65685),2936(5.65685

),7795(6.63325),7797(5.65685),7798(6.63325),7813(5.65685),7814(6.63325),7816(4),7817(6.63325),7820(5.65685),7823(5.65685)total neighbor30.

The computation steps:

1. Make an array of timer for each point. The value of timer indicates the current status of the point. If the timer value is in between 0 and 200, the point is in excited status. If the timer value is greater than 200, the point is in resting status. If the timer value is negative, the point has an excited neighboring point and will be excited in a later time when timer increases to 0. Initialize all the timer value to a large positive number such as 5000, which indicating the whole heart is at resting condition.
2. Select one particular point, and set its timer value to 0 as making this point the starting of the excitation.
3. For each point that the timer value is 0, look up the neighboring points of this point. We will call this point as starting point, and the neighboring points as ending points. Depends on the timer value, we will do one of the three possible tasks to each ending point's timer:
    a) if the ending point is already excited (timer value is within 0 to 200 range) do nothing
    b) if the ending point is in resting status (timer value > 200), set its timer value to a negative integer, which equals to floor(C*d), here C is a negative constant that represents the propagation speed of such excitation within heart tissue, and d is the distance between starting

point and the ending point, and the floor(C*d) function takes the maximum integer that's less than C*d.

c) if the ending point's timer already has a negative value, which means some other starting point has already set the timer of this ending point, compare its current timer value to floor(C*d) and set the timer value to the one with less absolute value. Smaller absolute value of the negative timer indicates excitation reach the ending point quicker.

4. Increment every point's timer by 1.
5. Repeat step 3 and 4 and we will observe excitation propagation by visualizing the 3D heart according to the timer value of each point.

4.2.4 Algorithm Discussion

Under the framework of our current model, we have specifically considered the following points:

1. Since the model use discrete time to calculate the status of the points, we have to assign integer value to the timer of each point, and thus the floor function is adopted to round off digits after point. The floor function will introduce certain error into the calculated results. However, we do not expect such error will be significant since we are modeling the process in a very fine resolution of 1ms.
2. The excitation propagation speed constant, C, should be determined with experiment and might varies under different situation.
3. The excitation duration from 0 to 200 ms is based on experimental results and might vary from case to case.

4. The location and number of initially excited point(s) can be picked arbitrarily
5. We can observe spinal wave formation under certain circumstances, such as excitation time span is too small or multiple excitation starting points are selected. This indicates the dangerous heart condition of spinal excitation rooted from the nature that the heart is an excitable media. Such media always has the potential of developing spinal wave pattern when the condition is right.

4.3. Materials and Experimental Design

4.3.1 Implementation Platform

a simulation application is implemented on an Intel dual core i3-2100 CPU @ 3.10GHz machine with 16G memory. The graphic card is an Nvidia Telsa 2075 with 6GB memory, for a fast rendering and fast GPGPU calculation. The operating system is 64bit Window 7. The simulation application consists of two major components, the simulation component and the visualization component. OpenGL was selected as the visualization / interaction platform because it is open source, light weight and universally supported by various operating systems. The application is developed with C++ under Microsoft Visual Studio 2010. The source code of the application should be relatively easy to transfer to other operating system since no special dependency to Microsoft VS IDE. OpenGL provided the basic rendering loop iteration and GUI mechanism. The simulation algorithm of application was implemented under several different versions, one conventional CPU version and several GPU versions based on NVidia CUDA platform.

4.3.2 OpenGL User Interface

The glew and freeglut libraries are used for common OpenGL rendering and UI event commands. OpenGL automatically adjusts the drawing frame rate to accommodate the simulation. The regular frame rates OpenGL supports are 60fps, 30 fps, 20 fps, 15fps, and 10 fps and below. To have interactive results, a 30fps and above frame rate is preferred. In each rendering loop, if the simulation takes less than 1/60 second, OpenGL will wait until 1/60 second is reached and draw one frame, thus resulting in a 60fps rendering rate. If the simulation takes longer than 1/60 second but shorter than 1/30 second, OpenGL will then wait until 1/30 second is reached and then draw one frame and thus resulting in a 30 fps rendering rate. Therefore, the ultimate goal of simulation is to efficiently calculate the status of all points in shortest possible time.

Several functions were introduced to facilitate realtime user interaction with the 3D model. They are listed below:

1. Restart simulation at any time
2. Manually select any excitation point and restart simulation at any time
3. Introduce new excitation at any time during a simulation process
4. Freeze the simulation at any time and observe simulation evolves step by step at any time.
5. Change the excitation time range and excitation propagation constant at any time

6. Slice the heart open and into pieces and observe realtime or frozen excitation propagation within the heart
7. Observe the simulation under point cloud mode or regular rending mode

All these functionalities are realized under OpenGL events and can be controlled by either keyboard or mouse.

4.3.3 CPU Simulation Implementation

A straightforward implementation under CPU architecture is carried out. The algorithm traverses through all the 148516 points in the 3D model to finish one round of iteration of the simulation calculation and determine the status (timer value) of every point in the 3D heart. It then calls the OpenGL to render the whole heart according to the timer value of every point. A gradient color scheme is adapted for rendering the heart. The white color represents the exact moment of excitation of the point (t=0), the red color represents the waiting and resting status and (t<0 and t>200). The color transitions from white to yellow, orange, and red when the timer value increases from 0 to 200.

4.3.4 GPU Simulation Implementation – Case 1

The original algorithm is parallel modified to adapt to NVidia GPU architecture. Instead of iterating through every point, we issue every point a thread, which is responsible for incrementing the timer of the thread, looking up the neighboring points, and calculating and storing the negative timer value for the point if applicable. We thus launch 148516 threads to the GPU multicore processors and run these threads in parallel. The status of the timer for each point is stored in an array located in the global video memory. The neighboring points and distance data are stored in arrays located in the global video memory as well. Each thread will access the global video memory when it needs to, and thus made the application a memory accessing intensive one instead of a processor/calculation intensive one. Once all the threads finish their work to update the timer value, a second GPU kernel is launched, again one thread per point to update a color buffer of the points, which is stored in global memory of the video card. The color is calculated according to the timer value of each point. This step is again a memory accessing intense step. After the color buffer is updated, CUDA hand OpenGL a pointer to this color buffer so that OpenGL can perform rendering based on the color data. No extra copy of the data in the color buffer array is needed. This technique is called OpenGL/CUDA coop, which will significantly reduce the time needed for rendering.

4.3.5 GPU Simulation Implementation – Case 2

Because the algorithm is a memory accessing intense one, it is possible to improve the performance by reducing the memory access. To do that, we consider moving the data into faster shared memory and accessing them from there. Since CUDA copy memory in a batched manner, instead of individually, when a particular start point needs to access the first neighboring point data, all its neighboring points data will be copied to shared memory at the same time. Therefore, we can copy only once and utilize the neighboring data multiple times. To utilize that, a thread for each neighboring points is launched. Because of the irregular data pattern, some points have up to more than 200 neighbors. Therefore, we launch 256 threads for each starting point. Each of these 256 threads will working on one neighbor point of that starting point. The first thread will be responsible to copy the entire data of the neighboring points into shared memory. Once that data is accessible in shared memory, all the thread will find a corresponding neighboring point to work with. The task performed are increment timer array that is located in global memory, calculate C*d if applicable and update timer value accordingly. If the thread amount is greater than the total neighboring points, extra threads will be idle and wasted. We perform such actions for all the 148516 points and finish one round of iteration of simulation computation. After this kernel is finished, a second kernel is launched to update the color buffer that's stored in the global memory. The second kernel operates in the same manner as the first implementation of GPU.

4.3.6 Performance Comparison

Three implementations were executed under the identical hardware and software environment. The average time required computing the status of all the points in the 3D heart model was recorded and compared.

4.4 Results

4.4.1 Visualization and User Interface

The heart 3D model is color coded according to its anatomic structure. The slicing function can be applied conveniently to show the inner condition of the heart. Totally 6 slice plane can be applied to divide the 3D model into any desirable configuration as shown in Figure 3 and 4.

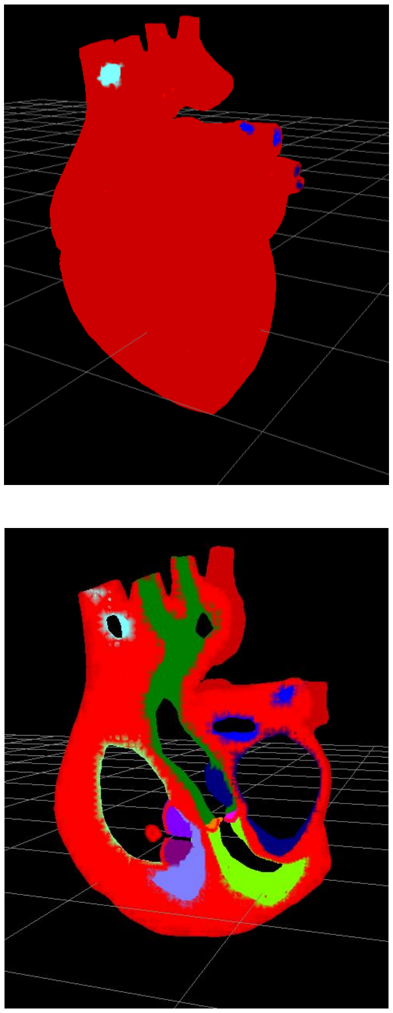
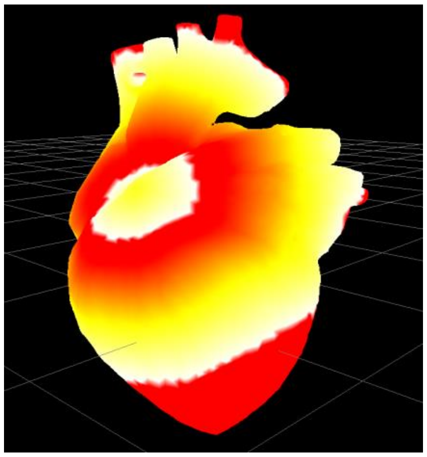
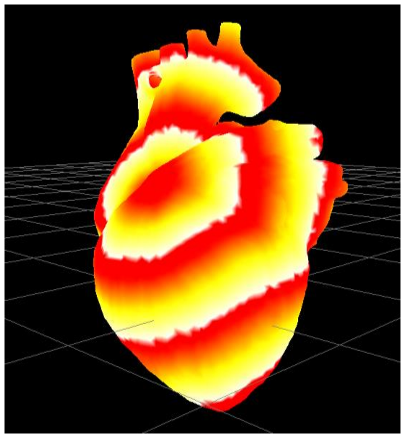


Figure 3: 3D heart model (above) and sliced into a piece (below)

The simulation process can be viewed continuously. The progress of the propagation of the heart excitation wave can be paused at any moment. The excitation and propagation speed can be adjusted manually as shown in figure 4 (a-b). When cells are firstly excited and the electrical potential reaches its peak value, the color turns white. As its status gets back to resting condition, the potential decreases and the color turns yellow, orange and eventually original red color. The excitation origin can be handpicked by mouse. The excitation can start from a single point, several points or a continuous region.



(a)



(b)

Figure 4: Different excitation time span and excitation band width. (a) Time span = 200, (b) Time span = 50.

Spontaneous spiral waves can be observed under certain conditions, such as repeatedly exciting a certain area of the heart as shown in the figure 5 (a-d). This is more prone to happen when the excitation band width is relatively narrow, and the excitation wave front has better chance to penetrate the wave thickness and reaches its own wave end. The slicing functionality helps visualize the formation and evolvement of the spiral wave inside the heart.

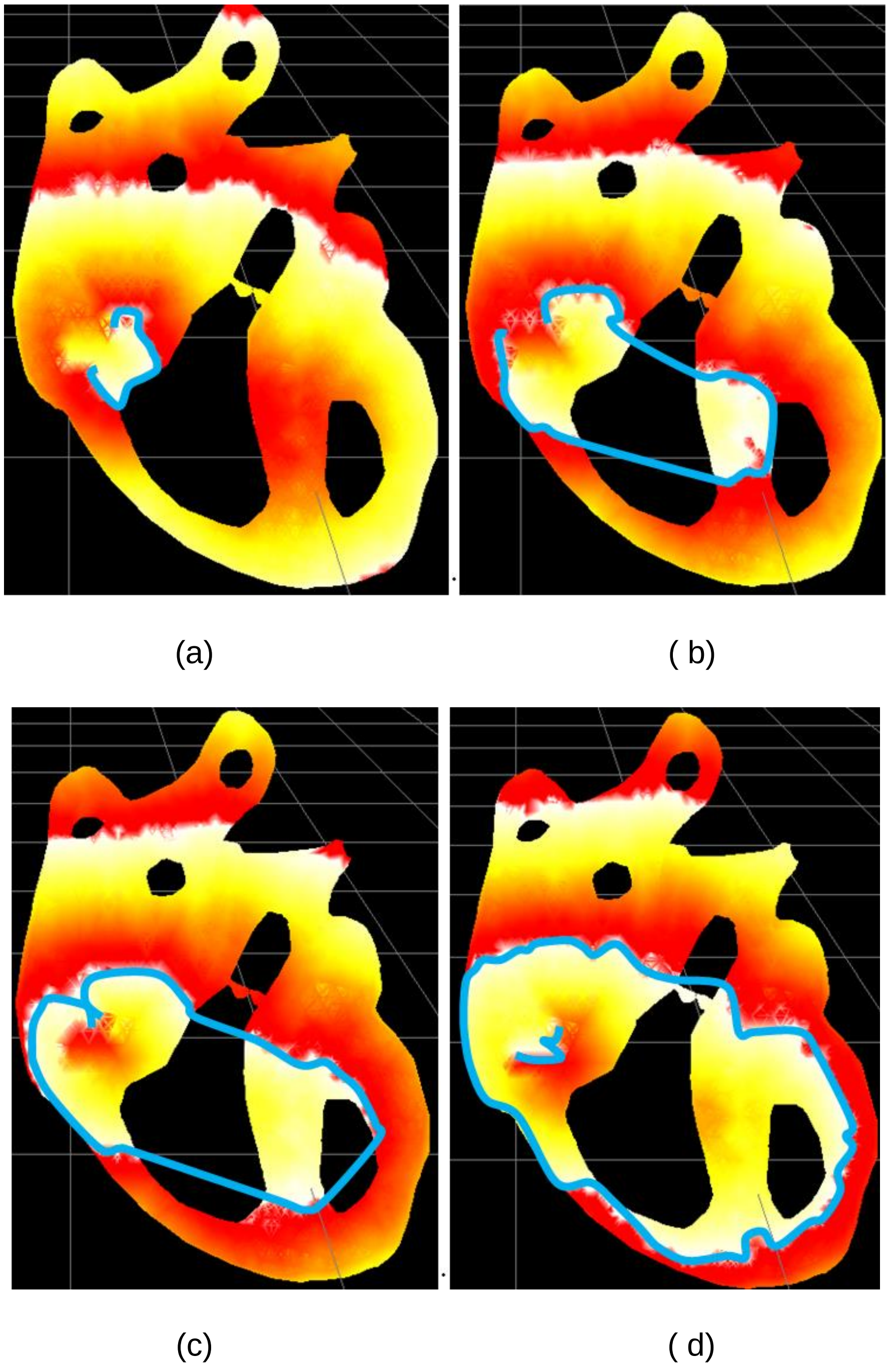


Figure 5: The heart is sliced to show spinal wave excitation. (a-d) the spinal wave is originated and circulating on the spot located at the middle left site inside the heart, which is made visible when the heart is sliced. The excited front of the spinal wave is marked with blue lines to show its progression and self-regeneration.

4.4.2 Performance Comparison

Even though the appearance of three different implementations is the same, the performances are drastically different. Figure 6 below showed the average time of calculating the status for all the points in the heart.

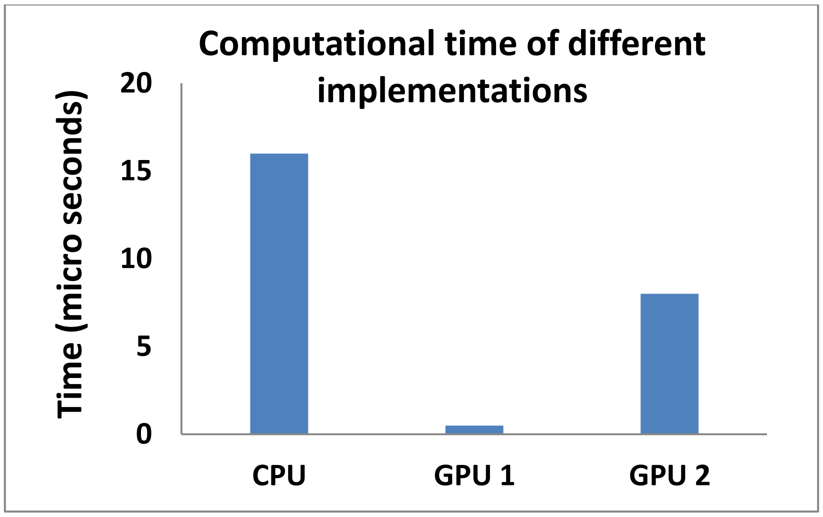


Figure 6: Performance comparison of three implementations

This simulation process for CPU implementation is relatively slow and can barely satisfy the realtime rendering requirement. The total time caused for the simulation to finish is approximately 16ms on our machine. The GPU 1 implementation launches one thread for each point and utilizes OpenGL/CUDA coop to save data transferring time. It has the best performance of an average of 0.5ms, which is 30 times faster than purely based on CPU. The GPU 2 implementation attempts to make use of the fast data transferring within shared memory. The overall performance of this implementation is not as fast as the first implementation of GPU, but still better than CPU implementation. An average of 8 ms was reported for this implementation. Shared memory did not accelerate the overall simulation process, possibly because most points has only approximately 30 neighboring points, and we launch 256 just to cover a few extremely crowed points. Therefore most of the points experience significant waste when the rest of the 200+ threads were not utilized. This behavior reduces possible benefit and prevents significant speed gain.

4.5 Conclusions

With modern massive parallel computation power of GPGPU, we have developed an effective 3D model of an anatomically realistic human heart to perform realtime simulation of cardiac electrical activities. This model is useful to demonstrate the natural formation of spiral wave, which indicates that such pattern of excitation behavior is an intrinsic character of the excitable media. It may be noted that this present work does not incorporate the micro-scale simulation of the ionic and molecular species. However, it is very possible to include such low level of modeling into the developed simulation platform. Given the rapid development in the field of GPGPU architecture, realtime, multi-scale, and comprehensive simulation of 3D human heart with unprecedented details and accuracy will be explored in future study.

REFERENCES

[1] Moore, Gordon E. (1965). "Cramming more components onto integrated circuits" (PDF). Electronics Magazine. p. 4. Retrieved 2006-11-11.

[2] New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies” – Fred Pollack, Intel Corp. Micro32 conference key note - 1999. Avi Mendelson, Intel.

[3] Kayvon Fatahalian, Running Code at a Teraflop: Overview of GPU Architecture, SIGGRAPH 2009: Beyond Programmable Shading

[4] A Kaufman, R Bakalash -Memory and processing architecture for 3D voxel-based imagery … Graphics and Applications, IEEE, 1988 - ieeexplore.ieee.org

[5] Adam Levinthal, Thomas Porter, Chap - a SIMD graphics processor SIGGRAPH '84 Proceedings of the 11th annual conference on Computer graphics and interactive techniques Pages 77 – 82

[6] N England A Graphics System Architecture for Interactive Application-Specific Display Functions Computer Graphics and Applications, IEEE, 1986 - ieeexplore.ieee.org

[7] S Gupta, RF Sproull, IE Sutherland A VLSI architecture for updating raster-scan displays SIGGRAPH Computer Graphics, 1981 - dl.acm.org

[8] England, J.N. A system for interactive modeling of physical curved surface objects. In Proceedings of SIGGRAPH 78 1978, 336-340. 1978.

[9] Potmesil, M. and Hoffert, E.M. The Pixel Machine: A Parallel Image Computer. In Proceedings of SIGGRAPH 89 1989, ACM, 69-78. 1989.

[10] Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U. and Varshney, A. Real-Time Procedural Textures. In Proceedings of Symposium on Interactive 3D Graphics 1992, ACM / ACM Press, 95-100. 1992.

[11] Trendall, C. and Steward, A.J. General Calculations using Graphics Hardware, with Applications to Interactive Caustics. In Proceedings of Eurogaphics Workshop on Rendering 2000, Springer, 287- 298. 2000.

[12] Wolfgang Heidrich. High–quality Shading and Lighting for Hardware – accelerated Rendering. PhD thesis, University of Erlangen–Nurenberg, April, 1999.

[13] Michael D. McCool and Wolfgang Heidrich. Texture shaders. In SIGGRAPH / Eurographics Workshop on Graphics Hardware, 1999.

[14] AMin´e and F Neyret. Perlin textures in real time using OpenGL. Technical Report RR-3713, iMAGIS–GRAVIR/IMAG/INRIA, 1999.

[15] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. Computer Graphics (SIGGRAPH '99 Proceedings), pages 277–286, August, 1999.

[16] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real–time robot motion planning using rasterizing computer graphics hardware. Computer Graphics, 24(4):327–335, August, 1990.

[17] Christian-A. Bohn. Kohonen feature mapping through graphics hardware. In 3rd Int. Conf.on Computational Intelligence and Neurosciences, 1998.

[18] Koji Yasuda (2008). "Accelerating Density Functional Calculations with Graphics Processing Unit". J. Chem. Theory Comput. 4 (8): 1230–1236. doi: 10.1021 / ct8001046

[19] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. (2008). "GPU acceleration of cutoff pair potentials for molecular modeling applications." In CF'08: Proceedings of the 2008 conference on Computing frontiers, New York, NY, USA: 273–282.

[20] Rio Yokota, Lorena A. Barta, "Treecode and fast multipole method for N-body simulation with CUDA" Chapter 9, GPU Computing Gems Emerald Edition (Applications of GPU Computing Series) Gems

[21] Mark Harris "Fast fluid dynamics simulation on the GPU" Proceeding SIGGRAPH 05 ACM SIGGRAPH 2005 Courses

[22] Dimitri Komatitscha,Gordon Erlebacherc, Dominik Göddeked, David Michéaa "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster", Journal of Computational Physics Volume 229, Issue 20, 1 October 2010, Pages 7692–7714

[23] Jerod J. Weinman, Augustus Lidaka, Shitanshu Aggarwal "large scale machine learning" GPU Computing Gems Emerald Edition chapter 19 2011

[24] Fang, Wenbin, et al. "Parallel data mining on graphics processors." Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS08-07 (2008).

[25] Stuart, Jeff A., and John D. Owens. "Multi-GPU MapReduce on GPU clusters."Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. IEEE, 2011.

[26] Won-Ki Jeong, Hanspeter Pfister, Massimiliano Fatica "Medical Image Processing Using GPU-Accelerated ITK Image Filters" Wen-mei, W. Hwu. GPU Computing Gems Emerald Edition. Morgan Kaufmann, 2011.

[27] Fung, James, and Steve Mann. "Using graphics devices in reverse: GPU-based image processing and computer vision." Multimedia and Expo, 2008 IEEE International Conference on. IEEE, 2008.

[28] Preis, Tobias, et al. "Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets." New Journal of Physics 11.9 (2009): 093024.

[29] Kim, Changkyu, et al. "FAST: fast architecture sensitive tree search on modern CPUs and GPUs." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010

[30] Han, Tianyi David, and Tarek S. Abdelrahman. "Reducing branch divergence in GPU programs." Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2011.

[31] Hou, Qiming, et al. "Memory-scalable GPU spatial hierarchy construction."Visualization and Computer Graphics, IEEE Transactions on 17.4 (2011): 466-474.

[32] Wong, Henry, et al. "Demystifying GPU microarchitecture through microbenchmarking." Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010.

[33] Bo Zhang, Xiang Yang, Fei Yang, Xin Yang, Chenghu Qin, Dong Han, Xibo Ma, Kai Liu, Jie Tian, "The CUBLAS and CULA based GPU acceleration of adaptive finite element framework for bioluminescence tomography," Opt. Express 18, 20201-20214 (2010)

[34] Ueng, Sain-Zee, Melvin Lathara, Sara Baghsorkhi, and Wen-mei Hwu. "CUDA-lite: Reducing GPU programming complexity." Languages and Compilers for Parallel Computing (2008): 1-15.

[35] Kayvon Fatahalian, Mike Houston "A closer look at GPUs." Communications of the ACM 51, no. 10 (2008).

[36] Volkov, Vasily, and James Demmel. "LU, QR and Cholesky factorizations using vector capabilities of GPUs." EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May (2008): 2008-49.

[37] Harris, Mark, Shubhabrata Sengupta, and John D. Owens. "Parallel prefix sum (scan) with CUDA." GPU Gems 3, no. 39 (2007): 851-876.

[38] Boyer, Michael, Kevin Skadron, and Westley Weimer. "Automated dynamic analysis of CUDA programs." In Third Workshop on Software Tools for MultiCore Systems. 2008.

[39] Nvidia CUDA programming guide

[40] Du D, Yang H, Norring SA, Bennett ES., "Multi-scale modeling of glycosylation modulation dynamics in cardiac electrical signaling." Conf Proc IEEE Eng Med Biol Soc. 2011;2011:104-7.

[41] E. Bartocci, E. M. Cherry, J. Glimm, R. Grosu, S. A. Smolka, S. A. Smolka, and F. H. Fenton, 2011, "Toward real-time simulation of cardiac dynamics," Proceeding CMSB '11 Proceedings of the 9th International Conference on Computational Methods in Systems Biology, 103-112

[42] FitzHugh R. (1969) Mathematical models of excitation and propagation in nerve. Chapter 1 (pp. 1–85 in H.P. Schwan, ed. Biological Engineering, McGraw-Hill Book Co., N.Y.)

[43] Barkley D. (1991) "A model for fast computer simulation of waves in excitable media". Physica D: Nonlinear Phenomena vol.49, Issues 1–2, pp 61–70, 1 April 1991.

[44] J. R. Silva and Y. Rudy, "Multi-scale electrophysiology modeling: from atom to organ," The Journal of General Physiology, vol. 135, pp. 575-581, June 01, 2010.

[45] E. V. Bondarenko and L. R. Rasmusson, "Simulations of propagated mouse ventricular action potentials: effects of molecular heterogeneity," Am J Physiol Heart Circ Physiol, vol. 293, pp. H1816-H1832, 2007.