Markov Chain Monte Carlo on the GPU

by

Michael D. Dumont

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Supervised by

Ivona Bezáková
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
March 2011

Approved By:

Ivona Bezáková
Professor, Department of Computer Science
Primary Adviser

Joe Geigel
Professor, Department of Computer Science
Reader

Zack Butler
Professor, Department of Computer Science
Observer

Contents

Abstract

Markov chains are a useful tool in statistics that allow us to sample and model a large population of individuals. We can extend this idea to the challenge of sampling solutions to problems. Using Markov chain Monte Carlo (MCMC) techniques we can also attempt to approximate the number of solutions with a certain confidence based on the number of samples we use to compute our estimate. Even though this approximation works very well for getting accurate results for very large problems, it is still computationally intensive. Many of the current algorithms use parallel implementations to improve their performance. Modern day graphics processing units (GPU's) have been increasing in computational power very rapidly over the past few years. Due to their inherently parallel nature and increased flexibility for general purpose computation, they lend themselves very well to building a framework for general purpose Markov chain simulation and evaluation. In addition, the majority of mid- to high-range workstations have graphics cards capable of supporting modern day general purpose GPU (GPGPU) frameworks such as OpenCL, CUDA, or DirectCompute. This thesis presents work done to create a general purpose framework for Markov chain simulations and Markov chain Monte Carlo techniques on the GPU using the OpenCL toolkit. OpenCL is a GPGPU framework that is platform and hardware independent, which will further increase the accessibility of the software. Due to the increasing power, flexibility, and prevalence of GPUs, a wider range of developers and researchers will be able to take advantage of a high performing general purpose framework in their research. A number of experiments are also conducted to demonstrate the benefits and feasibility of using the power of the GPU to solve Markov chain Monte Carlo problems.

1. Introduction

The purpose of this thesis is two-fold. First, work will be presented on the development of a framework for running generalized Markov chain simulations on the GPU. The intent of this framework is to develop a system that can easily be used by computer scientists and researchers in other fields. This also makes it easier to develop and write these simulations by abstracting away a lot of the development process that relates directly to the GPU hardware. Finally, the framework and benefits of GPU based Markov chain Monte Carlo simulation are benchmarked using a variety of experiments.

This thesis is of importance due to the fact that Markov chain simulations are used in a wide variety of fields from statistical physics and biology, to AI, and pure mathematics. Over the past few years, the GPU has evolved from a device designed for rendering geometry to a display into a highly parallel general purpose computational device. As the power of the GPU has been realized, more toolkits have been developed to allow general purpose access to the stream processing units on the device. This allows developers to perform very intense mathematical, data-parallel computations with incredible efficiency. We can see an example of a recent generation GPU and CPU compared to each other in Figure 1.1 [18, 4]. Despite these advances and the increased prevalence of the technology, there still has not been a significant research effort into developing a GPGPU based system for Markov chain problems. The development of such a framework would eliminate the need for developers working in related fields to rely on other specialized parallel architectures or environments. In addition, there has been increasing research into the development of graph based algorithms on the GPU. This can be seen in the works done by Vineet[20, 21], Harish[6, 20], Patidar[20], Narayanan[6, 20, 21], Hussein[8], Varshney[8], Davis[8], Vasconcelos[19] and Rosenhahn[19]. A variety of additional graph problems, such as matchings and colorings, can be formulated using Markov processes and such a framework will allow for further testing of these ideas.

	Core 2 Duo	NVIDIA GTX 285
Power	150 Watts	204Watts
# of Cores	4	240
Memory Bandwidth	8.5 GB/s	159 GB/s
Computation Power	45 GF/s	~1 TFLOP/s

Figure 1.1: A comparison of a recent generation CPU and GPU. [4]

2. Background and Related Work

2.1 Markov Processes

A Markov process is a stochastic process that exhibits the Markov property. The Markov property states that the current state of the process is only dependent on the state that it just came from and nothing that occurred before that point. Markov processes are commonly used in fields such as artificial intelligence, probability theory, and statistical sampling. We will primarily be concerned with the fields of probability and sampling for the scope of this thesis.

2.1.1 Markov Chains

Throughout this section we will follow the definitions and notations given by Jerrum[9]. A Markov chain can be defined simply as some set of states that make up our state space Ω and a transition matrix P that defines the probability of moving from one state to another. Since a Markov chain is defined to hold the Markov property we do not need to consider any history of transitions, the current state only depends upon the immediately preceding state. When we want to talk about transitions, we typically use one of two notations. $P(x, y)$ represents the probability of transitioning from state x to state y over the course of a single time step. If we use the notation $P^t(x, y)$ then we are referring to the probability of making this transition over the course of t time steps.

An interesting feature of finite Markov chains is the concept of stationary distribution. A stationary distribution is a probability distribution $\pi : \Omega \to [0, 1]$ that remains stable from one time step to the next. That is, it meets the following condition:

$$\pi(y) = \sum_{x \in \Omega} \pi(x) P(x, y)$$

We know that every finite Markov chain has a stationary distribution, but it is possible for a Markov chain to have multiple of these distributions. The only time that we have a unique stationary distribution is when certain properties hold for the Markov chain. The first is irreducibility. A Markov chain is considered irreducible if for all $x, y \in \Omega$ there exists a path from x to y and vice versa. What we mean by a path is that the probability of transitioning from state x to state y is non-zero over an infinite number of time steps. This ensures that our state space is fully connected and there is no way for us to lose access to any possible state. Another property we like to consider is aperiodicity. A Markov chain is considered aperiodic if it can return to a state at irregular intervals. Formally this is defined as:

$$\text{gcd}\{t : P^t(x,x) > 0\} = 1 \text{ for all } x \in \Omega \text{ and } t \in \mathbb{N}$$

We can ensure that a Markov chain demonstrates the property of aperiodicity by forcing each state to have a transition from itself to itself with some probability greater than 0. A Markov chain that demonstrates the properties of irreducibility and aperiodicity is known as an ergodic Markov chain. Ergodic Markov chains have been shown to guarantee a unique stationary distribution π. They also guarantee that $P^t(x,y) = \pi(y)$ as t approaches ∞ for every starting state x. This means that as we run our Markov chain it eventually stabilizes permanently at its unique stationary distribution. We define the practical amount of steps that we need to take to stabilize our Markov chain as the mixing time. This is often the amount of time it takes for us to approach the unique stationary distribution within a certain error tolerance. This tolerance can be quantified using a metric such as the total variation distance:

$$\parallel \pi - \pi' \parallel_{TV} = \frac{1}{2} \sum_{\omega \epsilon \Omega} |\pi(\omega) - \pi'(\omega)|$$

We can take samples from our Markov chain after this many steps and successfully approximate the probability distribution modeled by our Markov chain.

It may help to consider an example. Let us define a Markov chain on the binary strings of length two. This gives us a fairly small state space with only four states: 00, 01, 10, 11. We shall define the transition properties for these states as follows:

1. Remain at current state with probability 0.5
2. Choose the first bit and flip it with probability 0.25
3. Choose the second bit and flip it with probability 0.25

We can quickly see that this Markov chain is ergodic. Each state has a self loop which ensures the condition of aperiodicity and all of the states are connected in one component making it irreducible. This can be seen in Figure 2.1. This Markov chain converges to its unique stationary distribution after about 17 time steps (the stationary distribution also happens to be a uniform distribution across the four states in this case.)

Once a Markov chain has been developed to accurately model the probability distribution you are working with you can then use this as a tool to try to approximate the number of entities in your distribution.

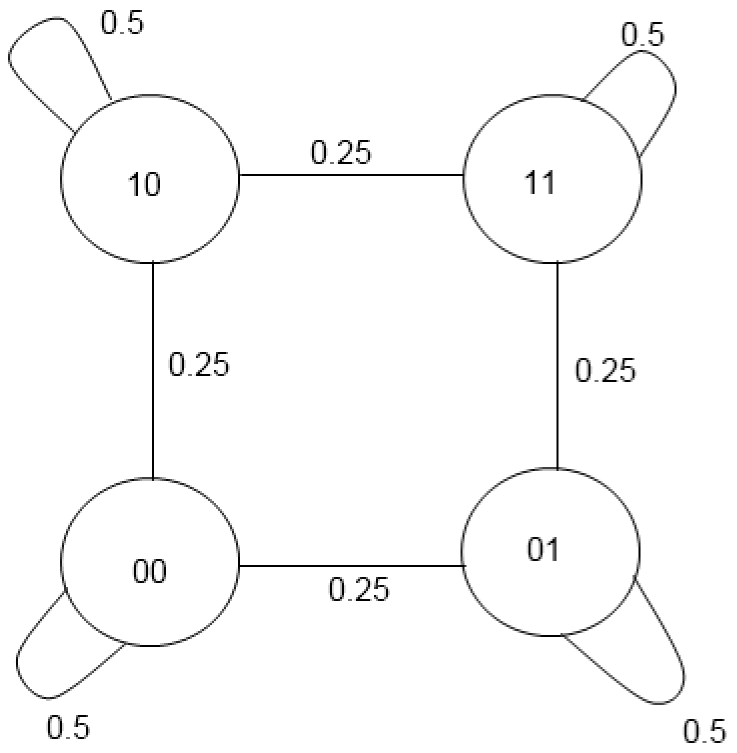

Figure 2.1: An example Markov chain on the binary strings of length 2.

2.1.2 Markov Chain Monte Carlo

Markov chain Monte Carlo refers to the concept of using Markov chains for random sampling of our state space as a tool for approximating the number of states that we have. Often times it becomes computationally complex for us to develop an efficient algorithm for deterministically counting the number of solutions to a problem. We call the class of counting problems the class of #P problems. A #P problem is more formally defined as counting all solutions to a given problem where a solution can be verified by a polynomial-time verifier. In addition, a #P problem is #P-complete if all other problems in #P are turing reducible to it. We will refer to the course notes provided by Bezáková[1] for the remainder of this section unless otherwise noted. Further analysis of this material is also provided by Jerrum[9].

A randomized approximation scheme is a randomized algorithm that can take an instance of a problem f along with an error tolerance and produce an approximation as to the number of solutions to the problem. More specifically, it will produce an output N that meets the following criteria (given an input problem x and an error tolerance ϵ):

$$Pr((1-\epsilon)f(x) \leq N \leq (1+\epsilon)f(x)) \geq \frac{3}{4}$$

The $\frac{3}{4}$ can be reduced down to a lower threshold by repetition of the algorithm. This is because you are able to generate a more statisically significant set of approximations with multiple applications of the algorithm.

We can extend this concept to a fully polynomial randomized approximation scheme. This is a randomized approximation scheme that runs in polynomial time based upon the size of the input problem x and the inverse of the error tolerance $\frac{1}{\epsilon}$.

Let us examine an example problem to demonstrate how we can use sampling as an approximation to counting. Assume we have some function $F(x)$ that returns a perfect matching on a graph G uniformly at random. We begin by selecting an edge e_1 in the graph and approximating the probability of a perfect matching containing e_1. This is done by sampling a number of perfect matchings from G in order to compute an approximation of the following ratio:

$$\frac{\# \ of \ perf. \ matchings \ on \ G \ including \ e_1}{\# \ of \ perf \ matchings \ on \ G}$$

As the number of samples goes to infinity we end up with a more accurate approximation of the above ratio. We can then define a new graph G_1 that is the graph G with the end points of e_1 removed from it (subsequently removing all of the incident edges). We select a new edge e_2 and repeat the same process to compute the following ratio:

$$\frac{\# \ of \ perf. \ matchings \ on \ G_1 \ including \ e_2}{\# \ of \ perf \ matchings \ on \ G_1}$$

We repeat this process until we have run out of edges in our graph (the only edges remaining are the ones that we selected at each stage) form a perfect matching, so we end up generating a final sample of 1). If we stop and consider that each time we remove an edge and generate samples from our new subgraph, we are actually computing the number of perfect matchings on our original graph that use the edges from the previous steps. This gives us a cancellation effect in our ratios when we multiply them together producing a final ratio of:

$$\frac{1}{\# \ of \ perf \ matchings \ on \ G}$$

This approach only holds if all of the ratios are bounded away from zero. This approach becomes very inefficient as the ratio approaches zero because the number of samples necessary to accurately approximate it becomes prohibitively costly. We can use a simulated annealing approach to achieve our estimation in an efficient manor. This approach has been demonstrated by Bezáková, Stefankovič, Vazirani, and Vigoda [2]. The basic idea is that the cooling schedule used in simulated annealing can represent the various aspects of a telescoping product, such as the one we are trying to compute for the matchings and colorings problems. The system starts out at a high temperature for the easier to compute parts of the telescoping product. As the system cools it represents the more difficult parts of the product to compute. This can be used to take a smaller number of samples from the system in order to compute our approximation.

A similar approach can be seen taken by Jerrum, Sinclair, and Vigoda [11] for approximating the permanent of a matrix with non negative entries.

2.2 General Purpose GPU Programming

The GPU was originally developed with the sole intent of processing geometry information so that it could be rasterized to the screen. GPUs started off by having a very fixed pipeline. A GPU would take in streams of vertex data and perform the exact same operations on each stream to generate the pixels to be displayed to the screen. Over time, languages such as Cg[14], GLSL[3], and HLSL[13] were developed. These languages were designed to allow developers to write their own custom programs that would be inserted at specific points of the graphics pipeline. This allowed developers to write their own shader programs that could run much quicker on the GPU than on the CPU. With this increased programmability, developers were also able to put arbitrary data into the vertex streams being passed to the GPU. They could write shader programs with the knowledge of this data and write the results of the computation to the screen. This output device is clearly not ideal as there is no way for the computer to easily retrieve the data from it, so developers began to take advantage of the ability to render to a texture object instead of the display. This allowed the data to be easily accessed by the program after the algorithm was done computing. Even more recently, tools such as OpenCL[5], CUDA[15], and DirectCompute[16] have been developed to allow programmers direct access to the GPU for general purpose computing. These tools remove the need for the developer to simulate data as vertices and use graphics features such as rendering to a texture object. [18]

2.2.1 GPU Architecture

The basic architecture of a GPU is based around the concept of stream processing. The general concept of stream processing is that each processor can rapidly run the same program over and over on a large set of data. The primitive structure in this model of computation is known as a stream of data, which is simply an array containing all of the elements in your stream. An overview of this architecture can be seen in Figure 2.2. A stream is a large collection of items of a common data type. We can perform a variety of operations on streams such as copying, generating subsets, and accessing the data in them to perform computations using programs known as kernels. When a kernel processes a stream it operates on the entire stream at once. This enforces some restrictions on what we are able to do with kernel programs. Firstly, kernel programs are only functions of their input streams and other data constants. Since we are processing the entire stream in parallel anytime we have to make computations depend on other elements in the stream we are losing efficiency. It is possible to use synchronization constructs to assist with this, but you always have to consider the tradeoffs being made. This enforces a very data-parallel structure to our programs which allows them to be optimized for the GPU computing architecture. This also allows the GPU to distribute an entire stream out to all of its stream processors at once since it has to synchronize any data dependencies in the stream[7, 18]

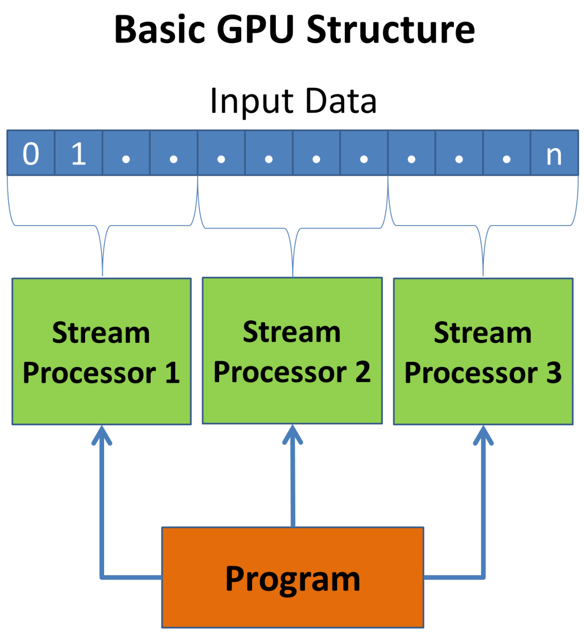

Figure 2.2: An overview of the GPU stream processing architecture

The GPU has two primary types of processors on it, vertex processors and fragment processors. Before looking at the workflow of a GPU, it is valuable to define a couple structures that are used in the rendering pipeline:

Vertex - An point in 3-dimensional space

Triangle - Defined as a set of three vertices. Used in meshes of 3D models to be rendered.

Pixel - An output in the final 2D projection of the 3D scene being rendered.

Fragment - Analogous to a pixel.

The typical simplified workflow of a GPU for graphics rendering is the following:

- 3D Models (comprising of triangles) undergo a series of matrix transformations that place them all in a ”world-space” in relation to each other. This takes each model from it’s own localized coordinate system into the global coordinate system of the scene we are trying to render.
- The vertex processor computes shading values for each vertex based on light sources, reflectance, and other surface properties defined for that vertex. These are interpolated between the 3 vertices of a triangle to compute the illumination data for the fragments/pixels that map to that surface.
- The world is normalized into ”view-space” where the origin is at the point of a virtual camera. This brings us one step closer to the final scene since we can now figure out which surfaces are visible, which are obscured, and what the depth of the object displayed by each pixel is.
- A projection transformation is then applied to the objects which puts them into clip space. This space is useful because it defines what objects will be visible when finally rendered to the display. We can remove these objects from any further processing.
- The next stage is the transformation into screen space. This scales the scene to the size of the viewport on the display. Now each vertex maps directly to a pixel.
- The rasterization stage takes the data at each vertex and interpolates it to compute a correct color value for every pixel in the output.
- The fragment processor gets a stream of pixels/fragments and computes any further operations on them such as texture mapping. These operations happen on a per-pixel basis and no knowledge of the original 3D scene geometry is available.
- The fragments are written out to a buffer that is sent to the display or stored into a texture object for other usage.

The OpenCL framework allows us to write our kernels in a variation of the C programming language known as OpenCL-C. We will write these as basic programs and we will setup input parameters that are bound to data we want sent to the GPU. [18, 4, 7] The process described here is simplified and more abstract than what is done for the 3D rendering pipeline. The basic idea is that we develop the function or kernel for the GPU, bind data into the parameters on the GPU, execute the function on these parameters in a streaming fashion (each set of inputs can be processed in parallel). Finally, we copy the memory off of the GPU to the host.

The basic workflow for GPGPU programming in OpenCL is the following:

- Identify the OpenCL compatible device you want to use.
- Create a context to the device and set up a command queue.
- Create a program on the device with the source code for your kernel, build the program, and instantiate the kernel.
- Create the buffers on the GPU for your inputs and bind the data to them. This does the copy of the data into GPU memory.
- Create the buffers on the GPU for your outputs. We do the bind after execution because this will copy the result data off of the GPU back to our host memory.
- Set each buffer to the appropriate kernel argument index. This binds the buffers in the GPU memory to the specific arguments to your kernel function.
- Execute the kernel with the number of instances you want running.
- Bind data back from the GPU.
- Cleanup your OpenCL structures.

This model binds the GPU kernel that you created into the graphics pipeline, but instead of rendering the result of the kernel to the display buffer, it gets rendered to the memory buffer you allocated so that you can extract the result back to the host for further processing and analysis.

2.2.2 GPU Techniques

There are a number of different patterns that we see common in GPGPU programming. This section will present a quick overview of these operations.

Map - The idea of mapping is to take an input stream and generate an output stream of data by applying the same function to each item in the original stream.

Reduce - Reduction allows us to apply an operation to a stream that will compute a single element (or smaller set of different outputs) as a result of combining the elements of the input.

Expand - The expansion of a stream allows us to generate a new output stream with multiple output elements for each input element.

Stream filtering - Stream filtering is similar to reduction, but the idea is that we can reduce a stream down to a smaller size, not necessarily one single value.

Scatter - The process of scattering refers to the ability of the vertex processor to place the data that it is receiving into random-access memory.

Gather - The gathering operation refers to the fact that the fragment processor can read textures from any position it chooses so it can collect its data from anywhere in the grid.

Sort - Sorting on the GPU is commonly implemented as a sorting network. These networks allow us to pass the data through a clear predefined set of comparisons and result in a sorted set.

Search - Searching on the GPU allows a user to find an element in the stream. Instead of speeding up the search via the GPU, we use the GPU to search for the item in parallel.

[7, 18]

The techniques used in this work are primarily map and reduce. We take a stream of inputs and reduce it to a stream of outputs, we then take these outputs and combine them in our final answer. In our work, the reduction will be done off of GPU since it is a trivial combination that will not benefit significantly from being performed on the GPU.

2.2.3 Performance Benefits and Issues

The performance benefits of the GPU come from the fact that it is a highly data-parallel device containing a large number of stream processors. This allows it to process large amounts of data through the same set of operations very efficiently. The cost that comes with this is the latency in moving data into the GPU memory and then back to main memory at the end of the computation. We define the efficiency of GPGPU programs with the term arithmetic intensity. Arithmetic intensity is defined as the number of operations per word of data transferred. It is important for our programs to have high arithmetic intensity so that we can overcome the cost of data transfer, making our program worth being implemented on the GPU. It is also important to realize that due to the extreme data-parallel architecture of the GPU there are some problems that will be significantly more difficult to implement for this model of computation. In addition there is also some overhead associated with setting up and instantiating the kernel on the device. This means that your problem should also be sufficiently large to make it worth running on the GPU.[18]

The performance benefit gained by stream processing is also based on the idea of single-instruction multiple data (SIMD). The idea of a SIMD architecture is that we can process everything in a vector and apply that one hardware instruction to multiple parameters at once. This architecture breaks down if you have programs that involve significant amounts of branching. Various constructs exist to help alleviate these restrictions. One way to reduce branching is to simulate fake data. If the function you are trying to execute can handle the idea of some identity data in it you can pad array sizes to reduce branching introduced by irregular loops. Another feature that is provided is the idea of a barrier. A barrier is a function call you can place in your kernel. This call will block until every kernel instance reaches it. This is useful after a loop of irregular size because you can rejoin your kernel instances to be executing the same instructions simultaneously again. Issues can arise if you place this in branching logic though since not every kernel will necessarily reach that branch. This will cause the kernels that hit the barrier to halt entirely.

2.2.4 Existing Tools

Today, there are three primary tools that have been developed for GPGPU programming. These tools allow the user to write basic GPU programs that can execute on the GPU and return data without having to interact with the older workaround model using the graphics libraries. The three primary tools that exist currently are DirectCompute by Microsoft, CUDA by Nvidia, and OpenCL which was originally created by Apple, but now is maintained as a specification by the Khronos group. Each of these tools has its own distinct advantages and disadvantages.

CUDA was one of the first GPGPU tool kits created. Due to this, it has the advantage of better driver stability, more support, a more optimized compiler. However, CUDA will only work on Nvidia hardware, which limits the accessibility of CUDA programs. Microsoft's DirectCompute is the most recent competitor in the field. It shipped with DirectX 11 in late 2009, but it is also supported on DirectX 10 hardware. DirectCompute is hardware independent, but is tied specifically to the Microsoft platform which also limits the accessibility of software written that uses it. Finally, this leaves us with OpenCL. OpenCL has solved a lot of the problems mentioned previously regarding CUDA and DirectCompute. It is hardware and platform independent which allows for the greatest amount of accessibility for developers and users. OpenCL exists as a specification and has been implemented on the major graphics platforms.

2.3 Graph Theory

The subject of graph theory is the study of pairwise relationships and connections between various objects. The field of graph theory has a wide variety of applications such as path finding, network flow and structure, and matching problems among many others. Graph theory covers a very wide area of material. In the context of this proposal I will be presenting the concepts, terms, and definitions and are of immediate relevance to the work being proposed. Except where otherwise noted we will refer to the definitions and results given by West [22].

A mathematical graph G consists of a set of vertices V and a set of edges E where each edge $e = \{u, v\}$ where $u, v \in V$. The vertices represent the objects that we are concerned with and the edges represent relationships between these objects. In the context of this thesis we will only be concerned with simple weighted and unweighted graphs. A simple graph is defined as an undirected graph that contains no self loops (edges from a vertex back to itself) or multiple edges between the same pair of vertices. Undirected means that an edge between u and v forms a bidirectional relationship. Some examples of graphs can be seen in Figure 2.3. In addition to this, we can also add weights to our relationships. An example of this might be used in a graph that is representing the distances between various locations. The locations would be listed as vertices of the graph and the edges would represent which locations are reachable from each other with weights that show the distance between them. Two vertices are considered adjacent if an edge exists connecting them. An edge and vertex are considered incident if they are connected.

Graphs often times can have separate disconnected components. An undirected graph consists of multiple components if there are multiple sets of vertices that are not connected to each other in some way via the edges of the graph. If we have some graph G and we can locate an edge or vertex whose removal will cause H to disconnect into multiple components then we refer to this edge/vertex as a cut-edge or cut-vertex. The bottom left graph in Figure 2.3 is an example of a graph with multiple components. It also has a cut-vertex and cut-edge highlighted in red.

When we talk about the connectivity of a graph, we oftentimes use the term k-regular. This term simply states that every vertex v in a graph G is connected to k other vertices in the graph. We can also talk about the term bridgeless. This term refers to the idea of edge-disjoint paths. Specifically, it refers to the fact that there must be two edge-disjoint paths between any pair of vertices in the graph. This allows us to see that there are no individual cut vertices or edges in our graph. The Petersen Graph, shown in the top right of Figure 2.3, is 3-regular and bridgeless. More information regarding regular graphs and their properties can be seen in the work done by Petersen [17].

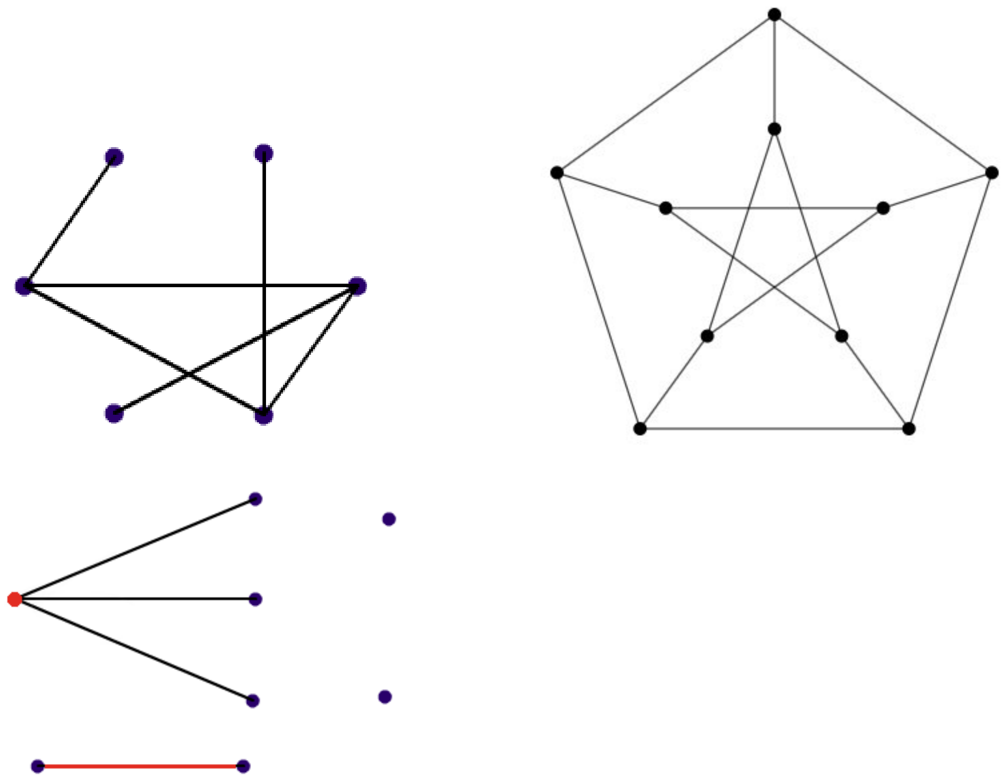

Figure 2.3: A basic graph on the top left, the Petersen graph in the top right, and a multi-component graph on the bottom left.

2.3.1 Matchings

The concept of a matching is the idea that we want to find a set of one-to-one pairings between vertices in our graph. This is formally defined as a set of pair-wise, non-adjacent edges. This means that we end up with a set of edges that have no common vertex between any of them, giving us the one-to-one matching that we were initially seeking. Matchings can range in size from zero edges to $\lfloor N/2 \rfloor$ where N is the number of vertices in the graph. If we have a matching that is of maximum size we call this a perfect matching. This is because it successfully pairs each vertex with another vertex. Figure 2.4 demonstrates an example of a perfect matching.

2.3.2 Colorings

The term graph coloring refers to a special set of problems where we are trying to label the vertices of a graph with a restricted set of colors. A valid coloring of a graph is one where each vertex is labeled with a color and no two adjacent vertices share a color. We refer

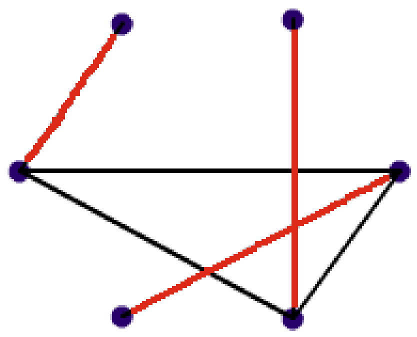

Figure 2.4: A graph showing a perfect matching.

to the minimum number of colors needed to color a graph in such a way as the chromatic number of the graph, $\chi(G)$. Graph coloring problems have many applications in scheduling, register allocation, and Ramsey theory. For example, if we wanted to compute the smallest amount of time it would take to complete a set of jobs, we would put each job as a vertex on a graph. Jobs that conflict with each other would be connected by an edge. The minimum amount of time needed to complete these jobs is equivalent to the chromatic number of the graph. You can see an example of a minimum colored graph in Figure 2.5.

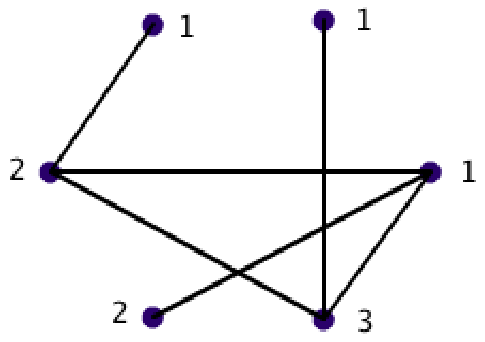

Figure 2.5: A basic graph showing a proper minimum coloring of three colors.

3. GPU Framework Goals & Architecture

3.1 Goals

When developing this GPU framework, there were several goals we had in mind in terms of simplification for the developer.

1. Reduce the need for the developer to understand the intricacies of GPU hardware.
2. Develop reusable code that limits or removes the need for the developer to interact directly with the GPU set up routines.
3. Allow for simulation of Markov chains for sampling or for reduction to approximate counting.

3.2 Architecture

The generalized framework presented here has been designed for the simulation of Markov chains and MCMC problems on the GPU. The system has been designed using the OpenCL framework in C for GPGPU computation. This allows the framework to be hardware and platform independent and function on the widest range of hardware that is easily accessible to other researchers.

The framework provides an interface that allows us to instantiate a context and all of the device information while only providing a string containing the kernel source code and the device type that we would like to execute on. Once this is done, you can use another method to register a pointer to an input parameter to the actual parameter in your OpenCL kernel. This is done by providing the memory address and size along with the parameter number in the OpenCL kernel. Finally, we have the interface for executing our kernel. We need to provide pointers to blocks of memory that the output should be stored in along with how many kernel instances we need to be instantiated.

The stream processing features of a GPU are taken advantage of by starting a large number of Markov chains in parallel for our Monte Carlo estimations. Each Markov chain is created as an additional element in the stream and they are all sent over to be processed in parallel. The Markov chains can also be split into sub-divided chains for sampling. As long as the chain is iterated for it's full mixing time between each sample then it will model the same distribution regardless of starting point. This means that even if we just want to do sampling we can take advantage of the parallelization on the GPU.

As you can probably see, this framework can be easily repurposed for general abstraction of the OpenCL host interface. The only thing that it doesn't really abstract away is the actual requirement to still write your OpenCL-C program.

3.2.1 APIs

In this section, we will define a basic list of the APIs that this framework provides.

int initialize(char *program, int size) - This function takes the source code of the OpenCL-C program for your kernel, compiles it, and initializes the various OpenCL structures that the library depends upon, returning an error code upon failure. It also takes in the size of the problem, which is the number of kernel instances it will run (and also the expected input and output size).

int bindInput(void* data, int paramNum, int sizeOfData) - This function takes a block of memory and a parameter number and sets it up to be bound onto the GPU as an input parameter to the kernel you have specified. This call must be committed with finalizeBinds, which should be invoked after all of the parameters have been bound.

int bindOutput(void* data, int paramNum, int sizeOfData) - This function takes a block of memory and a parameter number and sets it up to be bound onto the GPU as an output parameter to the kernel you have specified. This call must be committed with finalizeBinds, which should be invoked after all of the parameters have been bound. After the kernel has been executed, this pointer will be populated with the output of the kernel execution.

int finalizeBinds() - This should be used after you have set all of your bindings. It takes the pointers that you have created, commits them to the GPU, and then tells the GPU to bind those addresses of memory on the GPU to the various arguments. It will return an error code on failure.

int executeKernel() - This command will tell the GPU to execute the kernel you created during the initialize on the various arguments you bound to the function. It will populate the pointer to your output data with the resulting data, or return an error on failure. This call will block until the results are generated.

int cleanup() - This command will cleanup the various OpenCL structures, it should be used after execution of the kernel. It will not free the data that you passed in as inputs or that it returned as outputs. It is the caller of these API's responsibility to free those blocks of memory.

3.2.2 Example workflow

An example workflow using this framework might look like the following:

1. Generate data needed as input to the kernel.
2. Initialize the kernel with your source code and the size of your problem.
3. Bind the various inputs (mixing time, graph instances, random number seeds, etc) to the parameters.
4. Bind the output pointers to the parameters.
5. Finalize the bindings.
6. Execute the kernel. This is the actual Markov chain simulation implemented in OpenCL-C.
7. Cleanup the memory used by the framework.
8. Perform any post processing on the output of your kernel.

This is a very simple workflow by design. The goal of the framework was to abstract as much of the OpenCL interaction away from the user as possible. In this case, you just have you setup your instance, register your parameters with the system, and give it a run to get your results. The complex part of the interaction will probably be more along the lines of building your data to send into the kernel.

As an example from our colorings experiment, we have to take a graph instance and perform the following steps on it:

1. Generate a valid q-coloring on the graph.
2. Decompose each graph by removing an edge.
3. Compute other factors such as mixing time and sample count.

4. Experiments

In order to test the framework the following example problems have been implemented on top of it. These problems were chosen due to the fact that they are well researched problems. This will allow us to develop accurate deterministic solutions for these problems to give us a baseline for evaluation of the system.

For all of our experiments, we implemented them using several conventions. Firstly, a single kernel instance in our experiment is iterated for one mixing time and computes a single sample for our reduction. This means for every subgraph that we had in our reduction we needed to run a separate kernel instance. This was done for performance and stability reasons described later. Since we are running a random process on the GPU, we end up introducing a bit of branching. The most notable case is the fact that we are randomly selecting nodes and iterating over their set of neighbors. We use an OpenCL method known as a barrier to resynchronize our kernels. A barrier works in a similar fashion to a semaphore. It is instantiated at a value equivalent to the number of kernel instances that are being executed. It requires all kernel instances to hit the barrier before allowing any of them to continue on. This means you need to make sure only to use barriers in code segments that all kernel instances are guaranteed to execute.

4.1 Graph Colorings

The test application in this section is a Markov chain that is designed to sample the set of all K colorings on a graph G. This Markov chain is defined as follows by Jerrum[9].

1. Select a vertex $v \in V$, uniformly at random.
2. Select a color $c \in Q \backslash X_0(\Gamma(v))$, uniformly at random.
3. $X_1(v) \leftarrow c$ and $X_1(u) \leftarrow X_0(u)$ for all $u \neq v$.

where the terminology, Q defines our set of colors, X_0 defines our current coloring, X_1 is the coloring we are attempting to construct, $\Gamma(v)$ represents the neighbors of vertex v, and $X_0(\Gamma(v))$ represents the set of colors of the neighbors of vertex v.

This Markov chain selects a random vertex, picks a color from the set of other valid colors it could change to, and creates a new coloring by just changing that one vertex. Jerrum has demonstrated this Markov chain to be rapidly mixing in polynomial time based on the size of the problem and the precision of the results we are computing[9]. This gives provably correct estimates as long as $q \geq 2 * \Delta + 1$, where Δ is the maximum degree of our graph. It is not known if the estimates are correct in polynomial time when constrained by using a lower value of q.

4.1.1 Reduction of Approximate Counting to Sampling

In order to reduce the problem of approximating q-colorings to this Markov chain, we will apply the reduction as follows.

1. Remove an edge e from the graph.
2. Generate the specified number of samples on the graph.
3. Compute the ratio of the number of colorings where the two vertices incident to e are properly colored to the number of samples taken.
4. Repeat until all edges have been removed from the graph.

We then take the product of these ratios and use this to compute a percentage of q-colorings on a completely disconnected graph of n nodes.

We have broken this problem into a GPU kernel that runs independently on each graph that is created by the edge removal process. The result of this kernel is a stream of ratios to be multiplied together for the final approximation. Each kernel instance will take a sub-graph and generate the samples based on the mixing time in order to compute the specific ratio for that subgraph. This output stream of ratios is what is then used to generate our final approximation to the problem.

4.1.2 Implementation Pseudocode

The full implementation can be found on the CD included with this document.

```
__kernel void mcmckernel(__global Graph *input,
                         __global uint *random_seed,
                         __constant uint *q,
                         __constant uint *mixingTime,
                         __constant uint *sampleCount,
                         __global uint *node1,
                         __global uint *node2,
                         __global ulong *answer,
                         )
```

```
{
    //the gid is used to index the streams for our instance
    int gid = get_global_id(0);
    Graph g = input[gid];
    uint n1 = node1[gid];
    uint n2 = node2[gid];
    uint properColorings = 0;
    for (int i = 0; i < *mixingTime * *sampleCount + 1; i++) {
        select random color
        select random node

        change node to new color
        if (valid) {
            do nothing
        } else {
            revert color change
        }
        //if we have iterated for our mixing time
        //and the edge we are checking in this instance
        // is properly colored, we should increment our count
        if (i % mixing time) {
            if (node1 color != node2 color) {
                properColorings++;
            }
        }
    }
    answer[gid] = properColorings;
}
```

This is a basic pseudo code of the implementation that does not have any memory synchronization constructs. Those will be detailed in a later section. The random number generation is using a basic LFSR off of the random seed that is input for each instance.

4.2 Graph Matchings

This test application is designed to sample the set of all matchings on a graph G. We define the Markov chain for this application identical to how it is defined by Jerrum[9].

1. Select $e = \{u, v\} \in E$ uniformly at random.

2. There are three possibilities:
   1. If u and v are not covered by our matching X_0, then $M \leftarrow X_0 \cup \{e\}$.
   2. If $e \in X_0$, then $M \leftarrow X_0 \backslash \{e\}$.
   3. If u is uncovered and v is covered (or vice versa) by some edge $e' \in X_0$, then $M \leftarrow X_0 \cup \{e\} \backslash \{e'\}$.

3. If none of the above situations occur, then $M \leftarrow X_0$.

where the terminology X_0 defines our current matching and M defines the matching that we are attempting to construct.

As you can see, this is a simple Markov chain that allows us to transition through our possible matchings by adding edges, removing edges, or in the case where we want to add an edge that is partially covered, sliding edges.

This Markov chain has been demonstrated to be rapidly mixing in polynomial time based on the size of the problem and the precision we are computing our results within. This was shown by Jerrum and Sinclair[10].

4.2.1 Reduction of Approximate Counting to Sampling

We use this Markov chain to approximate the number of matchings in a similar way to how we approximated colorings.

1. Select an edge e from our graph.

2. Compute a ratio of the number of matchings containing e to the number of samples taken.

3. Remove both vertices that are incident to e from the graph.

4. Repeat the process until no edges remain to be selected.

5. Multiply these ratios together and we get the inverse of the number of matchings on the graph.

We implemented this Markov chain on GPU kernel in a similar fashion to the Markov chain on colorings. It runs independently on each graph that is generated by removing the vertices and then outputs a stream of ratios to be used in our final approximation. Jerrum[9] has shown that each of these sub-ratios is bounded away from 0. Specifically, by showing that the set of matchings on a subgraph G_{i-1} is a proper subset of the matchings on the graph G_i it has been shown that this ratio is bounded between $\frac{1}{2}$ and 1.

4.2.2 Implementation Pseudocode

The full implementation can be found on the CD included with this document. This implementation follows a very similar structure to the colorings example.

```
__kernel void mcmckernel(__global Graph *input,
                         __global uint *random_seed,
                         __constant uint *q,
                         __constant uint *mixingTime,
                         __constant uint *sampleCount,
                         __global uint *edge,
                         __global uint *node2,
                         __global ulong *answer,
                         )
{
    //the gid is used to index the streams for our instance
    int gid = get_global_id(0);
    Graph g = input[gid];
    uint e = edge[gid];

    uint properColorings = 0;
    for (int i = 0; i < *mixingTime * *sampleCount + 1; i++) {
        select a random edge
        if (both nodes on the edge are unmatched) {
            add the edge to the matching
        } else if (the edge is already in the matching) {
            remove the edge from the matching
        } else if (one node of the edge is matched) {
            remove the edge that is matching it
            add the current edge
        }


        //if we have iterated for our mixing time
        //and the edge we care about is currently
        //in the matching we should increment
        if (i % mixing time) {
            if (node1 color != node2 color) {
                properColorings++;
            }
        }
    }
    answer[gid] = properColorings;
}
```

5. Results

The results generated by the work presented here show several different things.

1. Your problem must be of substantial size to benefit on the GPU.
2. Special care must be taken with Markov chains that use irregular data structures, or anything else that can introduce branching.

Computing the solution to a problem on the GPU does encounter some overhead that requires the problem to be of a significant size to be able to truly benefit from the performance gains. This overhead occurs in two types. The first case is the time necessary to transfer all of the data to your kernel on the GPU. The second case is the fact that it does take some time to compile and initialize a kernel to run on the GPU.

Performance issues can also be introduced by unintentional branching on the GPU. Branching can occur for a variety of reasons, but most commonly in our kernels, was the case where we had a looping structure that ran for different iterations in each kernel. An example of this is looping over the neighbors of a randomly selected vertex in a graph. This breaks the basic SIMD structure of the program and reduces the efficiency. Several possibilities for branch minimization or correction are discussed later on.

All of these experiments were run on the following platform specifications detailed in Figure 5.1. The runtime results are the average of 5 experiments for each data point. The system was isolated as much as possible during the experimentation runs. No extraneous processes were running, network connections were turned off to prevent system services from reaching out to start activities, and as many services as could safely be stopped were stopped.

5.1 Graph Colorings

Our implementation was run on a set of experiments and compared against a reference implementation on the CPU developed by Tobias Kley[12]. We can see here that the smaller problems, when run in a parallel implementation on the GPU, performed only slightly better than the implementation on the CPU. As these problems grew in size they started to perform significantly better on the GPU than on the CPU. All of these experiments were executed on graphs with an error tolerance $\epsilon = 0.15$ and a q value of 11. If we recall

CPU	2.66 GHz Intel Core i7 (Dual-core)
Memory	4 GB 1067 MHz DDR3
GPU	NVIDIA GeForce GT 330M
Operating System	Mac OS X (10.6.6)

Figure 5.1: Platform Specifications

Experiment #	$\|\|V\|\|$	$\|\|E\|\|$	CPU Runtime	GPU Runtime
1	10	10	0.089 min	0.0047 min
2	20	20	0.294 min	0.0068 min
3	30	30	4.639 min	0.0473 min
4	40	40	8.637 min	0.124 min
5	50	50	46.551 min	0.187 min
6	100	100	$>$ 3 hours	”unknown”
7	500	500	$>$ 3 hours	”unknown”
8	1000	1000	$>$ 3 hours	”unknown”

Figure 5.2: Results of our experiments on graph colorings.

the definition of a randomized approximation scheme, this means that our output will be between 0.85 and 1.15 of the actual number of solutions to the problem $\frac{3}{4}$ of the time.

We can see the results of our experiments in Figure 5.2. Our first experiments performed better on the GPU, but the CPU run time was insignificant enough that this did not really demonstrate anything meaningful. As we hit our third, fourth, and fifth experiments we started to see significant speed improvements when running on the GPU. Our final experiments ran into issues on the GPU due to the execution time that a single kernel instance took. When all of the GPU cores were being monopolized for a significant amount of time the GPU kernels were actually killed off early so that control could be regained by the system.

5.2 Graph Matchings

We found that the results for the graph matchings problem closely matched the results of the colorings problem. Similar performance gains were seen to the graph colorings.

Experiment #	$\|V\|$	$\|E\|$	GPU Runtime
1	10	10	0.0056 min
2	20	20	0.0089 min
3	30	30	0.0568 min
4	40	40	0.105 min
5	50	50	0.196 min

Figure 5.3: Results of our experiments on graph matchings.

6. Future Work

6.1 GPU Interface

The biggest challenges presented by using the GPU are the following:

1. Lack of support for irregular data structures.
2. Random number generation.
3. Long running kernels.
4. Kernel dependencies.

One attempt at solving the first problem presented would be to investigate APIs provided by other GPGPU frameworks such as Nvidia's CUDA or Microsoft's DirectCompute. The issue is that in order to provide a structure of some kind to the GPU as part of a stream, you have to have the data of that structure all contained within that structures memory space. You cannot have any pointers to other dynamically allocated pieces of memory. This poses an issue, for example, when trying to specify graph structures to the GPU, because you cannot have a dynamically generated number of nodes or edges. One possible solution would be the implementation of a function with the following prototype:

```
(void*) gpuMalloc(uint size);
```

This function would allocate a piece of memory on the GPU and return a pointer to that memory on the GPU. You could then populate this memory with the data you need and store it in your structure. The performance implications associated with such a high level of random access memory would need to be investigated as well.

The random number generation system implemented for these experiments is a basic linear feedback shift register. Each kernel has their own that is started from a different seed provided by the host. These seeds are based off of the system time of the host device. It would be interesting to pursue development of a more uniformly random generator that could be used by multiple instances of the kernel in parallel. This would provide a higher degree of randomness to our samples and would ensure that randomness over all of the entries within our stream.

It was discovered, and confirmed by other projects, that having a single kernel instance that needs to execute for a significant amount of time will cause issues on the GPU. Notably, when all of the stream processors are being monopolized for a significant amount of time the GPU ends up freeing up the processors under the assumption that a kernel has entered some non-returnable state. This could potentially be solved by breaking up the mixing time in such a way that for one sample, we have a single kernel instance compute some percentage of the mixing time, and then another kernel continues after that one has finished. This is not an ideal solution to the problem however, because it breaks that data parallel nature of the GPU by causing a lot of dependencies between kernel instances.

6.2 Framework Improvements

The goal of the GPGPU framework developed here was to help abstract away the need for knowledge of GPU architecture and programming for the user. This goal was only partially successful, due to a number of reasons. First and foremost, in order to truly abstract the GPU away you would need to eliminate the need for a developer to write GPU kernels in languages like OpenCL-C. This could potentially be accomplished by implementing some sort of simpler language that is designed for describing Markov Chains and then cross-compiling that language into OpenCL-C. In the event that you already know your entire state space and you just want to simulate the Markov Chain without doing any approximation of it, you could easily pass in a data structure representing the state space and have a generic GPU kernel exist to process that. This is an extremely limiting and less useful scenario however, and does not take enough advantage of the GPU to make it worth it.

Related to the requirement that developers be able to write GPGPU code for their simulations, they also need to understand how to optimize it in certain scenarios so that you don't end up causing a serialization of instructions running on the GPU. The primary issue here arises whenever you introduce branching operations into your code. Since the GPU is designed as a very data-parallel architecture, it performs much better when all of the kernels are operating the same instruction at the same time. The issue we see with Markov chain Monte Carlo simulations is that we are often times making random selections and then looping over irregular amounts of data based on these selections. An example would be randomly selecting a node in a graph and then looping over its neighbors. OpenCL-C provides one feature to help correct these situations called a barrier. If you place a barrier in your code then it will require that al kernels reach that barrier point before allowing any kernel to proceed. This allows them to re-synchronize their instructions and resume operating in a vectored fashion. When working with barriers however, you must be careful to ensure that all of the kernels will reach each barrier. Therefore they should not be placed in conditional statements and only within loops that are guaranteed to run the same amount of time. An example can be seen below. The pseudo code for verifying whether an instance is correctly colored is:

```
for (Node n2 in neighbors(n1)) {
    if (n2.color == n1.color) {
        mark invalid;
        break;
    }
}
```

We can clearly see that this for loop may have an irregular number of iterations across kernel instances. This will cause the instructions across the instances to fall out of synchronization, which will result in the aforementioned performance loss.

We can rectify this using the barrier function in OpenCL:

```
for (Node n2 in neighbors(n1)) {
    if (n2.color == n1.color) {
        mark invalid;
        break;
    }
}
barrier(CLK_LOCAL_MEM_FENCE);
```

This will ensure that all kernel instances reach this point before continuing and that the instructions have been resynchronized across the instances.

7. Conclusions

While the GPU is definitely a viable platform for Markov chain Monte Carlo approximation algorithms, the performance benefits are only really seen for problems of a significant size, and stability issues are encountered when you begin to provide problems that take a significant amount of time to iterate over.

Care must also be taken when developing kernels for the GPU to ensure that they are designed to perform cleanly on the architecture. These details increase the complexity of abstracting away the entire developer interaction with the GPU, requiring some sort of pre-processing stage to minimize branching. Stability issues were also encountered when working on problems of a significant size. Care had to be taken to ensure that the kernel was not monopolizing the set of stream processors for too long in the event that the GPU decided that a kernel had entered a non-returning state.

The results shown here demonstrate that the GPU does indeed improve the performance via parallelization for Markov chain simulations and Monte Carlo approximations for sufficiently large problems and that it is possible to develop a basic framework for abstracting the OpenCL host code.

Bibliography

[1] Ivona Bezáková. Advanced algorithms: Markov chain monte carlo. `http://www.cs.rit.edu/~ib/Classes/CS801_Winter09-10/assignments.html`, 2009-2010. Course Notes.

[2] Ivona Bezáková, Daniel Stefankovič, Vijay V. Vazirani, and Eric Vigoda. Accelerating simulated annealing for the permanent and combinatorial counting problems. In *In Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 900–907. ACM Press, 2006.

[3] Khronos Consortium. Opengl shading language. `http://www.opengl.org/documentation/glsl/`. Product.

[4] David Gohara. Open cl tutorial - introduction to opencl. `http://www.macresearch.org/opencl_episode1`, August 2009.

[5] Khronos Group. Opencl. `http://www.khronos.org/opencl/`. Product.

[6] Pawan Harish and P.J. Narayanan. *Accelerating large graph algorithms on the GPU using CUDA*. Lectures Notes in Computer Science. Springer Berlin, 2007.

[7] Mike Houston. General purpose computation on graphics processors (gpgpu). `http://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf`, 2005.

[8] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on cuda. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, October 2007.

[9] Mark Jerrum. *Counting, Sampling and Integrating: Algorithms and Complexity*. Lectures in Mathematics. ETH Zürich. Birkhäuser, Basel, Switzerland, 2003.

[10] Mark Jerrum and Alistair Sinclair. The markov chain monte carlo method: an approach to approximate counting and integration. In *Approximation Algorithms for NP-hard Problems*, pages 482–520. PWS, 1996.

[11] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM*, 51:671–697, July 2004.

[12] Tobias Kley. Approximate counting of graph colorings. `http://sourceforge.net/projects/acgc/`, 2007. Project.

[13] Microsoft. Hlsl (windows). `http://msdn.microsoft.com/en-us/library/bb509561(v=vs.85).aspx`. Product.

[14] NVIDIA. Cg toolkit — nvidia developer zone. `http://developer.nvidia.com/cg-toolkit`. Product.

[15] NVIDIA. Cuda. `http://www.nvidia.com/object/cuda_home_new.html`. Product.

[16] NVIDIA. Direct compute — nvidia developer zone. `http://developer.nvidia.com/directcompute`. Product.

[17] Julius Petersen. Die theorie der regulären graphen. In *Acta Math*, volume 15, pages 193–220, 1891.

[18] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[19] Cristina Nader Vasconcelos and Bodo Rosenhahn. Bipartite graph matching computation on gpu. In *EMMCVPR '09: Proceedings of the 7th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 42–55, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171, New York, NY, USA, 2009. ACM.

[21] Vibhav Vineet and P. J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. *Computer Vision and Pattern Recognition Workshop*, 0:1–8, 2008.

[22] Douglas West. *Introduction to Graph Theory*. Prentice Hall, New Jersey, second edition, 2001.