Hybrid Parallel Computing Strategies for Scientific Computing Applications

Joo Hong Lee

Dissertation submitted to the faculty of
the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Computer Engineering

Paul E. Plassmann, Chair
Mark T. Jones
Thomas L. Martin
Amos L. Abbott
Christopher A. Beattie

August 29th, 2012
Blacksburg, Virginia

Keywords: Scientific Computing, Parallel Programming, Biological Systems Simulation, Pthreads, OpenMP, Multi-threaded Software Performance, Multiprocessor, Parallel Monte Carlo Algorithms, GPU Acceleration, Hybrid Algorithms, Radiative Heat Transfer, Hybrid Computing

Hybrid Parallel Computing Strategies for Scientific Computing Applications

Joo Hong Lee

ABSTRACT

Multi-core, multi-processor, and Graphics Processing Unit (GPU) computer architectures pose significant challenges with respect to the efficient exploitation of parallelism for large-scale, scientific computing simulations. For example, a simulation of the human tonsil at the cellular level involves the computation of the motion and interaction of millions of cells over extended periods of time. Also, the simulation of Radiative Heat Transfer (RHT) effects by the Photon Monte Carlo (PMC) method is an extremely computationally demanding problem. The PMC method is example of the Monte Carlo simulation method—an approach extensively used in wide of application areas. Although the basic algorithmic framework of these Monte Carlo methods is simple, they can be extremely computationally intensive. Therefore, an efficient parallel realization of these simulations depends on a careful analysis of the nature these problems and the development of an appropriate software framework. The overarching goal of this dissertation is develop and understand what the appropriate parallel programming model should be to exploit these disparate architectures, both from the metric of efficiency, as well as from a software engineering perspective.

In this dissertation we examine these issues through a performance study of PathSim2, a software framework for the simulation of large-scale biological systems, using two

different parallel architectures—distributed and shared memory. First, a message-passing implementation of a multiple germinal center simulation by PathSim2 is developed and analyzed for distributed memory architectures. Second, a germinal center simulation is implemented on shared memory architecture with two parallelization strategies based on Pthreads and OpenMP.

Finally, we present work targeting a complete hybrid, parallel computing architecture. With this work we develop and analyze a software framework for generic Monte Carlo simulations implemented on multiple, distributed memory nodes consisting of a multi-core architecture with attached GPUs. This simulation framework is divided into two asynchronous parts: (a) a threaded, GPU-accelerated pseudo-random number generator (or producer), and (b) a multi-threaded Monte Carlo application (or consumer). The advantage of this approach is that this software framework can be directly used within any Monte Carlo application code, without requiring application-specific programming of the GPU. We examine this approach through a performance study of the simulation of RHT effects by the PMC method on a hybrid computing architecture. We present a theoretical analysis of our proposed approach, discuss methods to optimize performance based on this analysis, and compare this analysis to experimental results obtained from simulations run on two different hybrid, parallel computing architectures.

Acknowledgement

First of all, I would like to thank my advisor, Paul Plassmann, for his support and guidance during the past 5 years. His philosophy in student advisory drives me a more independent researcher. Under my situation, I cannot imagine what my life would have been if he were a pushy advisor.

I would also like to thank my committee members, Mark Jones, Tom Martin, Lynn Abbott and Christopher Beattie. And I would like to thank William Baumann for his advice and support about the skills required during graduate school career.

My family support was crucial during the years of pursuing this degree. Especially, I thank my brother who guided me to start Ph.D. and supported me all the time until completing my degree.

Finally, I would like to acknowledge the financial support I received at Virginia Tech from the National Science Foundation and from graduate teaching assistantships. I would also like to acknowledge the assistance I received from the Advanced Research Computing operated by the Virginia Tech, where the computations done on the System X, Inferno and Athena were performed.

Contents

List of Figures

List of Tables

1 Introduction

Current computer architectures provide a computational platform that is much more complex than the platforms that existed five to ten years ago. The architecture of a typical workstation today includes multiple central processing units (CPUs) and attached graphics processing units (GPUs). In addition, these workstations can be easily networked together, and with a batch scheduling system and standardized message-passing libraries (e.g., MPI) that can be used as a parallel computing platform.

For scientists, this complex architecture provides the potential of a tremendous computational resource. However, on the downside, the development of efficient code for these complex architectures is a daunting challenge for even the most experienced programmer. When it comes to implementing software to solve specific scientific computing problems, it is hard to determine how to take advantage of these different architectural levels in a way that is best.

The goal of my dissertation is to attempt to make sense of this complex architectural picture through the detailed analysis and implementation of several representative scientific computing applications. These applications include simulations of biological systems at the cellular level and a general-purpose software framework for Monte Carlo algorithms. The ultimate goal of the analysis of these software implementations is to develop an over-arching approach to "hybrid computing," that is a software approach that takes the best

advantage of the various architectural resources available to the applications programmer.

1.1 Architectures and Programming Models

Perhaps the central difficulty with developing an approach for hybrid computing is the differences in the computer and memory architectures between the available compute platforms. In addition, each of these architectures has a programming model that is standardized and well-suited for the architecture, but these programming models differ and generally are not compatible.

In Figure 1.1, we show a coarse block diagram that illustrates these differences. For example, the standard picture of distributed memory architecture, shown in Figure 1.1 (a), is a number of individual general-purpose processers, each with its own memory and address space, connected together by some sort of high-performance network. One can program each processor in a high-level language, and blocks of processors can be scheduled via a batch scheduler to execute together to solve a single application instance. Communication between the processes is typically done via message-passing through using a standard API such as MPI [1]. The efficiency of this programming model depends on optimizing the performance of code on each processor and developing algorithms that maximize parallelization and minimize message-passing overhead [2].

The multi-core architecture, shown in Figure 1.1 (b), is more tightly coupled than the distributed memory architecture. One typically can run many processes or threads

simultaneously while having access to a global shared memory. One may or may not be able to "pin" a thread or process to a particular core [3]. The programming model takes advantage of the shared memory to allow processes or threads to exchange state through shared memory primitives such as synchronization locks and barriers. One can use a standardized API such as OpenMP or POSIX threads (e.g., pthreads) to implement programs that run on this architecture [4, 5].

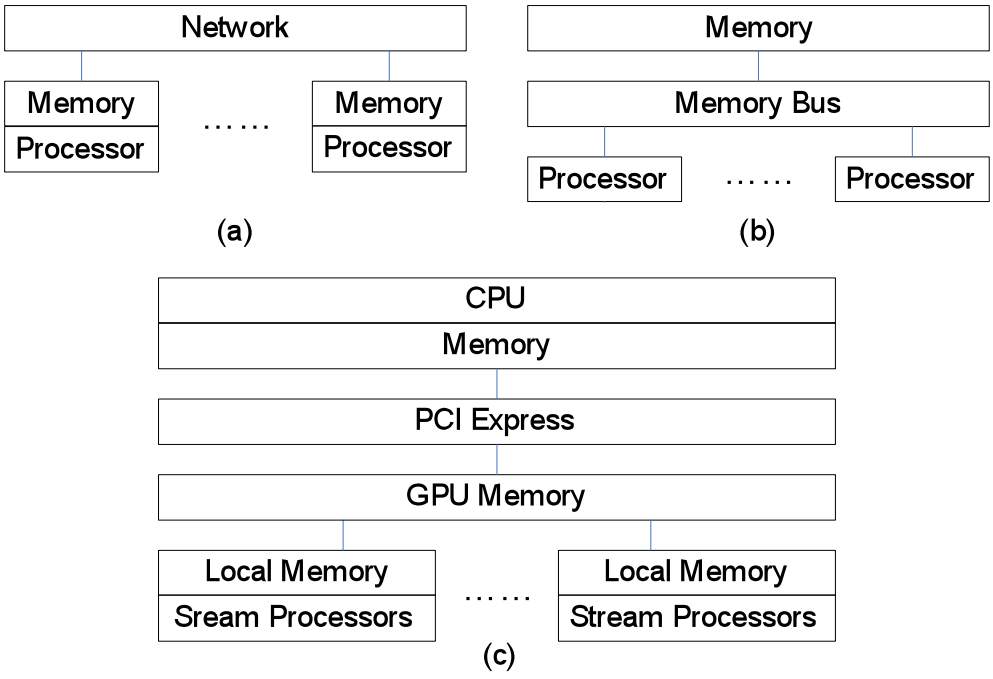

Figure 1.1 Coarse block diagrams of the computer and memory architectures for (a) distributed memory (b) multi-core, and (c) a CPU with an attached GPU processor. Each of these architectures has different characteristic memory access bandwidths and latencies. In addition, the programming model and programming API differs for each of these architectures.

Finally, the GPU architecture, shown in Figure 1.1 (c), is quite different than the previous

two architectures. Conceptually, the use of the GPU is procedural; a program that is running on the CPU offloads a large computational task and its necessary data onto the GPU. The code for the task is written in a language specific to the GPU, and the GPU code executes in a SIMD manner [6]. The data that is necessary to run the code must be explicitly copied from the CPU memory space to the GPU memory space [7]. The CPU must then start the GPU program and wait for it to complete execution. Once finished, output data must be explicitly copied back to the CPU memory space from the GPU.

Because of this data transferring delay between the CPU and the GPU, multiple blocks are generated which can be filled with the data generated by the GPU and also used by the CPU simultaneously. After the GPU fills the first data block, the CPU can start using the data. At the same time, the GPU can generated another data and fill in another block which is empty. In this way, the data transferring delay time can be saved. A good exemplary application of this scheme is a Monte Carlo method applied application, and it is described in detail in Chapter 4 of this paper. Note that standardized APIs for GPU programming include OpenCL and CUDA [8, 9].

1.2 Architecture Characteristics

It is useful to get a high-level picture of the performance characteristics of these architectures. In Table 1.1, we give rough numbers that can be used to characterize the cost of communication for the architectures and their relative processing power. Clearly,

the meaning of communication differs between the architectures, and is to a large extent a function of the programming model. However, these architectural characteristics result in upper bounds on the performance of particular algorithms based on their communication requirements.

For a distributed memory machine, we consider a parallel computer with state-of-the-art processors and an InfiniBand high-performance interconnection network [10]. The latency in the table corresponds to the "start-up" latency for an MPI message, and the bandwidth for a message sent via MPI [11]. The peak performance is the peak double precision (DP) performance of a typical AMD or Intel processor.

For a multi-core architecture, we use as latency measurement the time required for an OpenMP barrier function call [12]. Typically the coordination between threads running on a multi-core architecture is done via synchronization locks and barrier calls so that this gives a rough estimate of the latency overhead of coordinating memory access or task synchronization on this architecture. The bandwidth numbers are based on HyperTransport, the multi-processor interconnect used on AMD multi-core processors [13].

For a CPU with an attached GPU processor, there are a number of ways that one can characterize the architecture. However, for many applications it turns out that the dominant overhead cost is the cost of moving data back and forth between the CPU and GPU memories [14]. The time required for this data transfer has a "start-up" latency that

can be readily measured and bandwidth based on the underlying interconnection architecture (e.g., PCI Express) [15]. The peak processing power is a function of the number of stream processors available on the GPU (in the 700-3,000 range on current GPUs) and the clock rate (in 700 MHz range for a current GPU). One important difference to note in the peak processing rates is that the rate is for single precision (SP) floating point on the GPU.

Table 1.1 A table giving the rough characteristics of the distributed memory, multi-core, and attached GPU architectures. Obviously, the latencies and bandwidths have different meanings for the different architectures. These differences are described in the text.

Architecture	**Latency**	**Bandwidth**	**Peak Processing**
Distributed Memory	4μsec	1.3GB/sec	6.0Gflops(DP)/CPU
Multi-core	0.5μsec	6.4GB/sec	6.0Gflops(DP)/core
GPU	3msec	4.3GB/sec	1Tflops(SP)

Important things to note about these numbers are the relatively large amount of single precision floating-point performance available on the GPU coupled with the relatively high latency cost. In addition, note the imbalance between the peak processing on the GPU and the bandwidth to the GPU from the CPU. The upshot of these numbers suggests that any algorithm that uses the GPU must have a "data reuse" factor that offsets this bandwidth imbalance. That is, any data that is transferred to the GPU must be re-used by the algorithm by a rough factor of 1,000 to offset the transfer time. In addition, any transfer of data between the CPU and GPU must be done in increments large enough to offset the

relatively large latencies. For example, for data sent in block size of 4MB, the cost of the communication latency roughly equals the communication bandwidth cost. The experimental part of Chapter 4 shows how data reuse has an effect on the simulation time.

A final point to keep in mind that the processing power on the GPU depends on having an algorithm that can be framed in lightweight SIMD threads, not as a single-threaded program with complex data access patterns and control structures. Clearly, these restrictions imposed by the architectures have a significant impact on the application programmer's ability to develop software that obtains high efficiencies on an attached GPU. We formalize how these architectural issues affect the types of algorithms that can obtain good performance on this architecture in Chapter 4.

1.3 Specific Contributions in Dissertation

My dissertation attacks the problem of trying to develop algorithms suited for these hybrid architectures by considering a small set of canonical scientific computing problems, developing efficient algorithms for these problems, implementing these algorithms, and then developing a detailed analysis of these algorithms. The hope is that the specific analysis developed in this dissertation can be applied to a wider range of scientific and engineering applications.

The applications considered in this dissertation are specific aspects of PathSim2, a

framework for the modeling of biological systems at the cellular level [16], and a software framework for Monte Carlo applications [17]. These applications are, of course, quite different. However, these applications span a range of computational and inter-process communication requirements and, as such, can be used to target different architectural aspects and provide an interesting contrast as to how these architectural features can be used to solve real-world problems. For example, PathSim2 is implemented on distributed architecture to run multiple Germinal Centers (GCs) simulation and is implemented on multi cores with shared memory architecture to divide the workload of running a GC simulation. The Monte Carlo applications that require random samples use the pseudo-random numbers generated by the GPU. Like this, based on how to fix the simulation model to solve the problem, different architecture should be selected.

My specific contributions in this dissertation can be itemized as follows.

1. The development and analysis of a distributed-memory version of PathSim2. This modified version allows the user to solve loosely coupled, multiple domain problems that are too large to be solved on a single computer. This application and its results are presented in Chapter 2.
2. The analysis of a multi-threaded, shared memory implementation of an element-based processing scheme for Pathsim2. This application and its results are presented in Chapter 3.
3. The implementation and analysis of a GPU accelerated software framework for generic Monte Carlo applications. The software framework performance is

measured and compared to theory for two applications—a π value estimation scheme and a Radiative Heat Transfer simulation. An approach for optimizing the performance of these algorithms based on this analysis is also presented. These applications and their results are presented in Chapter 4.

1.4 Organization

The reminder of this dissertation is organized as follows. Chapter 2 presents a strategy for distributed memory architecture implementation and analysis of PathSim2 adapted for use on multiple, loosely coupled computational domains. Chapter 3 presents a strategy and analysis of a shared memory, element-processing scheme developed for PathSim2. Chapter 4 presents a strategy for a GPU accelerated software framework for Monte Carlo applications and a detailed analysis of the performance of an implementation of this framework. Finally, the dissertation concludes by summarizing its contributions and identifying possible future work in Chapter 5.

2 A Strategy for Distributed Memory Architecture

In this chapter, we consider a strategy for distributed memory architecture. To show this strategy, we implement a biological system simulation of multiple Germinal Centers (GCs) using PathSim2 on a distributed memory architecture [16]. PathSim2 is a software environment developed to simulate biological systems at the cellular level. The goal of these simulations is to model the adaptive immune response of the human tonsil [18, 19]. This biological system represents significant challenges from the perspective of computational science because of its multi-scale properties at the spatial and temporal scales. In particular, the efficient modeling of cellular motion and interaction, called inter-cellular model, must be coupled to sub-scale models of intra-cellular biochemical pathways to adequately model the whole immune system. The key to make such simulation tractable is the development of an overall approach that is able to couple inter-cellular and intra-cellular models in an efficient manner.

PathSim2 uses two parallelization strategies that can be used to speed up these simulations. These strategies work at different spatial scales. First, at the scale of an individual spatial element (a spatial discretization that contains a modest number of cells), PathSim2 can employ a multi-threaded approach that can update the state of independent elements in parallel. This approach would be appropriate for shared memory parallel machines or a multi-core architecture [20]. We presented an analysis of this approach in Chapter 3. Second, to simulate multiple GCs, we have developed a message-passing implementation

suitable for distributed memory computers. It is the second parallelization scheme that is the subject of this chapter.

With the appearance of the parallel hardware and software technologies, large-scale Biological System Simulation (BSS) programmers have adapted their programming models suitable for the parallel architecture [21]. A promising current trend is combining shared- and distributed-memory programming models together [22, 23]. These hybrid, parallel-programming techniques have evolved to take advantage of the emergence of multi-core, distributed memory computer architectures [24]. The parallelization approach developing for PathSim2 follows this trend toward hybrid parallelization strategies in BSS.

2.1 Simulation Model Overview

PathSim2 is a software framework that simulates the movement, aging, interaction and diffusion of agents within a discretized three-dimensional spatial region. The agents simulate biological elements at the cellular level such as various cell types and viruses. The spatial regions represent tissue wherein these agents move. Our target simulation is that of multiple GCs contained in human tonsils. A simplified illustration of a cross-section of this model is displayed in Figure 2.1.

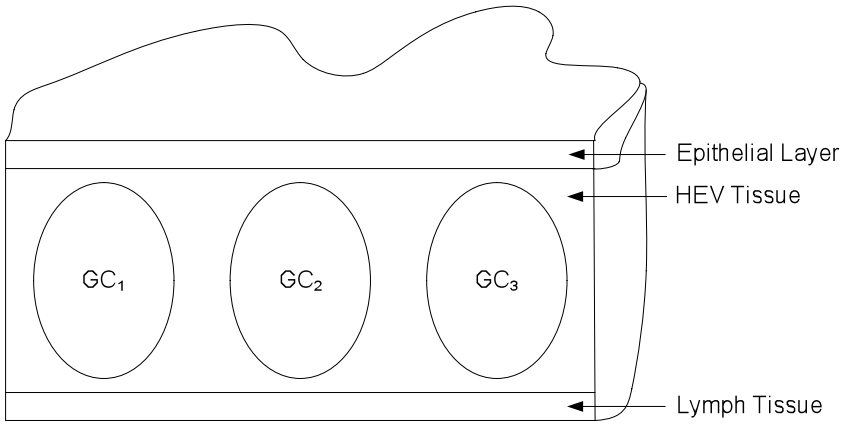

Figure 2.1 A Simplified illustration of GCs in a tissue [25].

In this model, multiple GCs are contained within a thin layer of lymphatic tissue that is bound on the top by an epithelial layer and below by lymph tissue. Cells within this region arrive from a vascular system, or High Endothelial Venules (HEV). Cells leave the system by draining into the lymph systems through connection in the lymph tissue. The tissues are discretized using a structured mesh with individual elements having specific attributes to conform to the complex tissue geometries such as those shown in Figure 2.1. To understand the scale of these models, the thickness of the tissue shown in Figure 2. 1 is roughly 1,000 microns and the width of a GC is roughly 500 microns. A typical element size is roughly 50 microns on a side. Thus, the number of elements used to discretize one GC is on the order of 2,000 spatial elements. In Figure 2.2 we show a cross-section through one of the simulated GCs. Individual cells are shown in addition to a colored background representing the concentration of a particular chemokine. The motion of cells through the tissue is largely determined by these chemokine concentrations.

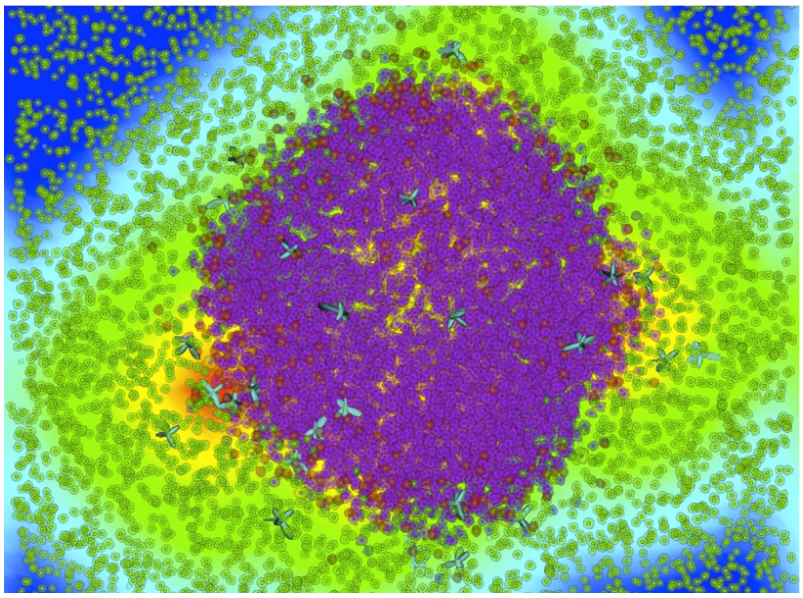

Figure 2.2 A cross-section of one GC model showing the simulated cells and a chemokine concentration [25].

To simulate the flow of cells into and out of the GC mesh, PathSim2 includes a "pool" model to represent well-mixed systems such as the blood and lymph. Because of the well-mixed nature of the blood and lymph compartments, it is not necessary to represent the spatial location of cells in these pools. In the case of the GC, the model includes four pools: blood, lymph, marrow and saliva. For the purpose of this simulation, it is sufficient to consider just the blood and lymph compartments. These pools are connected to multiple elements in the appropriate mesh regions. The pools are connected to each other and to the tissues types as shown in Figure 2.3. The illustration indicates flow rates between compartments; when the flow enters or exists the mesh, the total flow is divided among the connected elements. The flow between compartments is adjusted to ensure that the net

total flow to any compartment is zero.

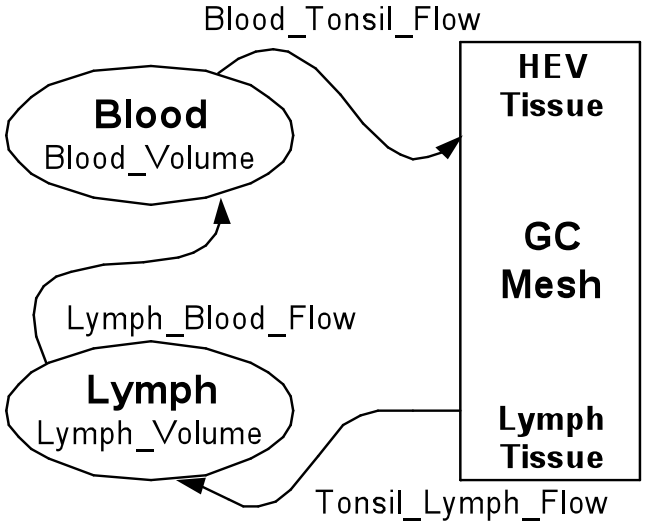

Figure 2.3 Connectivity between pools and the GC [25].

We first consider the case of how the computational model is implemented for a sequential computer. In this case, the blood or lymph pool is directly connected to GC mesh as shown in Figure 2.3. For example, with the HEV tissue, the blood pool is connected to HEV tissue and agents flow directly into elements of the HEV tissue. Likewise, with the lymph tissue, agents flow out of individual elements of the GC mesh and into the lymph pool. Within GC mesh, however, agents flow between adjacent elements based in the computed dynamics of individual cells. Typically, the computational complexity of computing the movement and interaction of agents within the GC mesh dominates the time required to compute the flow into, out of, and between the pools. For all the discussion that follows,

we can assume that computational requirements of updating the mesh dominate the time to update the pools.

Overall, a simulation of just one GC involves the modeling of roughly 100,000 cells, their interactions and movement at a sub-minute time-scale, and the evolution of the GC over weeks of simulated time. As such, the computational complexity of simulating a single GC is significant; these simulations typically run at 10 to 100 times real-time. As the over-arching goal of these simulations is to model tonsil tissues that include thousands of GCs, the potential of parallelizing the calculation and running the simulation on a parallel computer is an attractive option.

To achieve a model suitable for parallel implementation, the straightforward model presented above requires modification. The first modification is made by the observation that the primary motion of individual agents within a tissue containing multiple GCs is essentially limited to a specific GC. That is, from the perspective of the model, we can ignore the motion and interaction of agents between different GCs. We still have to have common pools as the agents mix once they enter the blood or lymph compartments, but for the spatial decomposition of the tissue, we can consider the mesh representing each GC as disconnected. From the perspective of the parallel implementation, this represents a significant simplification, as we do not have to represent communication between the discretizations of different GCs. Of course, if these GC were assigned to different processors, then this movement between GCs would require no inter-processor

communication. Based on this observation, we partition the tissue by assigning different GCs to different processors. This partitioning is illustrated in Figure 2.4.

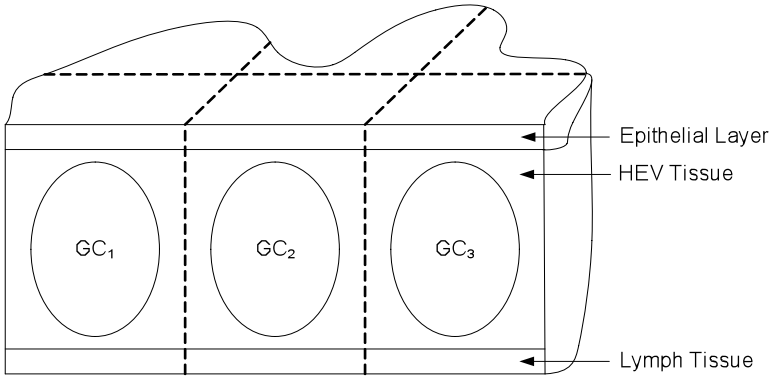

Figure 2.4 An illustration of the spatial partitioning, as indicated by the dotted lines, of different GCs in the tissue. The mesh corresponding to each of these different regions is assigned to different processors [25].

A second modification of the sequential version of the model is made to make a parallel implementation more efficient. The idea is break up the blood and lymph pools into two parts. The first part is the main pools. We call these the global pools and they are essentially the same pools as those used in the sequential model. As the computational cost of updating these pools is minimal, this part of the calculation does not need to be parallelized. However, to interface to the individual meshes on each processor, it is convenient to have a small, local pool represented on each processor. It is these pools that have connections to elements within the mesh assigned to that processor. As these connections are local to a processor, there is no inter-processor communication required. The inter-processor communication is only required to connect the local pools to the global

pools. This new computational model is illustrated in Figure 2.5.

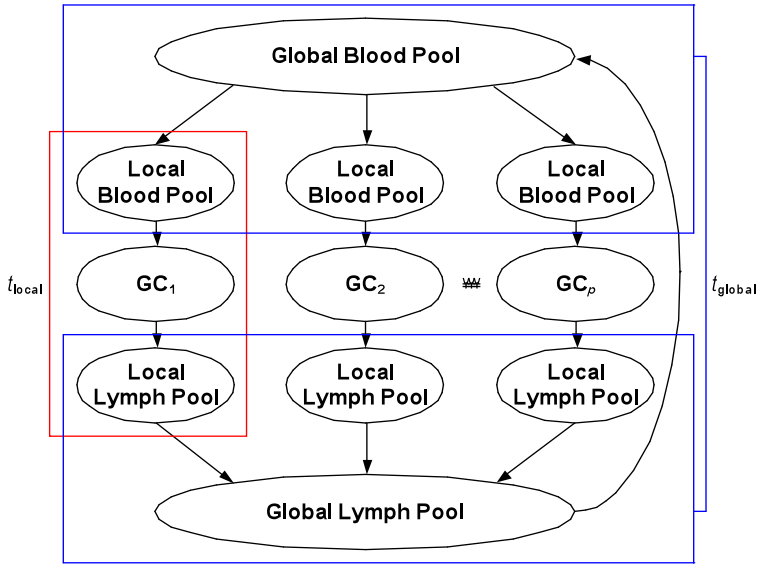

Figure 2.5 Programming model of multiple GCs [25]

In contrast to the sequential model, the parallel model has intermediate pools, one for each GC mesh. As mentioned earlier, the partitioning of the entire tissue assumes that the flow between GC mesh regions is small enough to be ignored by the simulation. In the parallel mode, all the processes are the same as the sequential model, except that the global pool distributes and gathers the information to each local pool as indicated in Figure 2.5.

A third observation that can be used to improve the performance of the model is to note that a longer time-step can be used in the updating of the local pools by the global pools that is required in the updating of the mesh elements and the flow from the local pools to the GC

meshes. In the following discussion let t_{local} denote this latter time-step, the time-step used to update the mesh and flow between the local pools and the GC mesh on each processor. In addition, let t_{global} denote the longer time-step corresponding to the update of the local pools by the global pool (this is the part of the calculation that involves inter-processor communication). The resulting parallel algorithm is summarized in the pseudocode given on the right in Figure 2.6. For comparison, the equivalent sequential code is given on the left in this figure. Again, note that for the parallel algorithm, the *global_local_pool_update*() happens at the longer time-scale corresponding to the t_{global} time-step.

```
while ( t < t_max ) {
        update_pools();
        update_mesh();
        t += t_local ;
}
```

(a)

```
while ( t < t_max ) {
        update_local_pools();
        update_local_mesh();
        if ( (t – t_last_global_update) > t_global ) {
                global_local_pool_update();
                t_last_global_update = t ;
        }
        t += t_local ;
}
```

(b)

Figure 2.6 Pseudocode for (a) sequential model and (b) parallel model. Note that the only routine that requires interprocessor communication is the function global_local_pool_update() [25].

2.2 Theoretical Analysis

To model the performance of the parallelization strategy presented in the previous section, in this section we develop a simple analysis based on the assumption that a message-

passing scheme (such as MPI) is used to communicate between processors. We also assume that processor architecture is homogeneous and the cost to send a message between any two processors is equal.

2.2.1 Scaled Efficiency

To analyze the performance of the proposed GC model simulation on multiple nodes, we model the time required for the calculation as follows. As discussed in the previous section, there are two time-steps associated with the simulation, t_{local} and t_{global}, which correspond to different parts of the multiple GC model as depicted in Figure 2.5. We denote by n the ratio of these time-steps, that is,

$$n = t_{\text{global}} / t_{\text{local}} \,. \tag{2.1}$$

For example, if t_{local} were 0.5 minutes and t_{global} were 30 minutes, n would be 60. To model one time-step of this "global" model (e.g., n time-steps of the "local" or individual GC simulation), let the simulation time for the sequential model be denoted by T_S and the parallel simulation by T_P. The time required for one time-step in the sequential model consists of two parts. The first part is the computational time required to solve for the update to the GC mesh, and the second part is the time required to update the pools. We denote the times required for these two calculations as T_{Mesh} and T_{Pool}. Thus, the required sequential simulation time for n time-steps is given by the equation:

$$T_S = n(T_{Mesh} + T_{Pool}). \quad (2.2)$$

For the parallel model, the time required can be decomposed into the time required to update the mesh, the local pools, the global pools, and the message-passing between processors. If we let the index of a processing node be i, we denote by $T^i_{Mesh,}$ and $T^i_{Local\_Pool}$ the times for updating the mesh and the local pools on each processor. Furthermore, let T_{Global_Pool} and T_M denote the time for updating the global pools and the message-passing time. Note that these last two times do not depend on the processor index as global synchronization points bracket them. The mesh and local pool update times are the times required for one local time-step. Thus, we can express the parallel simulation time to update the mesh and local pools at each processor node as:

$$T^i_P = n(T^i_{Mesh} + T^i_{Local\_Pool}) + T_{Global\_Pool} + T_M . \quad (2.3)$$

We note that the time required to update the mesh and local pools can differ on each processor because of differences in the biological interactions in each individual GC. Thus, we make the following definition,

$$T_{Mesh} + T_{Local\_Pool} = \max_{1 \le i \le p}(T^i_{Mesh} + T^i_{Local\_Pool}). \quad (2.4)$$

where p is the number of processors. In addition, recall that the global update and communication happens only once each n local time-steps. Thus, if we assume that this

maximum time is independent of time-step, then T_P can be expressed as:

$$T_P = n(T_{Mesh} + T_{Local\_Pool}) + T_{Global\_Pool} + T_M . \quad (2.5)$$

If the time to send one message with the required data between the processor with the global pool and a processor with a local pool is t_m, then T_M can be expressed as pt_m, as the messages sent to each processor are unique. Recall that the Scaled Efficiency (SE) is defined as T_S/T_P. Thus, the SE for our model can be expressed as:

$$\begin{aligned} SE &= T_S/T_P \\ &= n(T_{Mesh}+T_{Pool})/[n(T_{Mesh}+T_{Local\_Pool})+T_{Global\_Pool}+pt_m]. \end{aligned} \quad (2.6)$$

In practice, when the simulation time of mesh and local pool is compared to that of global pool, we note that the computation time for global pool is relatively short. In addition, we assume that the time to update the mesh on the sequential machine is essentially the maximum time for any of the GCs on the parallel machine. Thus, we can ignore T_{Global_Pool} and assume that the ratio of sequential mesh update time to the maximum parallel mesh update time is approximately one. Hence, SE can be approximated as:

$$SE = 1/ [\ 1 + pt_m /n\ (\ T_{Mesh}+T_{Pool})]. \quad (2.7)$$

Using the notation $\rho=1/n$ and $\kappa = t_m/(T_{Mesh}+T_{Pool})$, the SE can be approximated as

$$SE = 1/ (1 + p\rho\kappa). \quad (2.8)$$

2.3 Experimental Results

Computational experiments were performed on the System X parallel computer at Virginia Tech. This machine is made up of 1100 compute nodes; each node consists of dual 2.3GHz PowerPC 970FX processors. Each processor is connected to 4GB of ECC DDR400 (PC3200) RAM. The communication network is connected with SilverStorm Technology InfiniBand switches. Four-core switches and 64 leaf switches are used for network interconnections. Each core switch has 132 ports and each leaf switch has 24 ports. Overall, this network architecture satisfies our assumption that the node-to-node communication times are (to first order) independent of processor assignment.

For our experiments we measured the SE as a function of the number of processors. The number of processors used was 10, 20, 30, 40 and 50. In addition, we varied the ratio of the global to local time-step. Global time-steps of 30 and 60 minutes were used. The local time-step was fixed at 0.5 minutes. The GC model assigned to each processor was essentially identical; hence, the workload was evenly distributed to processors.

2.3.1 Scaled Efficiency Measurements

To accurately measure the simulation performance, we ran the simulation five times for a

fixed set of simulation parameters. From these measured simulation times we computed an average, a minimum, and a maximum time for each set of parameters. In Figure 2.7, we present the computed SE for the simulation. The symbols indicate the average values of execution times and the error bars shown at each point indicate the minimum and maximum values over the five runs.

Note that the observed average SE of the implementation with the 60-minute global time-step is consistently higher than that of the 30-minute global time-step. The difference results from the relative difference in the communication overhead as discussed in the preceding section.

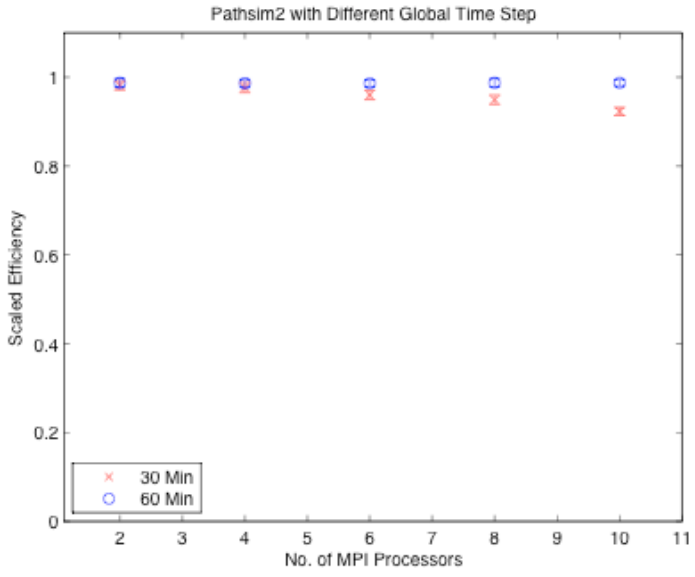

Figure 2.7 The scaled efficiency as a function of the number of processors for different global time-steps [25].

2.3.2 Communication Overhead Time Measurement

Using these results, we can estimate t_m by fitting the theoretical model developed in the preceding section to the experimental results. For the curve fitting, the least squares method was used. Based on this approach, the computed value for t_m is approximately 2.04×10^{-5} second. To see if this time makes sense, we can use the standard linear message-passing model. For System X, the start-up, and incremental message-passing times are given in Table 2.1. Recall that t_s is startup time, or the time to initialize the message (including operating system overhead) and t_w is the time to send a byte over a link in the network (i.e., the reciprocal of the network bandwidth). If we denote the message size (in bytes) sent over the network as m, t_m is given in this linear model as

$$t_m = t_s + mt_w \,. \tag{2.9}$$

The values for t_s and t_w computed from a "ping-pong" test program run on System X yield the values included in the following table.

Table 2.1 System X Parameters [25].

Parameter	Value
t_s	$2.0\text{x}10^{-5}$ sec/message
t_w	$1.9\text{x}10^{-9}$ sec/byte

In our simulation, the message passing occurs with the updates between the global and

local blood and lymph pools. For these messages, m is small enough that we can ignore the mt_w term in equation (2.9). Thus, given that t_m is essentially given by t_s, our model has excellent agreement with the least squares fit to the experimental data.

2.3.3 Verifying Experiment Results

To verify the adequacy of our model, we compare the experimental results with the theoretical model for the SE based on t_m computed with the message-passing parameters given in Figure 2.8. This figure shows the theoretical (solid line) and experimental results (data points) for the SE for the 30- and 60-minute global time-steps. The theoretical model is within the experimental error for the data obtained on System X.

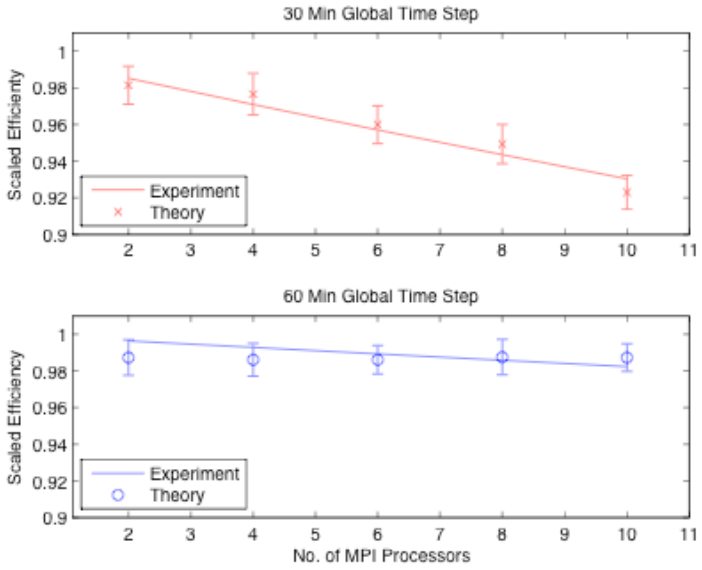

Figure 2.8 A comparison of the experimental data and theoretical model for two different global time-steps [25].

3 A Strategy for Shared Memory Architecture

In this chapter, we consider a strategy for shared memory architecture. To show this strategy, we consider efficient multi-threading strategies for multi-core with shared memory architectures as a second step in developing a hybrid implementation for GC simulation. At the scale of an individual spatial element (a spatial discretization that contains a modest number of cells), PathSim2 can employ a multi-threaded approach that can update the state of independent elements in parallel. This approach would be appropriate for shared memory parallel machines or a multi-core architecture [20]. For this purpose, we focus on different approaches to multi-threading, which enable these applications to make the efficient use of available CPUs on this architecture.

In particular, we study the relative performance of OpneMP and pthreads within PathSim2. The framework has been used for germinal center simulation [26-28]. In this chapter, the theoretical model for the performance of OpenMP and pthreads implementation is described then computational results are compared.

3.1 Simulation Model Overview

PathSim2 is a software framework that simulates the motion and interaction of biological agents (usually representing cells) within a discretized three-dimensional spatial region (usually representing tissue). In the discretization of the physical volume we refer to the discretized sub-volumes as elements and the collection of elements that make up the

physical volume as the computational mesh. Thus, the movement of cells in a tissue is modeled to the movement of agents between neighboring elements in this computational mesh. A simplified two-dimensional illustration of the elements and agents is displayed in the top image in Figure 3.1. In the bottom image of Figure 3.1, we show a cropped, two-dimensional cross-section from a PathSim2 simulation. This image shows a rendering of cells (agents) and elements (indicated by the size of the colored squares).

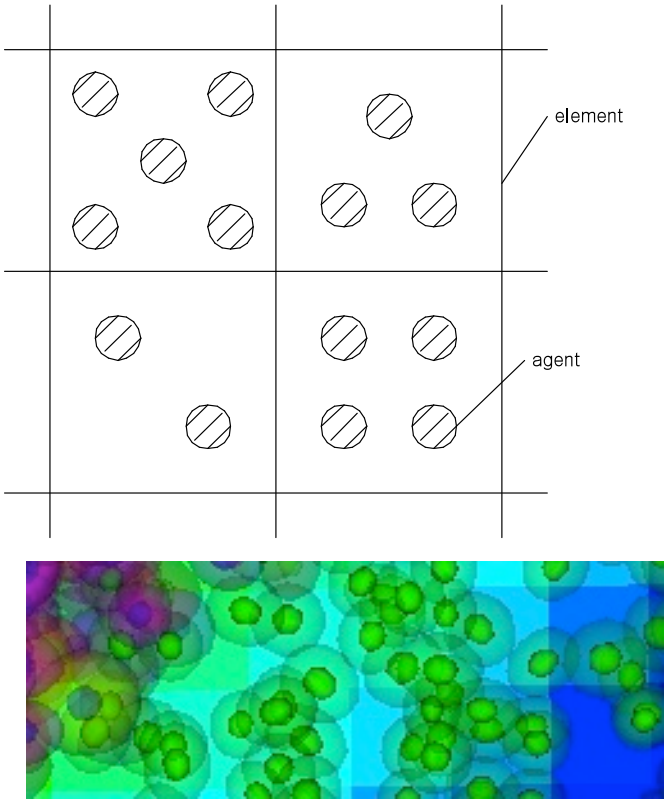

Figure 3.1 (Above) A simplified two-dimensional model of agents with elements; (Below) A close up of a two-dimensional cross-section from a PathSim2 simulation showing cells (agents) and elements (indicated by the colored squares [20].

The interaction of agents is handled by considering the agents within each element

independently. These interactions can be complex and thus can require significant computing time. However, since these element-based calculations are independent, they can be executed in separate threads. A simple approach would be the following. If we had *p* threads available, we could keep a list of the elements (and the agents contained in each element) and assign the interaction calculation for each element to a thread when it became available. Pseudocode expressing this algorithm is given in Figure 3.2. In this pseudocode, the function GET_NEXT returns the next element off of the list in a thread-safe way.

```
Main process:
   Create and start p threads with access to the element list L

Thread function:
   Barrier();    // wait until all threads are started
   // GET_NEXT retrieves the first available element from list
   while ((e = GET_NEXT(L)) != NULL) {
         compute_interactions(e);
   }
   // return NULL for the thread_join in the calling function
   return(NULL);
```

Figure 3.2 Pseudocode for element-based agent interaction [20].

A typical PathSim2 simulation involves computing the motion and interaction of agents over a long period of time. This is done by choosing an appropriate time-step and repeating the movement and interaction calculations. The simulation time is incremented by the time-step until the desired simulation time is reached. As these simulation times

can be long (e.g., days, months, or even years) and the time-steps are short (on the order of minutes), the computational time can be significant. Thus, it would be extremely useful if we could improve the simulation performance by taking advantage of the above multi-threaded approach done on multiple cores. To help us analyze the potential of this approach, in the following section, we develop a theoretical model for the performance of the multi-threaded approach.

3.2 Theoretical Analysis

The calculations done in PathSim2 are quite complex and as a result, difficult to analyze. Thus, to be able to draw conclusions about the multi-core performance of PathSim2, we first develop a simple model of the calculations done by the framework. In particular, the number of agents in an element and the nature of their interactions vary greatly. Also, the calculation at each time-step involves more than just interaction: it involves movement, aging, diffusion of chemokines, and the simulation of a number of other processes. For our simplified model, we consider just the interaction phase of the simulation (the most computationally expensive part of the simulation). We also initially assume that the work require per element is homogeneous.

3.2.1 Overhead of Threading

Our proposed approach is based on creating and destroying multiple threads, thus it is important to understand the overhead associated with this thread management. We

assume that the main process creates the threads sequentially, and that this process takes some small fixed amount of time, which we denote by T_s. This time may depend on the particular thread implementation used (e.g., pthreads or OpenMP), however we assume that this overhead time is constant for a particular implementation.

3.2.2 Speedup

To analyze the speedup of our multi-threaded implementation of the simplified model we employ the following assumptions. Let the number of elements in our simulation be N. Also, let us assume that the interaction calculation done for each element requires some fixed amount of work, say W, and takes time T_W. Finally, let the number of threads (and cores) be p. Ideally, one would like a computational job that is split up among p processors and complete in $1/p$ time. However, there will be overhead and sequential code portions that limit the efficiency of the code. To quantify how close we are to the ideal case, we use the speedup metric defined as the ratio of the time required running the calculation on one core to the time required on p cores.

Let $T_{seq}(N,W)$ be the time required to complete the task on one processor and $T_{parallel}(N,W,p)$ be the time required with p threads. Then, the speedup S is defined as

$$S = T_{seq}(N,W) \,/\, T_{parallel}(N,W,p) \tag{3.1}$$

The sequential time is simply NT_W. The parallel time consists of two parts, the sequential

work divided up between the p threads and the thread management overhead. To model the overhead, there are two aspects to consider. First, there is the time required to create and destroy each thread, and second, there is the synchronization overhead in getting elements from the list in a thread-safe manner. We assume that the first cost is the dominant cost. If we let T_s represent the time required to create and destroy one thread, then since the thread are created sequentially by the main routine, we have that

$$T_{parallel}(N,W,p) = \lceil N/p \rceil T_w + pT_s. \tag{3.2}$$

If we assume that N is large relative to p, then we can approximate the speedup by the equation

$$S \approx p / (1+p^2 T_S/NT_W) \tag{3.3}$$

3.2.3 Multi-threading of Pthreads and OpenMP

To implement this approach, the two most common choices for a multi-threading application programming interface (API) would be Pthreads and OpenMP. Pthreads is the POSIX standard API to handle the actions required in a multi-threaded application. These actions include (among others) thread creation, joining, argument passing, and termination. The Pthreads API requires a detailed specification of the thread context when it is created. In exchange for the amount of multi-threading code required, Pthreads provides extensive control over threading operations. On the other hand, the OpenMP standard was

developed to standardize the shared-memory programming model, and to ensure portability between different machines. We can use a subset of this API to provide a simpler (than the Pthread implementation) multi-threaded programming implementation.

Pseudocode example for different multi-threaded implementations is illustrated in Figure 3.3. In the following section we compare OpenMP and Pthreads implementations of our simplified model and analyze the performance of these algorithms with respect to the above sequential model. Finally, we consider the performance of PathSim2 relative to this simplified model.

3.3 Experimental Results

The computational experiments were performed on an SGI ALTIX 3700. This machine uses 128 Intel Itanium processors, each running at 1.6 GHz. The processors are connected to a 512GB main memory by a crossbar switch that can provide a peak bandwidth of 6.4 GB/sec. The cache memory for each processor is 16KB of L1, 256KB of L2, and 4MB of L3.

3.3.1 Test Conditions

In measuring the speedup of the different multi-threaded implementations, we consider three varying problem parameters: the number of threads, the number of mesh elements, and the amount of work required for each element. The number of threads used was 2, 4,

6, 8 and 10. The number of threads was limited due to the number of processors available on the SGI ALTIX 3700.

```
// Prototype for the thread function "thread_func,"
// thead_arg is a pointer to any information required by the thread.
// Pseudocode for the thread function is given in Figure 3.2.
void *thread_func (void *thread_arg )

// Create num_thread threads.
for i=1,…,num_threads do
        status = pthread_create(…., thread_func, &thread_arg);
        if (status != 0)
                display error and exit
         endif
enddo

// Wait for threads to complete and join.
for i=1,…,num_threads do
        status = pthread_join(…);
        if (status != 0)
                display error and exit
         endif
enddo
```

(a) Pthread-based pseudocode

```
//Creating and joining threads are executed by OpenMP directive
#pragma omp parallel
thread_func (&thread_arg );
```

(b) OpenMP-based pseudocode

Figure 3.3 Pseudocode for different threading methods [20].

To make the workload equal for all processors, the number of elements must be a common multiple of threading numbers. For this reason, the number of elements was set to be 120 - the least common multiple of number of threads used in the experiments.

Each computational mesh has N elements; in the real simulation each element contains some number of agents that interact with each other. To mimic this calculation in our test problem, we used a matrix-matrix multiplication of two n by n matrices. The work (e.g., the number of floating point calculations) required to multiply these matrices is $W = O(n^3)$. This calculation is similar to the interaction calculation in the real problem in that if we have n agents in an element, the work required depends nonlinearly on n. We used five different matrix sizes, with n = 20, 40, 60, 80, and 100.

3.3.2 Speedup Measurements

To measure the performance of the test code, for each set of parameters we have run the simulation ten times. From the measured execution times we compute an average and a minimum and maximum. In Figure 3.4, we present the measured speedup for the Pthreads and OpenMP implementations on the multi-processor shared memory systems. The left plot shows the Pthread implementation and the right shows the OpenMP implementation. In Figure 3.4, the symbols indicate the average values of execution times and the error bars for each point indicate the minimum to maximum values.

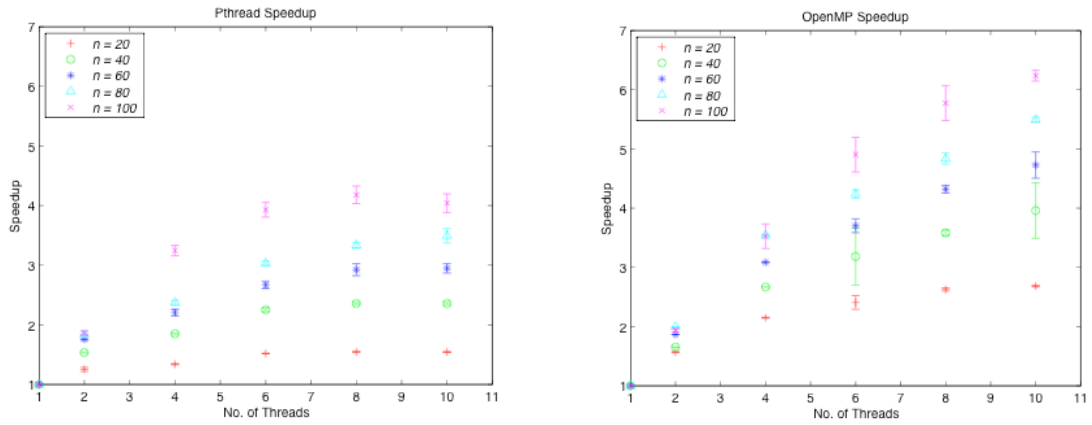

Figure 3.4 Speedup of Different Multithreading [20].

Note that the observed speedup of the OpenMP implementation is consistently higher than that of the Pthreads implementation. This difference results from the different thread management times for the Pthreads and OpenMP implementations. Also note that, as expected, the speedup increases for larger workloads.

3.3.3 Startup Time Estimates

Using these results we can estimate T_s by fitting the theoretical model developed in the preceding section to the experimental results. For the curve fitting the least squares method was used. Based on this approach, the computed startup times for each working set size are listed in Table 3.1. From Table 3.1, we can see that the startup time for OpenMP (as computed from our model) is approximately 2/3 the time for the Pthreads implementation.

Table 3.1 The startup time, T_S, computed from the theoretical model for different values of N [20].

	Startup Time (s)	
N	Pthreads	OpenMP
20	0.0625	0.0312
40	0.0672	0.0407
60	0.0707	0.0467
80	0.0720	0.0478
100	0.0711	0.0339
Average	0.0687	0.0424

3.3.4 Verifying Experiment Results

To verify the adequacy of our theoretical model, we can compare the experimental results with the speedup equation (3.3). For this verification, the value for T_s in equation (3.3) is the average startup time taken from the results in Table 3.1. In Figure 3.5, we show plots of experimental results for n = 100 and T_W = 1 second. The upper plot shows the results for the pthread implementation and the lower graph the results for the OpenMP implementation. As shown in Figure 3.5, the experimental results are consistent with the theoretical model for both multi-threaded implementations.

As shown in Figure 3.5, the experimental results are consistent with the theoretical model for both multi-threaded implementations.

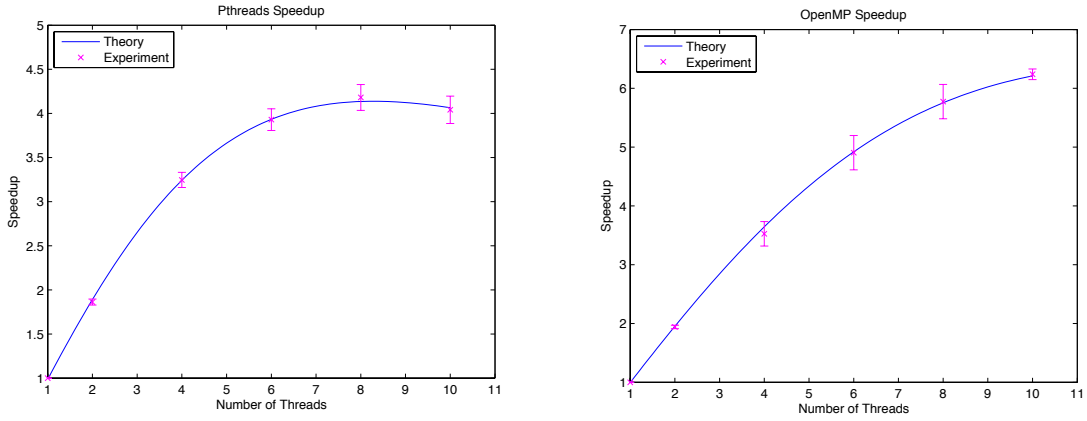

Figure 3.5 Speedup of Experimental and Theoretical Results when n = 100 [20].

3.3.5 PathSim2 Speedup Measurements

We can compare the results obtained from our simplified model to computational results obtained from running the complete PathSim2 simulation code. For a standard data set modeling a germinal center, we obtained running times for both OpenMP and Pthreads implementation of PathSim2. The computed speedups are shown in Figure 3.6 as a function of the number of threads used. Note that, as expected, the OpenMP model performs better than pthreads model. These results are consistent with the experimental results obtained when running the matrix multiplication code, and shows good scaling in spite of imbalances in the computational complexity between mesh elements.

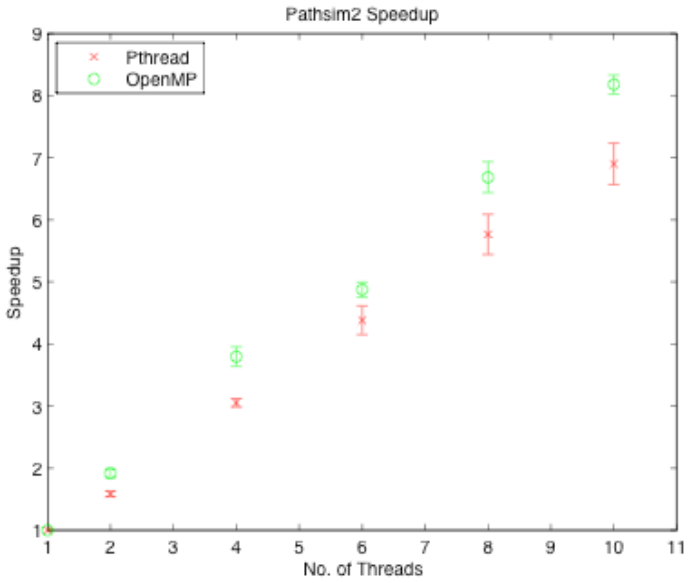

Figure 3.6 The speedup measured from a representative PathSim2 simulation using Pthreads (red) and OpenMP (green) implementations [20].

4 A Hybrid Parallel Computing Strategy for Monte Carlo Applications

4.1 Introduction

In this chapter, we present a strategy for utilizing a hybrid parallel computing architecture with good efficiency for a specific class of applications. In the previous chapters we presented strategies for the efficient use of shared memory and distributed memory architectures for the simulation of biological systems. However, the development of an over-arching hybrid strategy involves the integration of very different programming models and underlying architectures.

With this aim in mind, we have chosen to focus on a class of scientific simulation methods that are both widely used and have the potential of exploiting the full potential of a hybrid computing architecture. Monte Carlo methods are widely used in the scientific and engineering simulation community [46-48]. In addition, these methods are typically "separable" into two basic software parts: (1) a function of random variables that must be evaluated many times in order to achieve a desired statistical accuracy; and (2) a method for the generation of the uncorrelated random (or, more properly on a computer, pseudo-random) variables necessary to accomplish the function evaluations. Often times the function evaluation can be quite complex; for example, if high-order accuracy is necessary or in the simulation of complex physical phenomena. On the other hand, the generation of

the massive number of pseudo-random numbers necessary can be decoupled from the function evaluation, and can be accomplished by any number of well-know algorithms [49].

There are several reasons why Monte Carlo applications are a good candidate for a hybrid computing architecture. First, the function evaluation is typically "embarrassingly parallel." That is, if we have a distributed memory computer consisting of many multi-core nodes, each node can run as many threads as cores and accumulate statistics for each node. Then the results on each node can be combined using global, distributed memory communication to obtain statistics for all the nodes. Second, if each node has an attached GPU, the generation of the pseudo-random numbers could, in principle, be off-loaded to the GPU. The combination of these two strategies has the potential take full advantage of a hybrid parallel computing architecture and obtain good speedups for Monte Carlo applications.

In this chapter we present a new approach that accomplishes this goal. First, we present a portable class-based software library to support Monte Carlo applications. Second, we present a detailed analysis of the performance of this implementation on hybrid architectures. Third, we compare this analytical model with experimental results from a moderately sized hybrid parallel computing cluster for a real-world, radiative heat transfer calculation based on Monte Carlo ray tracing.

4.2 The GPU Architecture and its Potential Use

In Chapter 1 we gave a brief introduction to GPU architecture and its potential use to accelerate computation. The relative advantage of the GPU over the CPU is primarily due to the fact that it has a large number of simple, stream processers. A rough estimate of this performance advantage may be obtained by assuming that the GPU has 1,000 such stream processors, each running at a clock rate of 1.0GHz. This estimate yields a raw peak performance of this hypothetical GPU of 1Tflop, assuming 1 flop per instruction per processor. A typical CPU on the other hand might have a slightly faster clock rate of 2.5GHz. The CPU may at most perform 2 floating-point operations per clock cycle. Thus, the raw peak performance of this CPU would be 5Gflops. Hence, based on this rough comparison of the peak performance of the two architectures gives us a potential speedup of 200 for the GPU over the CPU. Much of what consider in this chapter is the answer to the question, "How much of these potential speedup can we obtain for a real-world application?"

In Figure 4.1 we give a more detailed block diagram of the CPU with attached GPU architecture than that given in Chapter 1. We begin by reviewing a number of important aspects of this architecture. Note that the thread (or stream) processors on the GPU are organized into groups. These groups are known as "compute units." For example, the ATI Radeon HD 5750 that we use for some of the experimental results presented in this chapter has 9 compute units, each with 80 stream processors, for a total of 720 stream processors. Note each compute unit has a local memory associated with it, although this local memory is typically quite small (e.g., 32KB for each compute unit on the ATI Radeon

HD 5750). Use of this local memory avoids memory contention issues that arise when using the much larger global memory on the graphics card, and sharing this memory among the compute units.

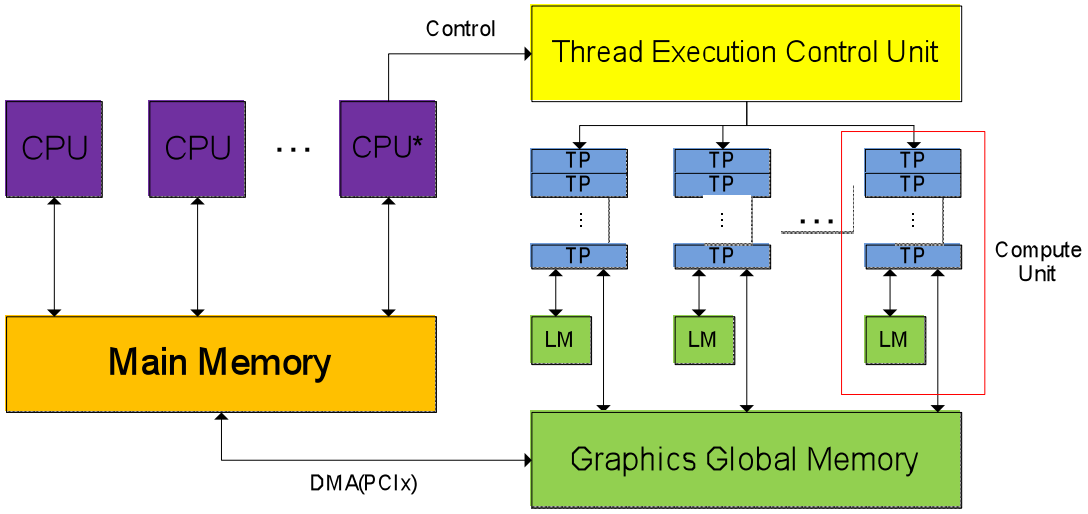

Figure 4.1 A block diagram of the architecture of a multi-core CPU and attached GPU. The abbreviations used in the diagram are: TP, thread (or stream) processor; and LM, local memory. In addition, note that only one of the core CPUs can run a program on the GPU at a time. In the diagram, the core CPU* is the one executing a program on the GPU. Data from the CPU main memory and GPU global memory can be transferred via DMA over the PCIx bus.

The program being run on each stream processors is based on the instructions issued by the "Thread Execution Control Unit." The instructions execute in SIMD fashion on each compute unit. Different compute units execute independently unless there is some global synchronization event. Each compute unit executes a number of logical threads, referred to as "work items" in the OpenCL terminology. These work items are scheduled to run on the individual stream processors in the compute unit by the control unit. Obviously, one

typically wants the number of work items assigned to compute unit to be a multiple of the number of stream processors. In OpenCL terminology, the number of work items assigned to a compute unit is known as the "work group size."

Data that is needed by the GPU must be explicitly copied from the CPU memory to the GPU memory. Likewise, results that are computed on the GPU must be explicitly copied from the GPU memory to the CPU memory. The physical connection between these two memories is the PCIx bus shown in the figure. DMA transfer is the mechanism used to execute the copies between the memories. These copies can achieve the full bandwidth possible over PCIx, typically 4.3GB/s. However, a distinct problem with these DMA transfers is the long latencies required to initiate the transfers. We refer to this latency in our analysis as a "start up cost." These memory transfer times can be improved somewhat by using some specific options (e.g., using memory pinning [57]). However, the start up cost remains large; for example, around 3ms for the experimental results we present in this chapter.

Finally, we note the GPU cannot be used by more than one CPU (although this may change with the new version of CUDA [56]). Thus, in the figure we depict one core, CPU*, as having exclusive control of the GPU. This CPU is the one that initiates any memory transfers, downloads and executes the GPU program, and retrieves the computed results when the GPU program finishes.

4.2.1 Programming the GPU

As is apparent from the detailed discussion of the GPU architecture, a standard programming language such as C/C++ is not appropriate for the GPU. The development of GPU specific programming languages has been the focus of much recent development [59]. Clearly, as the architecture is SIMD, and due to the importance of the memory hierarchy on the GPU, it is important that the language map well to the underlying GPU architecture.

Current GPU programming languages include the CUDA API of Nvidia [29] or Brook+ of AMD [30]. However, a distinct problem with these programming APIs is that they are vendor specific, and compatible only with their own vendor's hardware. In contrast, the OpenCL API is an open standard put forward, developed, and maintained by the Khronos Group consortium [8]. The OpenCL API has been adopted by a number of vendors including Intel, ATI, Nvidia, and Apple. The goal of this standard is to maintain portability while achieving performance comparable to the vendor specific implementations.

Given the portability of the OpenCL API and its ability to achieve good performance on GPU architectures, we have used this API for the GPU implementation for the pseudo-random number portion of the Monte Carlo software. The organization of the code developed is discussed in more detail in subsequent sections of this chapter.

4.2.2 A Simple Model for GPU Speedup

Based on the above discussion on the GPU hardware, and its performance characteristics, one can develop a simple model for the speedup of an application that takes advantage of an attached GPU for some of its processing. The basic idea for our model is that we assume two things. First, we assume that some amount of data, for example the computed results, have to be transferred between the CPU and GPU memories. Second, we assume that for each word transferred between the memory, the underlying calculation done on the GPU executes some number of instructions that is proportional to the number of words transferred. For example, with the pseudo-random number generation done on the GPU, we know that up to 200 instructions are generated on a stream processor for each number generated [49]. We can think of this factor as a form of data "reuse" and we denote it by the symbol ρ. Finally, we assume that our calculation achieves perfect speedup, so that the programs run independently on the stream processors and only the number of stream processors, p, on the GPU, limits that performance on the GPU.

Given the discussion above about transferring data between the CPU and GPU (or visa versa), we can use the following equation to model the time, $T_{Tranfer}(n)$, to transfer n words of data,

$$T_{Tranfer}(n) = t_w n + t_s \tag{4.1}$$

In this equation, t_w represents the time to transfer a word of data (say 4 bytes) and t_s represents the "start up" cost for each transfer. From the above discussion, we can approximate t_s by 3.0ms, and t_w by 4.0bytes/4.3GB/s $\approx 9.3\times10^{-10}$ s/word.

To estimate the time required for our model program to execute on a GPU with p stream processors, $T_{GPU}(n,p)$, we can use the following expression,

$$T_{GPU}(n,p) = \rho\, t_{GPU} \lceil n/p \rceil + T_{Tranfer}(n). \qquad (4.2)$$

Again, recall that the number of times that each word transferred is "reused" is given by the term ρ, and that the time to execute one instruction on the GPU is t_{GPU}. A simple approximation for t_{GPU} would be 2.0×10^{-9} s, assuming that the GPU clock was 500MHz (just to use round numbers). Note that the number of concurrent threads that can run is limited by p, the number of stream processors and we assume that each thread uses one word of data.

To compare the time on the GPU to the time for the same problem to execute on the CPU, we can assume (again rather unrealistically) that we can execute our code at the peak flop rate, so that t_{CPU} would be about 2.0×10^{-10} s for our 5Gflop CPU. Thus, the time required to execute our program on the CPU for n words, $T_{CPU}(n)$, can be modeled by the equation

$$T_{CPU}(n) = \rho\, t_{CPU}\, n. \qquad (4.3)$$

We can use these two expressions to compute an overall speedup for using the GPU over using the CPU. The resulting speedup, $S(n,p)$, is given by the expression,

$$S(n,p) = T_{CPU}(n) / T_{GPU}(n,p). \tag{4.4}$$

We can look at this expression for the speedup as two-dimension function of the number of words that we are using and the amount of reuse that each word of data gets once on the GPU. In Figure 4.2 we show a two-dimensional contour plot showing iso-speedup curves for several different speedups.

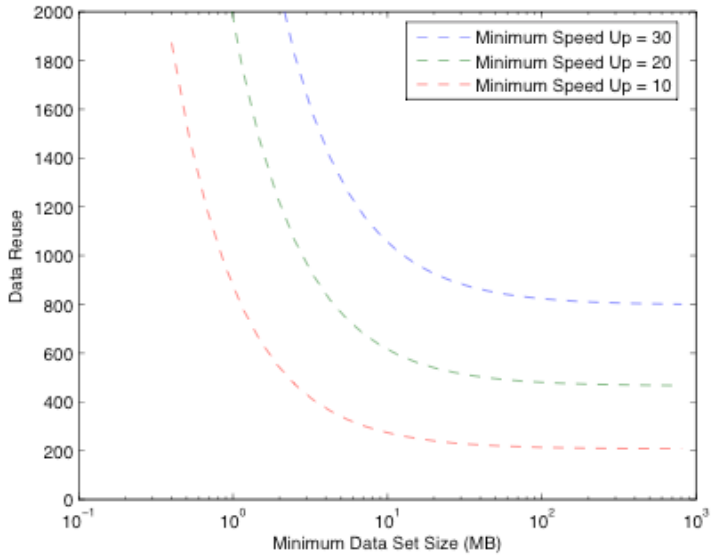

Figure 4.2 Iso-speedup curves for a simple computational model for different amounts of data reuse and data communication between the CPU and GPU.

In Figure 4.2 note that although the theoretical maximum speedup is 1,000, the

actual speedups that one can achieve are severely limited by the overhead inherent in the data transfers between the CPU and GPU. In order to achieve speedups greater than 10, the data reuse must be at least 200. This asymptotic relationship can be improved by increasing the data reuse, increasing the number of stream processors, or by decreasing the incremental transfer cost t_w.

In addition, note that the minimum data set size must be large, greater than 1MB, in order to achieve any reasonable speedup. This limitation is a result of the relatively large start up cost, t_s. Thus, the only way an implementation will achieve reasonable speedups is if the data transfers are large and combined into single transfers (or as few transfers as possible). Given that we know the pseudo-random number generation has a data reuse of about 200, this simple model predicts that the best speedups we will be able to obtain is in the range of 10 to 20. The results that we present later in the chapter are quite close to this rough estimate. In this results we do slightly better. This difference is due to the fact that the CPU does executes the PRNG at a slower rate than the 5Gflop rate of our model, and the fact that the GPU clock is slightly faster than the 500MHz that we used in our model. In any case, this simple model dramatically illustrates some of the limitations of using the GPU to accelerate a calculation.

4.3 Introduction to Monte Carlo

As noted at the beginning of this chapter, the Monte Carlo method is widely used in scientific and engineering applications [46-48]. To understand the basics of implementing the Monte Carlo method, an example of simple toy problem is illustrative. Suppose that we wanted to compute the number π. One way to do this would be to compute the area of the unit circle, which we know to be π. We could accomplish this task numerically using the following approach. Suppose that we have a function Random() that returns uniformly distributed, statistically independent random numbers on the unit interval [0,1]. Consider the illustration shown in Figure 4.3.

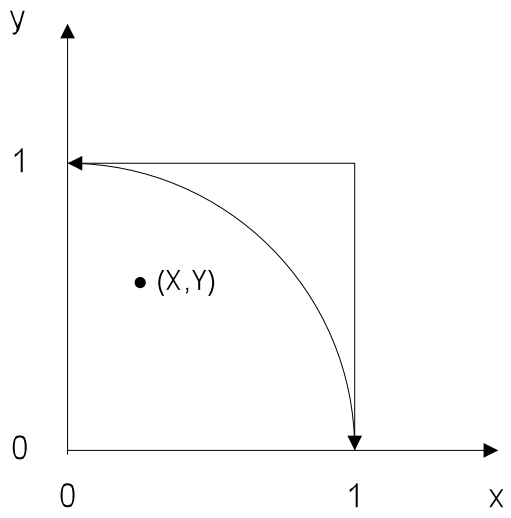

Figure 4.3 A simple example illustrating how π can be computed by using a Monte Carlo method. In this case, the random point in the unit square is checked to determine whether it is within the unit circle. Statistics on the number of such points can be used to estimate the value of π.

We can use two of the random values to generate a random point in the unit square, (X,Y), as shown in the Figure 4.3. This point is either within the unit circle or not. We generate many independent random points and keep track of the number of these points that are found to be within the unit circle. If we then compute the ratio of the number of points within the circle compared to the total number of points generated, this ratio would give us an estimate of the area of the quarter circle shown in the figure. Given the areas of the unit square and the quarter unit circle, this ratio would give us an estimate of the value of $\pi/4$.

```
// A sequential code to estimate π using a Monte Carlo scheme

// num_samples is the number of Monte Carlo samples to be done
int num_inside = 0;
for (int i=0; i<num_samples; i++) {

        // 1. Generate the random point (X,Y)
        float X = Random();
        float Y = Random();

        // 2. Compute the Monte Carlo "function" F
        float F = X*X + Y*Y;
        if (F <= 1.0) num_inside++;
}

// Estimate π
float pi_estimate = 4.0 * (float) num_inside / (float) num_samples;
```

Figure 4.4 A sequential code to estimate π using the Monte Carlo method

Another important point about this program is that the computational cost of evaluating the function Random() is relatively expensive. For example, the well-known pseudo-random number generator RANLUX requires from 24 to 389 instructions on the CPU to generate a pseudo-random number [50]. Thus, running time for the code to compute π given in Figure 4.4 is completely dominated by the computational cost of computing the pseudo-random numbers.

4.3.1 A Multi-Threaded Version of the Monte Carlo Method

A first step toward developing a hybrid parallel version of the Monte Carlo code would be to adapt it to a multi-core, shared memory architecture. The obvious way to implement such a code would be to use a multi-threaded approach where a "master" process spawns a number of "worker" threads, each of which computes independent evaluations of the Monte Carlo function. The master process then waits for all the worker threads to complete and then uses these results to compute an improved Monte Carlo estimate. For example, using the toy problem discussed above, a pthread version of this multi-threaded Monte Carlo code could look something like the code given in Figure 4.5.

It is clear that the above approach is "embarrassingly parallel." One would expect that if the number of samples (*num_samples*) is large and the number threads used (NUM_THREADS) is equal to the number of cores available, then the overall speedup would close to the number of cores. However, there are some critical aspects to making this scheme work. First, the pseudo-random number generators (PRNG) have an internal

state. Thus, each thread must have its own instance of a PRNG so that the state is not shared. The PRNG are usually designed to be thread-safe (e.g., GSL [43]); however, using a single instance would mean that only one thread would be allowed to alter the state at a time defeating the whole purpose of the parallelization scheme. Second, if each thread has its own instance of the PRNG, then the state tables must be initialized in a way that ensures that the numbers generated are uncorrelated [51].

4.3.2 Asynchronous Generation of Pseudo-Random Numbers

The multi-core approach discussed in the previous subsection can achieve excellent speedups; however, the running time of the program would still be dominated by the time required to compute the pseudo-random numbers. Our goal is to develop an approach that takes advantage the tremendous computational resources available on an attached GPU to dramatically reduce the time required to generate these numbers. In this section we produce a new, asynchronous algorithm that allows us to take full advantage of GPU to achieve these speedups.

There are several key observations that we can use to develop our algorithm. First, although we think of the calls to the function *Random()* in Figures 4.4 and 4.5 as generating the pseudo-random number after the function call, and then returning the result, there is no reason that the numbers cannot be generated earlier. All we require is that the numbers are statistically independent. Thus, in principle, we could use a large array to hold statistically independent pseudo-random numbers that are generated asynchronously, prior

```
// A mutli-threaded code to estimate π using a Monte Carlo scheme

// A data structure to pass arguments to the worker threads
typedef struct {
      int num_samples;
      float pi_estimate;
} threadArgs;

// the worker thread
void *pi_estimate_thread (void *ptr)
{
      threadArgs *thread_arg_ptr = (threadArgs *) ptr;
      int num_inside = 0;
      for (int i=0; i< thread_arg_ptr->num_samples; i++) {
            // 1. Generate the random point (X,Y)
            float X = Random();
            float Y = Random();
            // 2. Compute the Monte Carlo "function" F
            float F = X*X + Y*Y;
            if (F <= 1.0) num_inside++;
      }
      // Estimate π
      thread_arg_ptr->pi_estimate = 4.0 * (float) num_inside / (float) num_samples;
      // exit thread
      pthread_exit(NULL);
}

// The main "master" routine that spawns the worker threads, waits for these
// threads to complete, and then merges the pi estimates from each thread

int main()
{
      threadArgs   thread_args_array[NUM_THREADS];
      float    thread_pi_estimate[NUM_THREADS];

      // Start all the worker threads
      for (int i = 0; i < NUM_THREADS; i++) {
            threadArgsArray[i].num_samples = num_samples;
            pthread_create(&threads[i], &attr,   pi_estimate_thread,
                  (void*) &thread_args_array[i]);
      }
      // Wait for all threads to complete
      for (int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
            thread_pi_estimate[i] = thread_args_array[i].pi_estimate;
      }
      // Compute overall estimate for π
      float sum = 0;
      for (int i = 0; i < NUM_THREADS; i++) {
                  sum += thread_pi_estimate[i];
      }
      float pi_estimate = sum / (float) NUM_THREADS ;

      return(0);
}
```

Figure 4.5 A multi-threaded code to estimate π using the Monte Carlo method

to any of the *Random()* function calls. A call to *Random()* could just read the next number available from this array and increment a pointer keeping track of which numbers had been used. Second, if we compute the pseudo-random numbers on the GPU, for efficiency we will have to compute large blocks of these numbers at a time because of the time required to copy these numbers from the GPU memory to the CPU memory where they are used. Third, on the GPU we can use the same scheme that we used in the multi-core program where we have different instance of the PRNG state for each independent thread running on the GPU. This will ensure that the numbers generated are statistically independent.

In addition to the three observations detailed in the previous paragraph, there are constraints on how the GPU can be used. In particular, only one thread can execute a GPU program at time [52]. Because of this constraint, it makes sense, from a programming perspective, to have a single thread that we can think of as a daemon that "manages" the execution of the GPU programs that generate the blocks of pseudo-random numbers. On the other hand, the worker threads that use the pseudo-random numbers should be able to run asynchronously with respect to this daemon. All each worker thread requires is access to a unique block of pseudo-random numbers. When presented in this way, this approach fits into the producer-consumer design pattern [53-55]. We have one producer, the daemon that manages the GPU programs and produces full blocks of independent pseudo-random numbers. And we have multiple consumers, the worker threads that use the full blocks of pseudo-random numbers and return them empty to be re-filled by the GPU managing daemon.

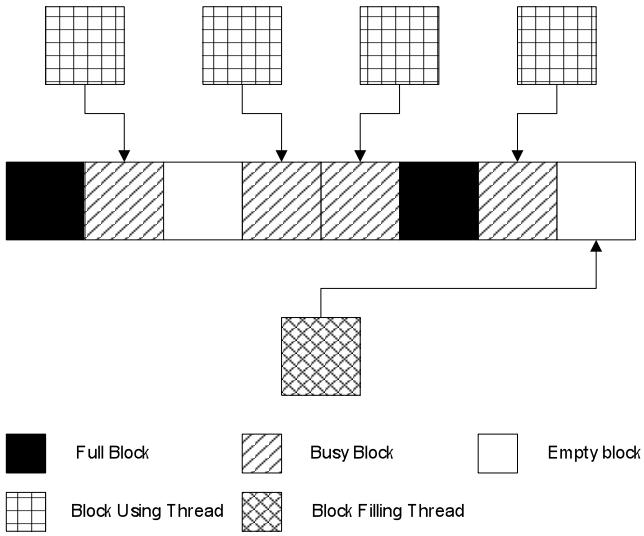

Figure 4.6 An illustration of how the random number blocks are managed between the GPU managing thread and the Monte Carlo application threads. The thread that manages GPU acts as a producer; it has exclusive access to a memory block and fills this block with data from the GPU program. Multiple Monte Carlo application threads act as consumers of these blocks; they may obtain exclusive of a filled block, use the numbers, then return the block marking it as empty. The empty blocks will be in turn re-filled by the managing thread [32].

This scheme is illustrated in Figure 4.6. We have a number of memory blocks that can be filled with pseudo-random numbers. If any of these blocks is empty, the GPU managing daemon runs and executes a GPU program to generate one memory block of pseudo-random numbers. Once the GPU program completes, the daemon copies this data back to CPU memory and puts the data in the unused block of memory. If all the memory blocks are full, or are being used by worker thread, the daemon blocks execution until a memory block becomes free. On the consumer side, if a worker thread needs a block of pseudo-random numbers, it asks for any of the available blocks. If one is full, this thread is given exclusive access to this block and uses the numbers for its calculation. If one is not full,

the worker thread blocks until a full block becomes available. Once the worker thread finishes with a block, the block is marked as empty and returned to the pool of blocks to be filled by the daemon.

4.3.3 A Class-Based Library to Support GPU Accelerated Monte Carlo Methods

In designing the software to implement the algorithms discussed in the previous subsections, a number of design choices were made. First, software should be designed to make it easy for a user to add this functionality to their application. Second, the software should be flexible to allow the use different pseudo-random number generators with minimal changes to the software base. Third, the managing thread software should be portable and robust. The software that developed to support Monte Carlo applications achieves each of these goals. In this subsection we detail how these goals were achieved.

The first software goal, ease of integration into existing Monte Carlo codes is done through the specification of simple API that matches up one-to-one with the API for sequential PRNGs. The interface is based on class-based library including a class *randClass* that is directly instantiated by the user. The example code given in Figure 4.7 shows how the multi-threaded thread code for computing π would be modified to use the Monte Carlo libraries. Note that the worker code has hardly changed at all. In place of the unique PRNG state for each worker thread, we have substituted the instance of the class *randClass*.

The argument to the *randClass* constructor determines the underlying PRNG to use. The instance of *randClass* is destroyed when the worker thread exits.

```
#include "randClass.h"

…

// the worker thread modified to use the Monte Carlo library
void *pi_estimate_thread (void *ptr)
{
      threadArgs *thread_arg_ptr = (threadArgs *) ptr;

      // instantiate randClass in each worker thread
      randClass myRand("GPURandLux");

      int num_inside = 0;
      for (int i=0; i< thread_arg_ptr->num_samples; i++) {

            // 1. Generate the random point (X,Y)
            float X = myrand.Random();
            float Y = myrand.Random();

            // 2. Compute the Monte Carlo "function" F
            float F = X*X + Y*Y;
            if (F <= 1.0) num_inside++;
      }

      // Estimate π
      thread_arg_ptr->pi_estimate = 4.0 * (float) num_inside / (float) num_samples;

      // the user must call the exit function before exiting the worker thread
      myRand.exitFunction();

      // exit thread
      pthread_exit(NULL);
}
```

Figure 4.7 A version of the example worker thread from Figure 4.5 modified to use the Monte Carlo class-based library. This library enables the user the option of using pseudo-random numbers generated asynchronously on the attached GPU.

The second software goal, that of having the flexibility to add different PRNG generators to the Monte Carlo library is done by defining an abstract base class, *randNumGenerator*, and

an API that can be used by the managing thread independent of the underlying PRNG. The interface defined by this class includes the API shown in Figure 4.8.

```
class randNumGenerator {
  public:
        virtual int getBlockSize() = 0;
        virtual void init() = 0;
        virtual void generateBlock(float *) = 0;
    …
}
```

Figure 4.8 The API defined for the abstract base class *randNumGenerator* .

This base class can then be used to implement a derived class to implement any of a number of PRNG. Of course, we are most interested in developing a GPU accelerated scheme. The interface includes a method that returns a preferred block size. This preferred block size is used by the managing thread to determine the memory size required to hold the generated pseudo-random numbers. The block size is determined by the PRNG because this size depends on the particular algorithm and specific characteristics of the GPU hardware. The virtual method *init()* does any required one-time initialization of the PRNG. For example, the initialization of the PRNG state tables for each thread run on the GPU would be done here. Finally, the method *generateBlock(float*)* fills the memory block at that pointer with a block size of pseudo-random numbers generated by the implemented method. For the GPU version, the implementation of this method would have to do everything required to copy data back and forth to the GPU, compile the GPU

program, and initialize and run the GPU program. Again, this class, *GPURanLux*, which is a derived class from *randNumGenerator* is not instantiated by the user. Instead, the user instantiates *randClass*, and the *randClass* constructor take care of creation and destruction of the particular pseudo-random number generator used.

We now discuss the implementation of the class *randClass*. The user interface to this class is designed to be simple and easy to use, but the class has a number of subtle features in its implementation. As shown in Figure 4.7, each thread has its own instance of the class. However, all these class instances make use of shared data and a single thread that manages the generation of the pseudo-random numbers. To see how this is accomplished, in Figure 4.9 we show a small portion of the class definition.

The idea is that the first instance of the class is the one that allocates all the buffer space for the pseudo-random numbers (in ***buffers*) and starts up the managing thread. A number of variables are declared as static (i.e., global to the class). In this way there is only one copy of each of these variables and so that each instance has access to these variables. Once started, the managing thread asynchronously fills the buffers based on a version of the producer/consumer algorithm. The static semaphores are used to unblock and block the managing thread based on whether there are buffers to be filled or if all the buffers have been filled. Each instance of the class uses the shared mutex to claim or release a buffer of numbers. Each instance keeps track of its current buffer and where it is in the buffer with the non-static variables *currentBuffer* and *bufferPointer*.

```
class randClass {
  public:
        // constructor, the string argument selects the particular PRNG
        randClass(string);
        // returns a uniform pseudo-random number
        double Random();
        // An exit function that each instance must call
        void exitFunction();
  private:
        // static buffer and thread variables
        static cl_float **buffers;
        static pthread_t randThread;
        …
        // static synchronization variables
        static sem_t *emptySemaphore;
        static sem_t *fullSemaphore;
        static pthread_mutex_t modifyMutex;
        …
        // non-static variables, i.e., unique to each instance of randClass
        int currentBuffer;
        int bufferPointer;
  …
}
```

Figure 4.9 A portion of the class definition for *randClass*.

Another potentially tricky part of the implementation is the destruction of the class instances. In particular, the managing thread and the buffer memory should be released only when the last instance of the class is destroyed. This is ensured by having the user call the method *exitFunction()* prior to exiting the worker thread and the class destructor being

called (typically because the class instance goes out of scope). The correct way to do this is shown in the code segment in Figure 4.7. The exit function keeps track of the number of individual instances. When the last instance is being destroyed, this method gracefully cancels the managing thread and frees all the allocated static memory. Following the call to this function, the worker thread can exit safely and the destructor called. Following the call to the exit function by the last instance of the class, if another thread instances *randClass* then the buffer memory would be re-allocated and the managing thread restarted.

4.3.4 Ensuring Statistical Independence

A subtle, yet important issue is the statistical independence of the pseudo-random numbers used by all the Monte Carlo application threads running on a hybrid, parallel computing architecture. There are two places where this issue arises. First, on the GPU each work item executes as an independent thread. Clearly, each of these work items must have a state table for its PRNG that ensures this independence. Second, if we are running the application in a distributed memory environment, the state tables used on different nodes must be independent.

The first issue, the multiple state tables on a single GPU is solved by initializing the state tables with independent data. Then following the execution of the GPU program, both the generated pseudo-random numbers and the current state tables are copied back to the CPU. In this manner, the next time the GPU program is executed, the current state tables can be

restored to the GPU. In Figure 4.23 we show the correct convergence of a Monte Carlo calculation done on a single node demonstrating the independence of these state tables.

The second issue, what to do in a distributed memory environment is solved by using an offset to the state table that is derived from the node number. This number can be obtained from an MPI call, and then the state tables can be offset so that the state tables on each node are independent. This independence is illustrated in the radiative heat transfer results shown in Figure 4.32.

In the following subsections we discuss the actual implementation of the GPU-accelerated PRNG and develop a detailed analysis of its performance. Using this analysis one can select a buffer size for the *randClass* instances that optimizes the overall performance of the implementation. The actual implementation of this PRNG is done in the derived class *GPURanLux*. As optimizing the performance of these algorithms is particular to this implementation, this class internally determines a buffer size to use and returns this size to *randClass* through the *getBlockSize*() method as defined in Figure 4.8.

4.4 Analysis of GPU-accelerated Pseudo-random Number Generation

In this section we develop a theoretical model to analyze the performance of our GPU-based pseudo-random number generation framework. For this analysis, we consider only the thread that manages the GPU kernel that is used to compute the blocks of pseudo-random numbers. We assume that the computation time is not constrained by the time

required by the Monte Carlo threads that consume the numbers generated by the GPU thread. In this case, the time required by the framework is completely limited by the time required to compute the pseudo-random numbers on the GPU.

4.4.1 Data Transfer between GPU and CPU

We first consider the problem of modeling the time to read and write data between the CPU memory and the GPU memory. As has been noted elsewhere [33], a linear model can accurately represent the time required to transfer data between these memories as a function of the amount of data transferred. We show experimental results for the measured transfer times for both writing from the CPU memory to the GPU memory and reading from the GPU memory to the CPU memory. Note that the linear approximations differ slightly.

Using the GPU to generate pseudo-random numbers involves three main factors: the transfer of state tables from CPU to GPU memory, the actual computation of the pseudo-random numbers on the GPU stream processors, and finally the copying back of the state tables and the pseudo-random numbers from the GPU to CPU memory.

The memory transfer time between the CPU and GPU and back again can be modeled by a linear dependence with respect to the amount of data transferred. Accordingly, if we denote the time required to copy m bytes of data from the CPU to the GPU by $T_{CPU \rightarrow GPU}(m)$ and the time required to copy m bytes of data from the GPU to the CPU by $T_{CPU \rightarrow GPU}(m)$, we have the linear relations

$$T_{CPU \rightarrow GPU}(m) = t_{CG}\, m + t_s \qquad (4.5)$$

$$T_{GPU \rightarrow CPU}(m) = t_{GC}\, m + t_s.$$

In these formulae t_s is a "start up" time for the copy, t_{CG} is the incremental time required to copy each addition byte of data from the CPU to the GPU, and t_{GC} is the incremental time required to copy each addition byte of data from the GPU to the CPU. These constants are architecture dependent and can easily be measured. For example, for a ATI Radeon HD 5750 GPU attached machine used for the results presented in the experimental section of the π value calculation in this chapter, we obtained the data shown in Figure 4.10.

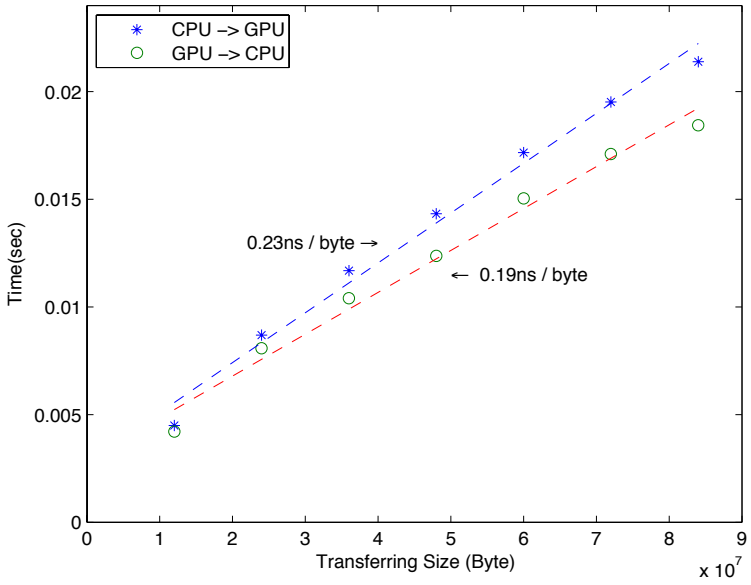

Figure 4.10 Experimental results from a ATI Radeon HD 5750 GPU attached machine showing the time (in seconds) required transferring data between the CPU and the GPU (and visa versa) as a function of the number of bytes transferred. Note the different incremental transfer rates to and from the GPU. [32].

We also obtained the data for the Athena system used for the results presented in the experimental section of the RHT simulation in this chapter, and it is shown in Figure 4.11.

Using a linear least squares fit to the data shown in Figure 4.10 and Figure 4.11, we obtain the values for the constants in Equation 4.5 as shown in Table 4.1.

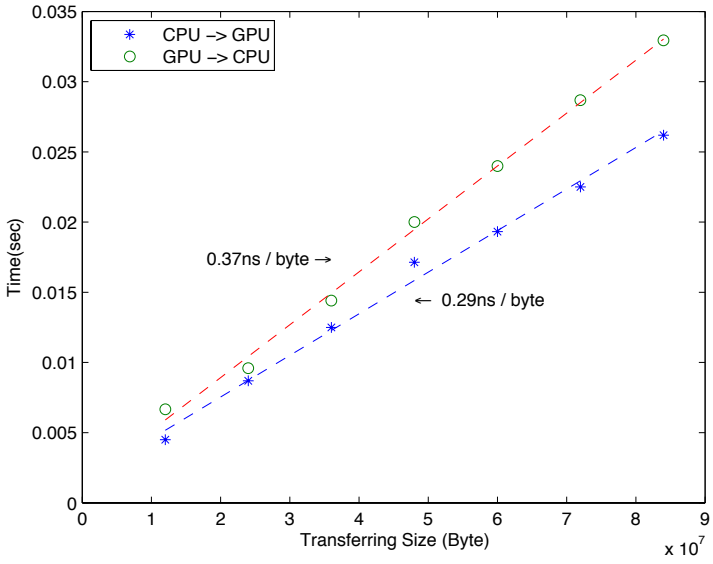

Figure 4.11 Experimental results from the Athena system showing the time (in seconds) required transferring data between the CPU and the GPU (and visa versa) as a function of the number of bytes transferred. Note the different incremental transfer rates to and from the GPU.

Before starting the next aspect of computing the pseudo-random number on the GPU, the GPU architecture need to be explained in advance. The number of threads that can execute concurrently on the GPU is limited by the number of available stream processing units. However, the architecture of these stream processing units is important to account

for. The stream processing units are organized into compute units, and the number of threads that are assigned to each compute unit is given by the work group size. For example, for the ATI Radeon HD 5750 used for π value calculation experiment, the number of stream processing units per compute unit is 80. Thus, the work group size used must be at least as large as the number of stream processing units per compute unit. Overall this GPU has 9 compute units for a total of 720 stream processing units. Note that for the Athena systems used for the RHT experiment, the number of stream processing units per compute unit is 32 and the compute unites are 14. Overall this GPU has 448 stream processing units.

Table 4.1 The constants t_s, t_{CG} *and* t_{GC} obtained by a linear least-squares fit to the data in Figure 4.9 and Figure 4.10. These constants are for the ATI Radeon HD 5750 attached machine used for the π value calculation results, and the Athena system, the hybrid computing system used for the RHT experimental results.

Constant	Time (ATI Radeon HD 5750)	Time (Athena)
t_s	2.9*ms/copy*	3.0*ms/copy*
t_{CG}	0.23*ns/byte*	0.29*ns/copy*
t_{GC}	0.19*ns/byte*	0.37*ns/copy*

4.4.2 Computation Model for Pseudo-random Number Kernel Code

To develop an analysis for the computational time required to generate a block of pseudo-random numbers by the GPU thread it is necessary to examine the GPU architecture and GPU kernel code in some detail. An overview of the key section of the GPU thread code

that calls the GPU kernel is shown in Figure 4.12. In Figure 4.13 we give a high-level view of the OpenCL kernel code that is executed by each thread on the GPU.

From the OpenCL pseudo-code shown in Figure 4.12, one can see that the required computation time is comprised of the time required to complete three types of tasks.

```
// Write the pseudo-random number state tables to the GPU memory
queue.enqueueWriteBuffer(PRN_Tab, PRN_Tab_Size, GPU_PRN_Tab);

// Set kernel arguments
Kernel_PRN.setArg(0, KernelCycles);
Kernel_PRN.setArg(1, GPU_PRN_Tab);
Kernel_PRN.setArg(2, GPU_PRNs);

// Iterate by calling the GPU kernel a number of times to compute
// an entire block of pseudo-random numbers
for (int Iter=0; Iter<NumIterations; Iter++){
     // Execute the pseudo-random number kernel on the GPU
     queue.enqueueNDRangeKernel(Kernel_PRN);
   // Read back a partial block of newly computed pseudo-random numbers
   queue.enqueueReadBuffer(PRNs, PRNs_Size, GPU_PRNs);
}

// Read back the pseudo-random number state tables
queue.enqueueReadBuffer(PRN_Tab, PRN_Tab_Size, GPU_PRN_Tab);
```

Figure 4.12 A simplified overview of the OpenCL calls used to compute a block of pseudo-random numbers on the GPU. The variables *PRN_Tab* and *PRNs* are pointers to arrays in the CPU main memory for the pseudo-random number state tables and the buffer of pseudo-random numbers. The variables *GPU_PRN_Tab* and *GPU_PRNs* are pointers to memory on the GPU [32].

First, data must be written from the CPU memory to the GPU memory. This task is

accomplished by calling the OpenCL function *queue.enqueueWriteBuffer*. Second, the OpenCL kernel must be run on the GPU. This task is accomplished by the OpenCL function *queue.enqueueNDRangeKernel*. Note that the kernel arguments are set by the OpenCL calls to the function *Kernel_PRN.setArg*. And, third, data must be read back from the GPU memory to the CPU memory. This task is accomplished by the OpenCL function call *queue.enqueueReadBuffer*.

```
__kernel void KernelPRN( global KernelCycles, global float *PRN_Tab, global float
*PRNs){
// Number of workgroup
int gid = get_global_id(0);

// Number of workgroup size
int global_size = get_global_size(0);

// four-vector used as a return argument for the pseudo-random number generator
float4 randomnr = 0;

// Generate pseudo-random numbers and then copy to GPU PRN buffer
for ( int i = 0; I < KernelCycles; i += 4){
        randomnr = random_generator();
        PRNs[gid   +   (i+0) * global_size] = randomnr.x;
        PRNs[gid   +   (i+1) * global_size] = randomnr.y;
        PRNs[gid   +   (i+2) * global_size] = randomnr.z;
        PRNs[gid   +   (i+3) * global_size] = randomnr.w;
}
```

Figure 4.13 A high-level view of the kernel code run on the GPU. The arguments passed to the kernel include the number of OpenCL function call t be written from the CPU memory to the GPU memory. This task is accomplished by to the pseudo random number block (output). Each GPU thread uses its workgroup number and size to write the numbers it computes to the correct GPU memory location in the *PRNs* buffer [32].

From the above kernel code, the next aspect of computing the pseudo-random numbers on

the GPU is the time to execute the "kernel" (the OpenCL program that contains the instructions that are executed on the GPU). This time can be modeled as consisting of two parts, a "kernel start up time" T_s and a "kernel execute time" which we denote by T_e. The total time to execute the kernel, T_k, is modeled as the sum of these two terms as

$$T_k = T_e + T_s. \tag{4.6}$$

The pseudo-random number generator works by generating a sequence of numbers in a loop—we denote the number of times through the loop, or the "kernel cycle," as n_k. Empirically we determine that T_s can be modeled as a function of n_k by the equation

$$T_s(n_k) = an_k + b, \tag{4.7}$$

where a is an incremental rate measured to be 200*ns/kernel-cyle*, for the ATI Radeon HD 5750, and 100*ns/kernel-cyle*, for the Athena machine. The constant b is a fixed setup time measured to be 0.9*ms*, for the ATI Radeon HD 5750, and 1*ms*, for the Athena machine. Each kernel cycle, the GPU tries to schedule some number of threads, n_{wgs}, called the "working group size" on each compute unit in the GPU. The GPU performance is, however, limited by the number of stream processors that it has per compute unit. We denote this number by p_{wgs} (this number is 80 for the ATI Radeon HD 5750 and 32 for the Athena system). Given that it takes some amount of time to execute the thread, say t_c^{GPU}, then the second term, the "kernel execute time," can be modeled as

$$T_e \ (n_k, n_{wgs}) \ = n_k \, t_c^{GPU} \lceil n_{wgs} / p_{wgs} \rceil, \tag{4.8}$$

where t_c^{GPU} was measured to be 160 *ns/number* for the ATI Radeon HD 5750 attached machine and 400*ns/number* for the Athena system. Using the Equations 4.7 and 4.8 to model the overall execution time for one kernel cycle as given in (4.6), we obtain the black '*' points shown in Figure 4.14.

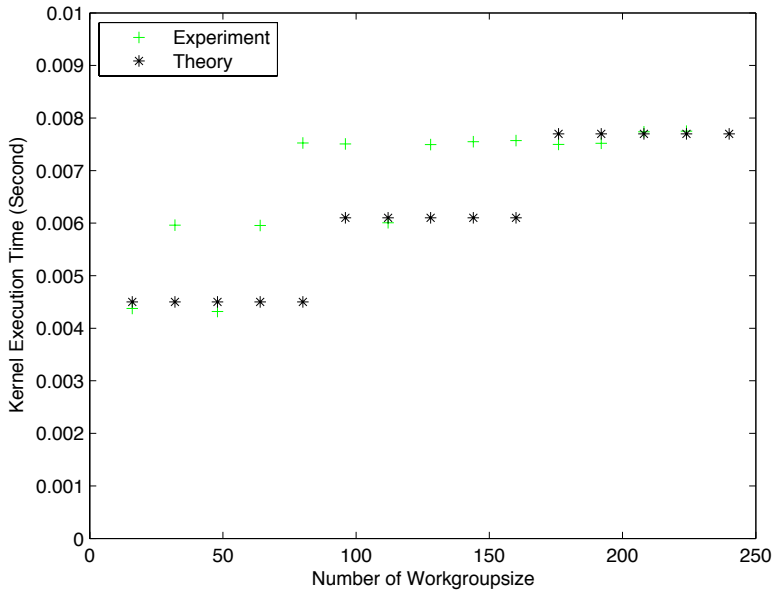

Figure 4.14 The time measured for the kernel to execute as a function of the work group size per work group (or compute unit) for the ATI Radeon HD 5750 attached machine. For this data we fixed the number of work groups to be one. The experimentally measured data from the Athena system is shown as the green '+' points, the modeled times, based on Equation 4.6, are shown as the black '*' points on this graph. In this figure the number of kernel cycles is fixed at 10,000.

Likewise, for the Athena System we can use its machine constants to obtain the black '*' points to model the overall execution time for one kernel cycle. This data is shown in Figure 4.15 where it compared to the experimentally measured times for the Athena machine.

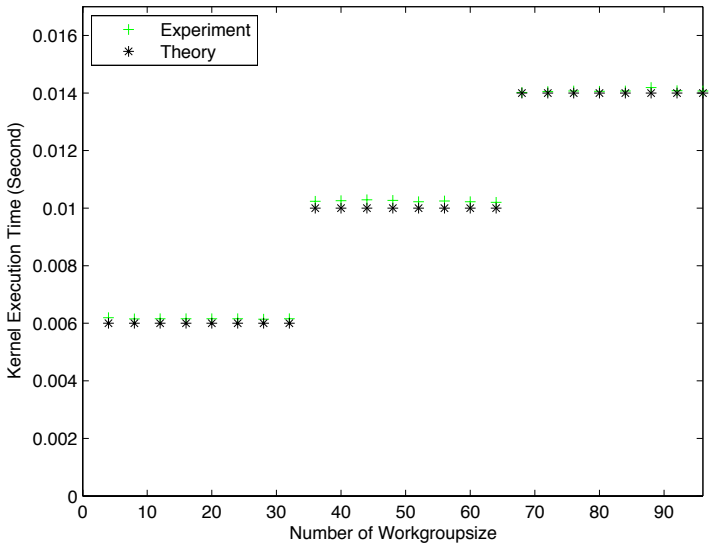

Figure 4.15 The time measured for the kernel to execute as a function of the work group size per work group (or compute unit) for the Athena System. For this data we fixed the number of work groups to be one. The experimentally measured data from the Athena system is shown as the green '+' points, the modeled times, based on Equation 4.6, are shown as the black '*' points on this graph. In this figure the number of kernel cycles is fixed at 10,000.

In Figures 4.14 and 4.15 the number of kernel cycles, n_k, is fixed at 10,000; we then measure the time it takes for the kernel to execute. The measured times are shown as the green '+' symbols in this graph. As the work group size increases beyond multiples of 32

(e.g., 32, 64, and 96) for the ATI Radeon HD 5750 attached machine and 80 (e.g., 80, 160, and 240) for the Athena system, we observe discrete jumps in the measured times as predicted by Equation 4.8.

A good way to parameterize the performance of the pseudo-random generator is in terms of the number of work items, n_{wi}, which is the product of the number of work groups, n_{wg}, and the work group size, n_{wgs}, as follows

$$n_{wi} = n_{wg}\, n_{wgs}\,. \tag{4.9}$$

This a good way to parameterize the scaling of the parallel algorithm because the number of work items, n_{wi}, represents the number of independent "tasks" that are to be executed on the GPU. However, there are two limitations to the number of these tasks that can be executed in parallel. First, only one work group can use a compute unit at a time; hence, the number of work groups that can execute in parallel is limited by the number of compute units on the GPU. We denote the number of compute units on the GPU by n_{CU}. Second, as discussed above, the number of stream processors per compute unit, p_{wgs}, limits the number of threads that can execute at one time on a compute unit.

A complete model of the time required to generate the pseudo-random numbers requires that we also include the time necessary to copy the pseudo-random number seed tables back and forth between the CPU and GPU memory (one table for each work item) and to copy

the pseudo-random numbers from the GPU to the CPU memory. To help amortize the cost of copying the seed tables between the CPU and GPU memories, we iteratively run the GPU kernel code n_i times. Each time the GPU kernel code is run we generate $n' = n_k\, n_{wi}$ pseudo-random numbers. Thus, the total number of pseudo-random numbers generated is given by $n = n_i\, n'$.

Therefore, in our model we have four separate parts to consider: (1) the time to upload the seed tables, T_{seedUp}; (2) the time to download the seed tables $T_{seedDown}$; (3) the kernel execution time, T_k; and (4) the time to download the n' pseudo-random numbers, $T_{GPU \rightarrow CPU}(n')$.

On the first line of the program outline given in Figure 4.11, the data written to the GPU are the pseudo-random number state tables. We use n_{tab} to represent the number of bytes comprising one of the number state tables. We require a unique state table for each independent pseudo-random number generator thread that we run on the GPU. Thus, the time required to write the state tables to the GPU is given by the expression

$$T_{seedUp} = n_{wi}\, n_{tab}\, t_{CG} + t_s\,. \tag{4.10}$$

The time required reading the state tables back from the GPU (the last line of the program segment in Figure 4.12) is given by the expression

$$T_{seedDown} = n_{wi}\, n_{tab}\, t_{GC} + t_s\,. \tag{4.11}$$

In addition, to reading and writing the state tables, we also need to read the pseudo-random numbers generated on the GPU back to the GPU. Then the time required reading these numbers back would be given by the expression

$$T_{numDown} = n_k\, n_{wi}\, t_{GC} + t_s\,. \tag{4.12}$$

After generating pseudo-random numbers on GPU, those numbers need to be downloaded to CPU memory side. Thus the time required downloading those numbers is given by the expression

$$\begin{aligned} T_{numDown} &= n_k\, n_{wi}\, t_{GC} + t_s \\ &= t_{GC}\, n' + t_s\,. \end{aligned} \tag{4.13}$$

Combining these factors, the time to generate n pseudo-random numbers using the GPU, $T_{RN}^{GPU}(n)$, can be expressed as:

$$\begin{aligned} T_{RN}^{GPU}(n) &= T_{seedUp} + T_{seedDown} + n_i\,[T_k(n_k, n_{wgs}) + T_{GPU\rightarrow CPU}(n')] \\ &= T_{seedUp} + T_{seedDown} + n_i\,[T_k(n_k, n_{wgs}) + t_{GC}\, n' + t_s]\,. \end{aligned} \tag{4.14}$$

The seed table size is 28 words, and we require a separate seed table for each work group.

However, given that the number of work groups is at most in the thousands, and the number of pseudo-random numbers that we will be copying back from the GPU is typically in the millions, we can ignore T_{seedUp} and $T_{seedDown}$ in Equation 4.14 and use the following approximation,

$$T_{RN}^{GPU}(n) \approx n_i [T_k(n_k,n_{wgs}) + t_{GC} n' + t_s] . \quad (4.15)$$

Also note that in our formulation of $T_k(n_k,n_{wgs})$, we tacitly assumed that there are an unlimited number of compute units available on the GPU. In practice, of course, the GPU architecture has limited number of compute units, n_{CU}. When the number of work groups is larger than the number of compute units, the extra work corresponding to these additional work groups must be scheduled sequentially on the GPU. To take this effect into account, we introduce a modified kernel execution time function, $T_k^*(n_k, n_{wgs}, n_{wg})$. The modified kernel execution time can be expressed as

$$T_k^*(n_k, n_{wgs}, n_{wg}) = T_k(n_k,n_{wgs}) \lceil n_{wg} / n_{CU} \rceil . \quad (4.16)$$

To compute the speedup of the GPU accelerated algorithm, we need a model for the running time of the sequential algorithm on the CPU. As this running time should depend linearly on the number of pseudo-random numbers generated, we model the time to generate the numbers using the CPU, $T_{RN}^{CPU}(n)$, as:

$$T_{RN}^{CPU}(n) = n_k\, n_{wi}\, n_i\, t_c^{CPU} \qquad (4.17)$$
$$= t_c^{CPU}\, n\,.$$

The pseudo-random number generator used on the GPU is an implementation of RANLUX [42]. Thus, on the CPU we use the GNU Scientific Library (GSL) implementation of this pseudo-random number generator [43]. For the GSL implementation of RANLUX with a luxury level of 0 (gsl_rng_ranlxs0) the value measured for t_c^{CPU} for the ATI Radeon HD 5750 attached machine for the π estimation is 65*ns/number* and for the Athena system used for the RHT experiments is 50*ns/number*. For a luxury level of 2, this value was measured to be 120*ns/number* for the Athena system. Combining the two models, the speedup of generating *n* pseudo-random numbers using the GPU compared to CPU can be computed as

$$S_{RN}(n) = T_{RN}^{CPU}(n) \,/\, T_{RN}^{GPU}(n)\,. \qquad (4.18)$$

This speedup is presented below in Figure 4.16 for the Radeon HD 5750 attached machine and in Figure 4.17 for the Athena system. In those figures we show both the experimentally measured speedup and the theoretical speedup based on the model presented above computed using the machine parameters for the Athena system. Note that we plot the theoretical speedup for two models. The first model assumes an unlimited number of compute units, and the second model is based on Equation 4.16, where the number of compute units is limited to 9 (as for the ATI Radeon HD 5750 attached machine)

and 14 (as for the Athena system).

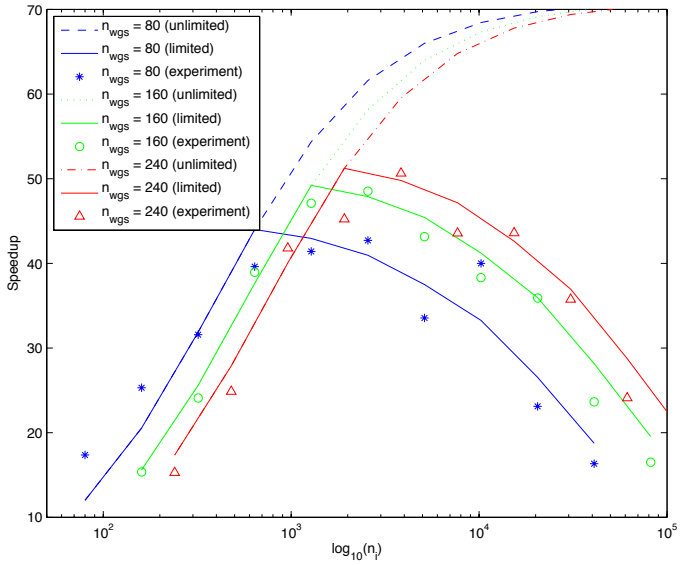

Figure 4.16 Speedup plots comparing the GPU execution time to the CPU execution time for the pseudo-random number generation library for the ATI Radeon HD 5750 GPU. Three different work group sizes (80, 160 and 240) are used. The speedup results are plotted as a function of the number of work items. As explained in the text, the number of work groups and the number of kernel cycles are both varied in order to compute the same number of pseudo-random numbers for each data point. These results are for RANLUX with luxury level 0.

How we obtain the increasing number of work items in Figure 4.16 and Figure 4.17 requires some additional explanation. We use work group sizes, n_{wgs}, of 80, 160, and 240 for the ATI Radeon HD 5750 GPU and 32, 64, and 96 for the Athena system and we fix the number of iterations, n_i, to 10. We adjust n_{wg} and n_k to generate the same number of pseudo-random numbers (245,760,000 for the ATI Radeon HD 5750 GPU, the results shown in Figure 4.16, and 98,304,000 for the Athena machine, the results shown in Figure

4.17). The number of work groups, n_{wgs}, is varied in the following manner. In the case of the Athena machine, when n_{wgs} is 32 and n_i is fixed as 10, the number of work groups, n_{wg}, is increased from 2^0 to 2^9 and n_k is respectively decreased from 600×2^9 to 600×2^0 to generate the same number of pseudo-random numbers. The parameters are changed in a similar manner (as described above) to obtain the results presented in Figure 4.17.

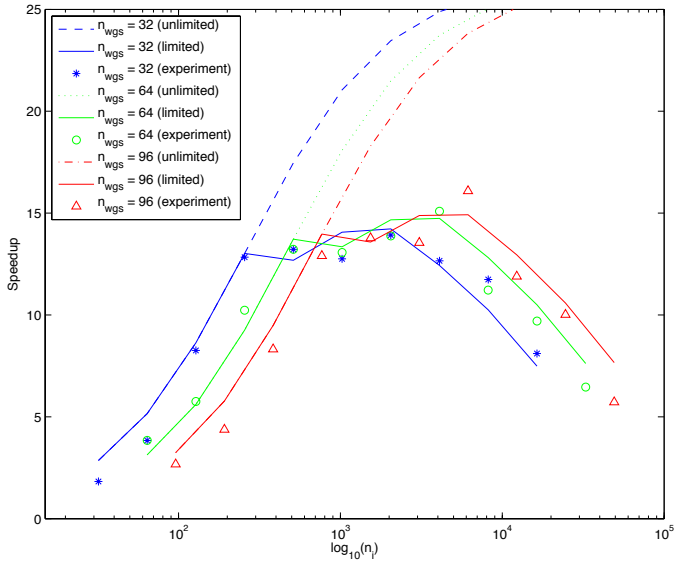

Figure 4.17 Speedup plots comparing the GPU execution time to the CPU execution time for the pseudo-random number generation library for the Athena system. Three different work group sizes (32, 64 and 96) are used. The speedup results are plotted as a function of the number of work items. As explained in the text, the number of work groups and the number of kernel cycles are both varied in order to compute the same number of pseudo-random numbers for each data point. These results are for RANLUX with luxury level 0.

Note that the experimentally measured results agree well with the modified theoretical model. We include in the plot the theoretical model for an unlimited number of compute

units. As one can see, the overall speedup is limited and ultimately approaches an asymptotic value. This limit results primarily from the time required to copy back the pseudo-random numbers from the GPU to the CPU memories.

4.4.3 Optimization of the GPU Performance

The speedup plots shown in Figures 4.16 and 4.17 have the undesirable property that the speedup is not uniformly good as a function of the number of work items. To help address this problem, we can take advantage of the excellent correspondence between the analytical performance model and the experimental results. As there are many possible parameter choices that generate similarly sized blocks of pseudo-random numbers, the analytical model can be used to select parameters that optimize the GPU performance. Thus, one can think of the problem of optimizing the GPU performance as an optimization problem subject to constraints. The constraints might be, for example, that we want to generate a fixed number, n, of pseudo-random numbers with a fixed number of kernel iterations, n_i. As minimizing the running time on the GPU maximizes the speedup, we can express this optimization problem as the following problem

$$min\ T_{RN}^{GPU}(n),\ s.t.\ n,\ n_i\ fixed. \tag{4.19}$$

As this function is discontinuous and the parameter choices are discrete, we consider an ad hoc approach toward solving this optimization problem. First, as we know that n must be large in order to amortize the cost of reading the numbers back from the GPU, we can fix n

to be the same value used for the experimental results in the preceding subsection. Thus, we fix n at 245,760,000 for the ATI Radeon HD 5750 GPU and at 98,304,000 for the Athena machine (the values used for the results presented in Figures 4.16 and 4.17). In addition, we fix the number of kernel iterations, n_i, to 10, as was used in these figures.

Given these constraints, one of the first things to notice from Equation 4.16 is that the number of work groups, n_{wg}, should be an exact multiple of the number of compute units, n_{CU}. This choice maximizes the number of pseudo-random numbers generated on the GPU without increasing T_k. Choosing these values optimizes the overall speedup for a range of choices for n_{wg},

A second thing to notice from Equation 4.8 is that when executing the kernel, there is a fixed startup time b. If we can increase the work group size, that is the number of work items on each compute unit, n_{wgs}/n_{wg}, we can amortize this startup cost. You can see this on the graphs in Figures 4.14 and 4.15 as this approach corresponds to the measured times on right hand side of each graph. However, if we fix the number of pseudo-random numbers generated, we can only do this by decreasing the number of kernel cycles, n_k. We do not want to have the number of kernel cycles too few as the startup cost b would again begin to dominate. To do this optimization, we have implemented a Matlab program for the analytical model of the running time, and by varying these parameters within a driver program we can determine a minimum running time for fixed n.

The speedup plots presented in Figure 4.18 and 4.19 show the speedups for the two architectures for the optimized parameter values chosen using the above-described method. Recall that the number of pseudo-random numbers generated is the same as for the results in Figures 4.16 and 4.17. In addition, the number of iterations, n_i, is again 10 for these results. Thus, we can directly compare the speedups in these graphs to demonstrate the advantage of optimizing for the parameter choices.

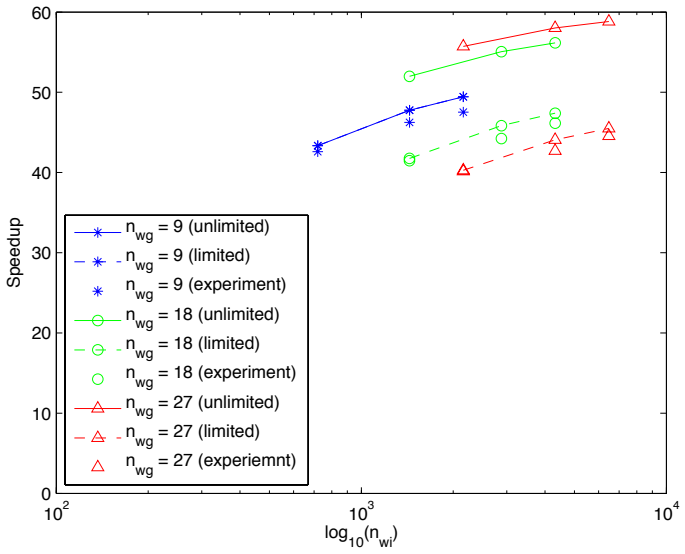

Figure 4.18 The experimentally measured speedup for generating pseudo-random numbers using ATI Radeon HD 5750 using parameters optimized based on the analytical running time model. The experimental results are compared to the model. The speedup curves labeled "unlimited" are computed for a model with an unlimited number of compute units.

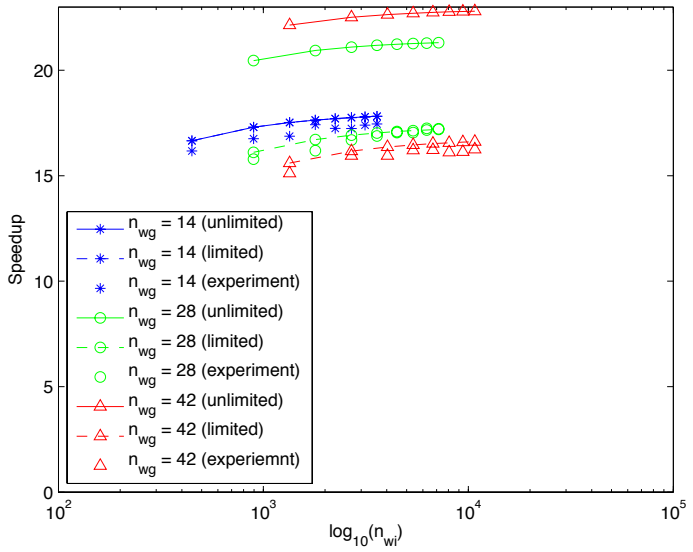

Figure 4.19 The experimentally measured speedup for generating pseudo-random numbers using Nvidia S2050 using parameters optimized based on the analytical running time model. The experimental results are compared to the model. The speedup curves labeled "unlimited" are computed for a model with an unlimited number of compute units.

As discussed above, in Figure 4.18 we show of the speedup of generating pseudo-random number after optimization for ATI Radeon HD 5750, and in Figure 4.19 we show the results for the Nvidia S2050 (the Athena machine). For the ATI Radeon HD 5750, the number of compute units is 9 so we set the number of work groups, n_{wg}, to be 9, 18, or 27 (a multiple of the number of compute units). We then increase the work group size, n_{wgs}, from 80 to 240 in increments of 80 (so that this number is evenly divided by 80, the number of thread processors per compute unit). Based on these choices, the number of kernel cycles is chosen to generate the same number of pseudo-random numbers, 245,760,000.

For the case of the Nvidia S2050 (the Athena machine) the number of compute units is 14 so we set the number of work groups, n_{wg}, to be 14, 28, or 42 (a multiple of the number of compute units). And, as before, the work group size, n_{wgs}, is varied from 32 to 256 in increments of 32 (so that this number is evenly divided by 32, the number of thread processors per compute unit). The total number of pseudo-random numbers generated is fixed at 98,304,000 for the results presented in Figure 4.19.

As one can see for the results presented in 4.16 and 4.17, when compared to the results in Figures 4.18 and 4.19, the overall speedup achieved with the optimized parameters is more consistent and always greater than for the un-optimized parameters. The maximum speedup obtained is not dramatically increased, but the overall performance is clearly superior. Currently, the optimized parameters are computed within the code; however, this would clearly be a useful feature and could be a topic for future work.

4.4.4 An Analysis for the Overall Running Time

To analyze the speedup of the GPU accelerated speedup with multiple threads, we consider the speedup of GPU accelerated Monte Carlo simulation. We can compute a speedup for the simulation using a hybrid-computing scheme (i.e., using the GPU to compute the pseudo-random numbers) relative to using solely the CPU. First, we consider the time required on a single thread using the CPU to compute the pseudo-random numbers. Let the required number of pseudo-random numbers be denoted by n. Note that the number of pseudo-random numbers required increases linearly with the number of samples and

photons used in the π value calculation and the RHT simulation. We can express the sequential simulation time, $T_S(n)$, as

$$T_S(n)=T_{RN}^{CPU}(n)+T_{MC}(n), \tag{4.20}$$

where $T_{MC}(n)$ is a time required for the Monte Carlo simulation (excluding the pseudo-random number generation) and $T_{RN}^{CPU}(n)$ is the time for generating pseudo-random numbers using the CPU.

An analysis of the time required for the Monte Carlo simulation running in parallel using multiple threads and a single GPU to compute the pseudo-random numbers is more complex as it involves the parallel execution of the thread managing the GPU and the GPU program itself [32]. As discussed in the preceding subsection, the time to compute a pseudo-random number on the GPU is a complex function of a number of factors specific to the GPU used and its configuration. We can, however, simplify the analysis by considering only a single parameter, the block size B. As a general rule, the efficiency of computing the numbers increases with the block size. However, the exact efficiency depends on the number of work items, n_{wi}, the number of iterations, n_i, and the number of kernel cycles, n_k (as discussed above and in detail in reference [32]). Given these parameters, the block size B is given by the formula

$$B = n_k\, n_i\, n_{wi}\,. \tag{4.21}$$

As noted earlier, the number of work items, n_{wi}, can be configured in a number of different ways by selecting different values for the number of work groups, n_{wg} and the work group size, n_{wgs} as the number of work items is the product of these two parameters. We assume that these parameters are chosen to maximize the efficiency of computation on the GPU. Given these definitions, the parallel simulation time, $T_P(p,pn,B)$, can be expressed as

$$T_P(p,np,B) = pT_{RN}^{GPU}(B) + \max\{p(\lceil n/B \rceil - 1)\, T_{RN}^{GPU}(B), \lfloor n/B \rfloor\, T_{MC}(B) + T_{MC}(n - \lfloor n/B \rfloor B)\}, \quad (4.22)$$

where $T_{RN}^{GPU}(B)$ is a time to fill in the pseudo-random numbers using the GPU for the block size B, and p is the number of threads.

The timeline for the Monte Carlo simulation for a single thread in the above equation is illustrated in Figures 4.20 and 4.21. Figure 4.20 shows the simulation time when the pseudo-random number block generation time takes longer than the Monte Carlo time; note how the Monte Carlo part of the code is blocked while it waits for a new block of pseudo-random numbers is generated by the GPU.

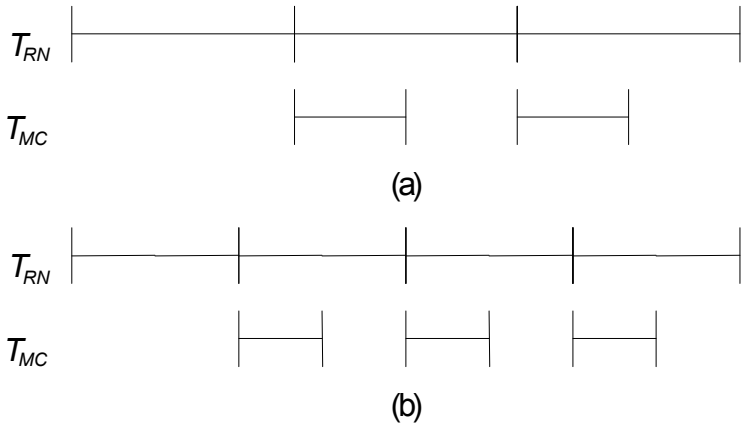

Figure 4.20 A timeline showing what the Monte Carlo thread and the pseudo-random number thread manager are doing relative to each other when $T_{RN}(B)$ is longer than $T_{MC}(B)$. In (a) we show the case the block size is larger than in (b).

To illustrate where the "max" arises in Equation 4.22, in Figure 4.21 we illustrate the simulation time when the $T_{RN}(B)$ (the time to generate a block of pseudo-random numbers on the GPU) is shorter than the time $T_{MC}(B)$.

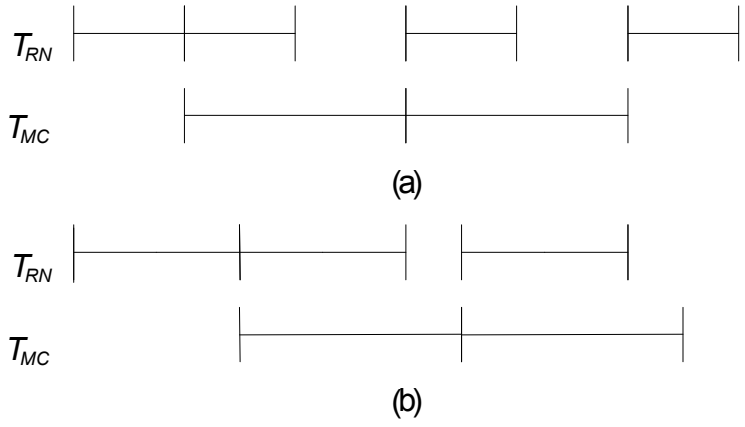

Figure 4.21 Illustrations of the simulation timeline showing the Monte Carlo thread and the pseudo-random number managing thread. In (a) the block size is large enough that $T_{RN}(B)$ can generate a new block before $T_{MC}(B)$ completes. In (b) we illustrate the case when the block size is even larger than in (a).

In Figure 4.22 we illustrate the case when the simulation is multi-threaded with multiple threads consuming the pseudo-random numbers generated. Again, note how the "max" in Equation 4.22 is used to describe this case.

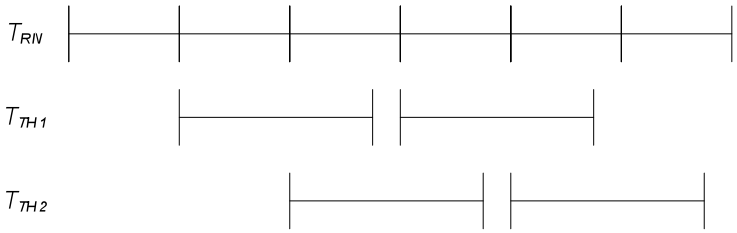

Figure 4.22 An example illustrating multiple threads executing the Monte Carlo part of the code using the pseudo-random numbers generated from the single thread managing the GPU.

Finally, the overall scaled speedup, *SS,* for the hybrid version the Monte Carlo simulation using the GPU-accelerated libraries can be expressed as:

$$SS\,(p,np,B) = p\;T_S\,(n) \,/\, T_P\,(p,np,B). \tag{4.23}$$

4.5 Using the GPU Accelerated Pseudo Random Number Code in Monte Carlo Applications

In this section we examine the overall performance of Monte Carlo applications that take advantage of our GPU-acceleration scheme. We consider two model applications. The first is the toy problem we introduced earlier, a numerical integration scheme to estimate the

value of π. The second model application is a complex, real-world application to simulate the radiative heat transfer for a three-dimensional domain using the Monte Carlo ray tracing method. In addition to presenting an analysis of the performance of these applications, we also consider the issue of the statistical independence of the pseudo-random numbers used.

4.5.1 Statistical Independence of the Pseudo-Random Numbers

In using a pseudo-random numbers generator, one would like some assurance that the pseudo-random numbers are statistically independent. There are two approaches that we have used. The first check is a simple statistical analysis of average function values computed by the Monte Carlo method. The second way is to use a much more rigorous empirical statistical test of all the pseudo-random numbers generated using the software library TestU01.

The basic idea for the first approach, using the computed function values, is as follows. First, we subdivide the set of computed samples (i.e., function values) into some number of independent subsets and compute the mean of each of these subsets. For example, suppose with our toy problem to estimate π, we use the multi-threaded code we discussed earlier. Each core will compute an estimate for pi. Suppose the cores are labeled $i = 1, \ldots, p$. Suppose the mean estimate from each core i is μ_i. A consequence of the Central Limit Theorem [58] is that these means should (roughly) follow a normal distribution. In addition, as we increase the number of samples used to compute each mean, the standard

deviation of these means should decrease. In particular, this standard deviation should decrease with the square root of the number of samples.

This first approach can be used as a simple sanity check for the pseudo-random numbers generated on the GPU. We can compare the results obtained on the GPU with results using a standard, high quality PRNG on the CPU. In Figure 4.23, we show the computed standard deviations for both the CPU and GPU versions of the code for the toy problem to estimate π. As shown, the standard deviations for both versions decrease as the square root of the number of samples.

A second, much more rigorous test, is to use the software library TestU01 [35] to employ a suite of empirical statistical test on the pseudo-random numbers generated by our GPU code. The TestU01 library contains three sets of tests: SmallCrush, Crush, and BigCrush. These tests apply a variety of statistical tests (respectively 15, 144, and 160 tests) to check for correlations in large sequences of pseudo-random numbers. We tested our GPU implementation of RANLUX using these three test batteries.

One modification that we had to make to the pseudo-random numbers generated on GPU to use these tests is as follows. The GPU implementation of RANLUX is done in single precision, and thus has only 24 bits of resolution in the mantissa. The TestU01 assumes that the pseudo-random numbers are double precision. Thus, to convert our numbers to double precision, the additional mantissa bits in the double have to be filled with

statistically independent values. Two single precision numbers were combined to obtain one double precision value for the test. Once this was done, then the pseudo-random numbers passed the SmallCrush battery. For the Crush and BigCrush battery, all of the tests were passed except for one of tests from each battery. The passing of these tests indicates a high degree of confidence in the statistical independence of the pseudo-random numbers generated by the GPU implementation of the RANLUX algorithm.

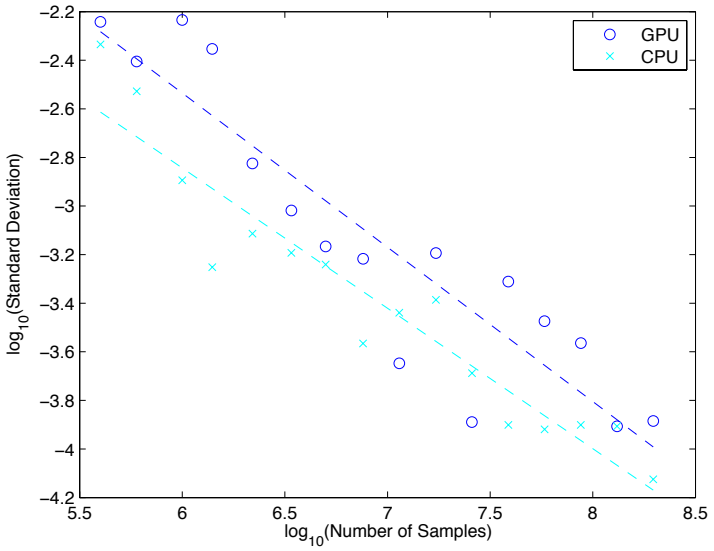

Figure 4.23 The standard deviation of the computed means for the Monte Carlo toy application as a function of the number of samples used. The standard deviations should decrease as the square root of the number of samples (the dashed lines in the figure). This satisfies a simple first test for statistical independence of the pseudo-random numbers used in the calculation.

4.5.2 Performance of the Monte Carlo Code for the Toy Application

To observe the performance improvement for using the GPU to generate the pseudo-random numbers in a simple Monte Carlo application we first consider our toy problem, a numerical integration scheme to estimate the value of π. In measuring the performance of this model, we are using the ATI Radeon HD 5750 GPU. This GPU contains 9 compute unites and each compute unit consists of 80 processing elements. The overall software framework is illustrated in Figure 4.6. The framework has a managing thread that fills empty memory blocks with pseudo-random numbers. The threads that use numbers for the Monte Carlo application access these full memory blocks via a shared-memory producer/consumer implementation. The pseudo-random numbers can be generated either on the CPU or on the GPU. When using the CPU, the RANLUX numbers are generated by routines from the GNU scientific library [34]. When using the GPU, memory blocks are filled with the method described in Figure 4.12 and Figure 4.13. The simulation times for the CPU and the GPU are compared and the resulting speedup is shown in Figure 4.24. This graph shows that generating the pseudo-random numbers using the GPU makes the Monte Carlo application run significantly faster when compared to using the CPU.

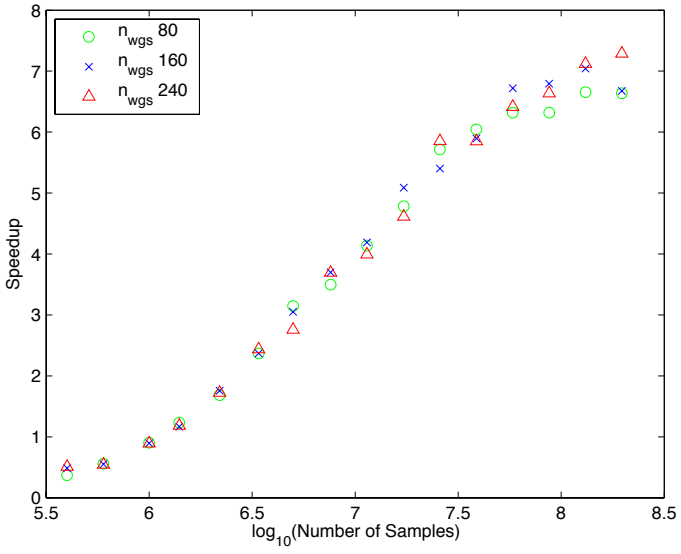

Figure 4.24 The speedup of a simple Monte Carlo simulation using the GPU acceleration scheme with work group sizes of 80, 160 and 240 [32]. The speedup is relative to using the CPU is plotted as a function of the number of pseudo-random number samples used by the Monte Carlo method.

To verify the adequacy of our model, we present the estimation of π using Monte Carlo method as a simple application. The estimated value of π is presented with error and mean in Figure 4.25. The theoretical value is within the error range of experimental result. Also with more samples, the experimental result approximates to π.

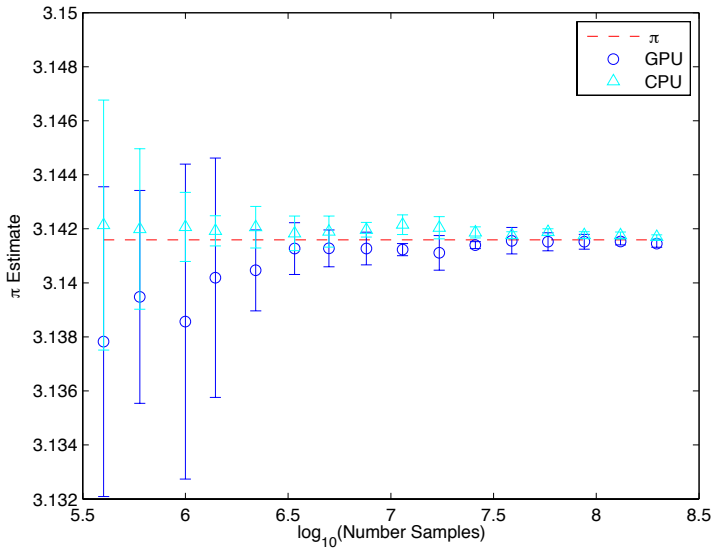

Figure 4.25 Convergence of the theoretical and experimental estimation of π by numerical integration with the Monte Carlo framework as a function of the number of samples used.

4.5.3 Radiative Heat Transfer—A Complex, Real-World Application

The other application that we consider is the RHT simulation with PMC method. The computational cost of modeling RHT effects accurately can be extremely high due to its highly nonlinear and non-local nature [36]. This nonlinearity arises because the RHT rates typically depend on the fourth power of the temperature [37]. Thus, applications that involve the computation of these rates, such as with combustion, are highly sensitive to the accuracy of these temperature calculations. Omitting the RHT effects from simulations in such applications can lead to inaccurately computed temperature profiles, which in turn affects the stability and the accuracy of the calculation of other variables [38].

The non-local nature of RHT comes from the fact that the photons that carry radiation (and energy) can be absorbed far from the physical position that they are emitted. Because of these non-local effects, conservation laws cannot be applied over an infinitesimal volume, but instead must be applied over the entire computational domain. The Photon Monte Carlo (PMC) method can be effectively used in the solution of thermal radiation problems [39]. This method is based on a model of radiative energy traveling in discrete packets (like photons) and the computation of the effect of these photons while traversing, scattering and interacting with matter within the computational domain. Advantages of this method include: an ability to deal with complex geometries; an ability to handle non-uniform temperature fields; the ability to include photon scattering; and the ability to employ a great variety of methods to include specialized radiative properties of the enclosure or the transport domain [40].

4.5.3.1 The Photon Monte Carlo Method

The Photon Monte Carlo (PMC) method is a sampling method based on simulating the movement and absorption of photon bundles (rays) through a discretized computational domain. The advantage of this approach, as opposed to other RHT approximation schemes, is that its overall computational cost grows slowly as a function of the complexity of the RHT problems [39]. An additional advantage is that increased accuracy can be obtained by using larger numbers of photon bundles. Hence, the PMC method is well suited to radiation calculations that include complex geometries, non-trivial absorption properties, and singular effects such as scattering. In this section we describe the basic

PMC algorithm. We also review how pseudo-random numbers are used within the PMC algorithm and ultimately in the overall RHT simulation.

In numerical simulations that include a RHT component, the computational domain contains a participating medium (a material that both emits and absorbs photons). The PMC method traces a statistically significant sample of photon bundles from their point of emission within the medium to a point of absorption within the medium or its boundary. When the photon bundle is absorbed, its energy is added to the local energy of the absorbing element within the discretized medium. With this approach, the PMC method is able to calculate the energy gain or loss for every element within the computational domain.

The tracking of the photon bundles through the computational domain requires that the PMC algorithm model several types of element interactions. These interaction types are illustrated in Figure 4.26. On the left, Figure 4.26 (a), we show a photon bundle entering an element and being absorbed within the element. In the middle, Figure 4.26 (b), we show a photon bundle entering an element and being scattered off of a particle within the element. On the right, Figure 4.26 (c), we show a photon entering an element, traversing the entire element, and exiting the element to enter a neighboring element. For the computational results presented in this paper we employ a software framework capable of modeling these element interactions and tracing the photon bundles through a computational mesh [41].

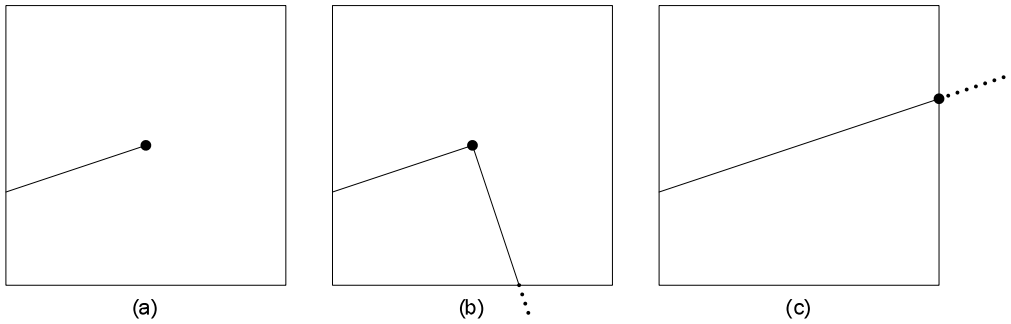

Figure 4.26 Possible photon interactions within an element. In (a) we show the photon bundle being absorbed within the element, in (b) the photon is scattered off a particle within the element, and in (c) the photon bundle is transmitted through the element.

The PMC algorithm can be used to solve a wide range of RHT problems. In this experiment, we use the algorithm to solve a simple model problem. The model problem we use is a three-dimensional rectangular solid domain with two plates with different temperatures on opposite walls and periodic boundary conditions in the other two orthogonal dimensions [39]. The RHT problem that we solve involves computing the temperature as a function of position between the two plates. For a fixed computational mesh the overall computational work grows linearly with the number of photon bundles that we track through the domain. By increasing the number of photon bundles used in the simulation, we can examine the "scaled" speedup of the simulation when run on a hybrid parallel computer architecture. In addition, as we increase the number of photons, the accuracy of the computed solution improves and we can verify the statistical independence of sampling done by the photon bundles by looking at the convergence of the temperature from independent samples.

4.5.3.2 The Structure of the Radiative Heat Transfer Code

For every photon bundle, the PMC algorithm must determine a point of emission, a direction of emission, a wavelength, a point of absorption, and various other properties that are independently chosen from probability distributions. Because of the large number of required pseudo-random numbers, a profile of the PMC code running a standard CPU shows that 90% or more of the computational time is spent generating these numbers. This percentage can be even higher as the complexity of the radiative properties and the accuracy required from the simulation increases. Based on this observation, a more efficient scheme for generating these pseudo-random numbers can dramatically improve the overall performance of the PMC algorithm.

The scheme to improve the performance of the PMC algorithm is similar to the PathSim2 experiment. The RHT computation can be allocated to multiple threads and be distributed to multiple CPU cores. The pseudo-random numbers required for the PMC algorithm for the RHT simulation can be generated on the GPU.

In the following section, we present a GPU accelerated pseudo-random number generation algorithm. In addition, we present an analysis of the running time of this algorithm and show how it achieves much of this potential efficiency gain for the RHT simulation.

4.5.3.3 Experimental Results for the Radiative Heat Transfer Problem

For the experimental results presented in this section, we use the Athena system in Virginia Tech [44]. Athena is a cluster system with GPUs and large RAM memory. The system is made up of 16 nodes and each node consists of 4 octa-cores. Each node also has one NVIDIA S2050 GPU. Each GPU contains 14 compute units, and each compute unit consists of 32 processing elements [45]. For all the results presented in this section, a luxury level of 2 is used for the RANLUX pseudo-random number generators.

To measure the scaled speedup on this hybrid architecture, the CPU-only version of the simulation is first run and timed on a single CPU. These measurements are compared to the running time of the simulation on the hybrid architecture for a scaled instance of the problem. In Figures 4.27 and 4.28 we show the speedup of the RHT simulation using the GPU accelerated random number generator and a single CPU as compared to the solely CPU-based version. In Figure 4.27, note that when the number of photons used by the simulation is small, the scaled speedup is limited by the time required to generate the first block of pseudo-random numbers (e.g., see the timelines in Figures 4.21). For larger numbers of photons, this initial time is amortized as many blocks are used. Note that by using a smaller block size, this transition to improved speedup occurs for a smaller number of photons. The results in Figure 4.27 correspond to the timelines shown in Figure 4.21 where the asymptotic speedup is determined by the relative amount of time spent in RHT portion of the simulation.

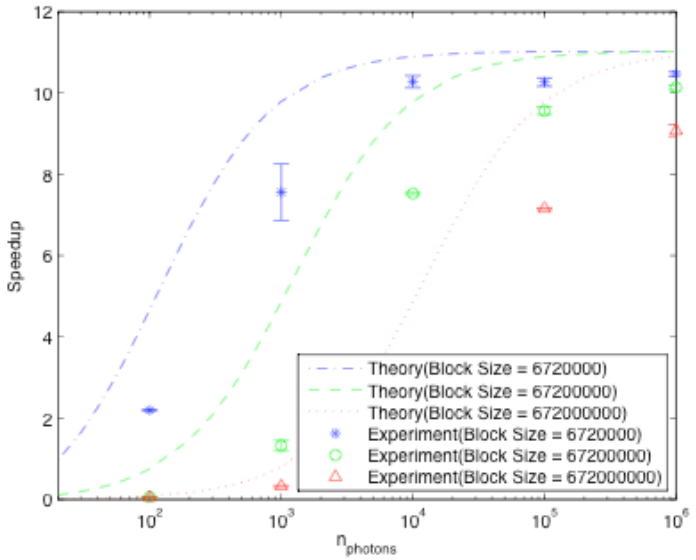

Figure 4.27 The speedup for the GPU-accelerated simulation run on a single CPU for large block sizes. The photon numbers are increased from 10^2 to 10^6 in order to vary the workload. The work group size is 96, the number of work groups is 14, and the number of kernel cycles is 500. To change the block size, the number of iterations is respectively set to 10, 100 and 1000.

In Figure 4.28 we show the speedup for the timeline case shown in Figure 4.20. In this case, the asymptotic speedup is limited by the speedup of pseudo-random numbers being generated on the GPU. As shown in the figure, this speedup decreases with smaller block sizes.

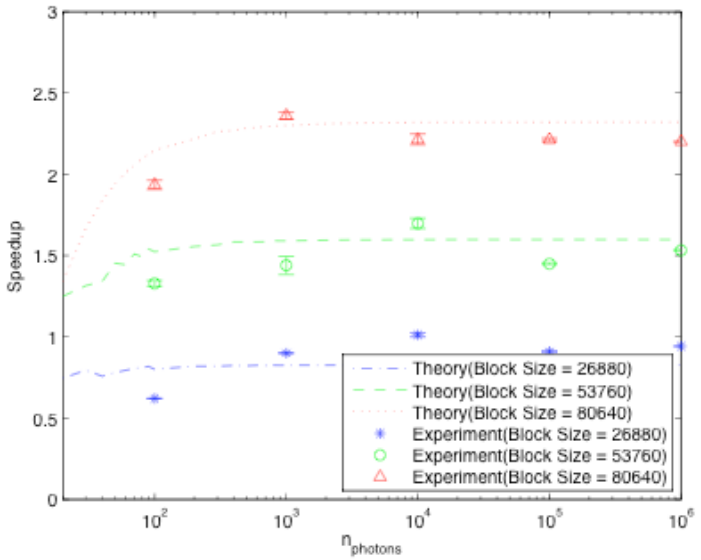

Figure 4.28 The speedup of the RHT simulation for on a single CPU for small block sizes. The photon numbers are increased from 10^2 to 10^6 in order to vary the workload. The work group size is 96, the number of work groups is 14 and the number of iterations is 1. To change the block size, the number of kernel cycles is set to 20, 40 and 60.

The scaled speedup obtained with multiple threads on a single node is in Figure 4.29. This plot shows that the overall performance of the simulation increases almost linearly. As with the results in Figure 4.29, the speedup is limited for small numbers of photons by the generation of the initial blocks of pseudo-random numbers. This fact is illustrated in the timelines shown in Figure 4.22. Again, as in Figure 4.27, this initial cost is amortized as multiple blocks are used for larger numbers of photons.

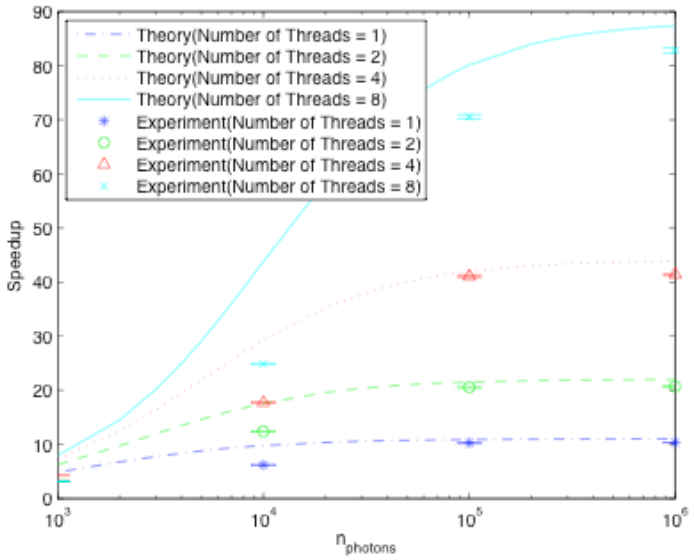

Figure 4.29 The measured scaled speedup for GPU-accelerated version of the RHT simulation using 1, 2, 4 and 8 threads on a single node (i.e., with one GPU). The photon numbers are increased from 10^3 to 10^6 in order to vary the workload. The work group size is 96, the number of work groups is 14, the number of iterations is 100 and the number of kernel cycles is 500.

The relative cost of the RHT portion of the simulation can be varied by increasing the computed accuracy of the energy contributed to an element during the transmission of a photon (as depicted in Figure 4.26(c)). This increased accuracy requires the use of more pseudo-random numbers for the Monte Carlo integration involving the absorptivity when traversing the element. The effect of this increased accuracy on the speedup is shown in Figure 4.30. The speedups for three different numbers of samples per element (80, 160, and 240) are shown in this figure.

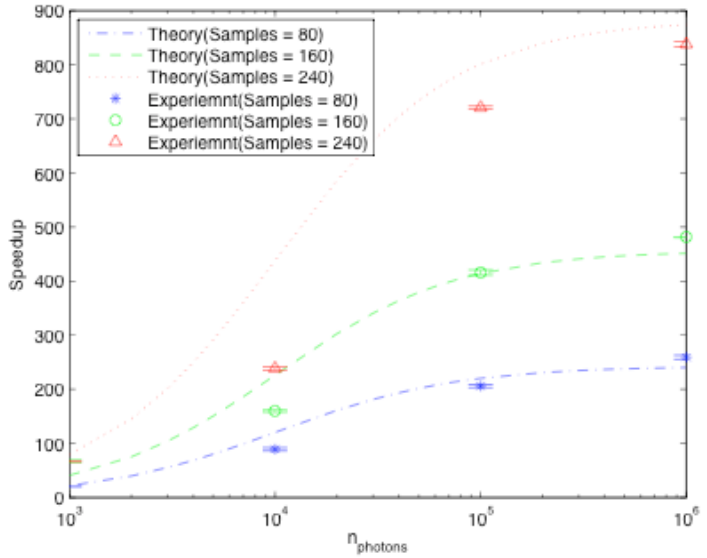

Figure 4.30 The speedup for the GPU-accelerated algorithm run on a single CPU for different transmission sampling strategies (as described in the text). The number of samples per element is respectively set to 80, 160 and 240. The number of photons used is increased from 10^2 to 10^6 in order to vary the workload. For these results the work group size is 96, the number of work groups is 14, the number of iterations is 100 and the number of kernel cycles is 500.

To be able to generate the theoretical curves shown in Figure 4.27-4.30, one needs to know how the RHT portion of the simulation, $T_{RHT}(B)$, scales with block size. In Figure 4.31 we show how this time varies with block size and with differing numbers of samples per element during the transmission calculation.

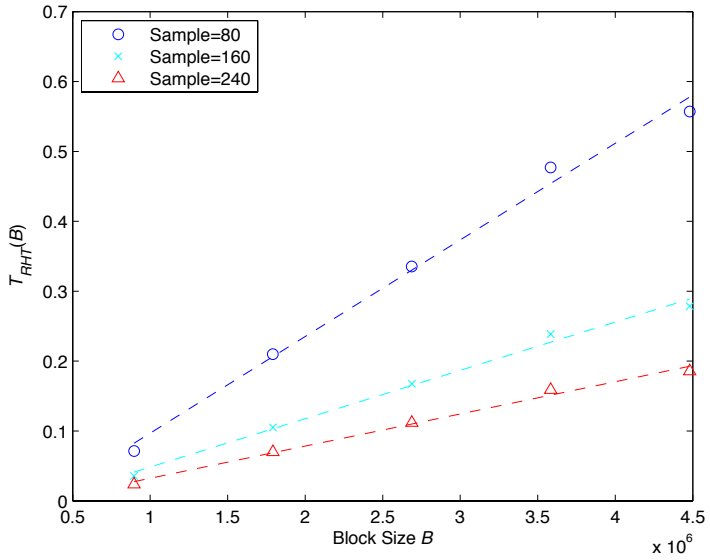

Figure 4.31 The change of $T_{RHT}(B)$ as a function of block size B. The block size B is increased from 896,000 to 4,480,000 for the data shown on this plot. Linear least squares fits to these data points are shown as the dashed lines in this figure. For sampling of 80, 160 and 240 points per element, the respective slopes from the least squares fit are 1.38×10^{-7}, 6.9×10^{-8} and 4.6×10^{-8}.

Finally, the RHT simulation was run on a complete hybrid architecture including distributed memory (using 10 nodes), using multiple threads on each node (8 threads per node), and using the GPU-accelerated pseudo-random number generator (using one GPU per node). The scaled speedup results obtained on this hybrid architecture are shown in Figure 4.32. Clearly, significant scaled speedups can be obtained for a modestly sized hybrid architecture using this approach.

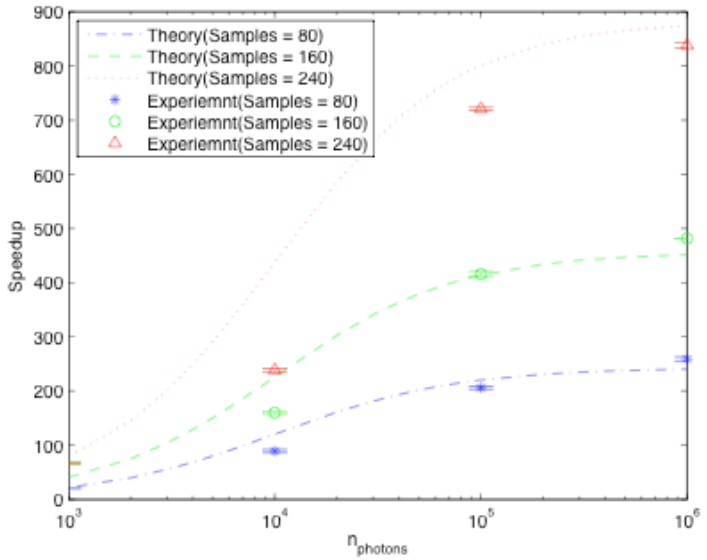

Figure 4.32 The scaled speedup plots for the RHT simulation using the GPU-accelerated pseudo-random number generator on a hybrid computing architecture. The number of photons used in the Monte Carlo simulation is increased from 10^3 to 10^6 in order to vary the workload. 10 compute nodes are used with 8 threads per node for the simulation. For the GPU the work group size is 96, the number of work groups is 14, the number of iterations is 100 and the number of kernel cycles is 500.

4.5.3.4 Statistical Analysis of the Experimental Results

To ensure that the simulation results obtained on this parallel system are statistically independent when using multiple nodes, we computed the mean of the temperature at one point in the interior of the domain for the multiple threads within each node. We then compute the standard deviation of these means (from the 10 nodes). This standard deviation is plotted in Figure 4.33 as a function of the number of photons used in the simulation. The slope of this graph is -0.5, which is what one would expect (from the

central limit theorem) and is consistent with statistically independent simulation results from different nodes. Note that this result does not prove statistical independence, but it does show that the data is not statistical dependent.

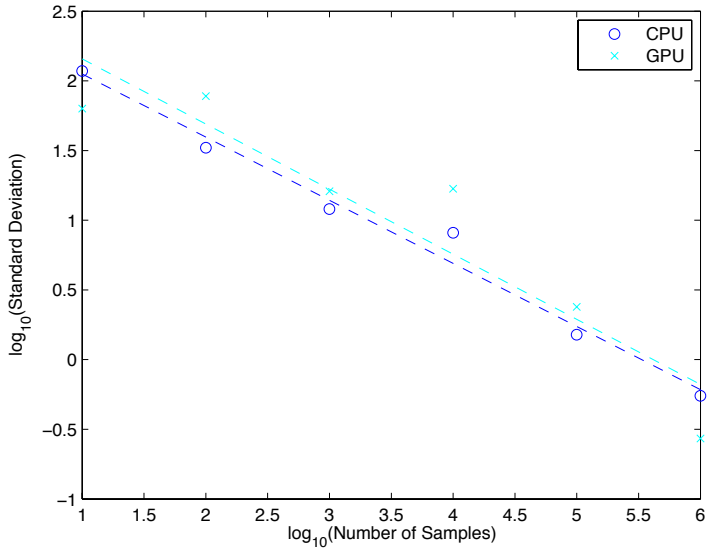

Figure 4.33 The standard deviation for the mean temperature obtained on different compute nodes as a function of the number of samples. Note that the slope of the best-fit line in this graph is -0.5, which is consistent with the Monte Carlo simulation data on different nodes being statistically independent.

4.5.4 Toward New Algorithms for Biological Systems Applications

In Chapter 3 we considered the discretization of the physical volume, where we referred to the discretized sub-volumes as elements and the collection of elements that make up the physical volume as the computational mesh. For these biological simulations, the movement of cells in a tissue is modeled to the movement of agents between neighboring

elements in this computational mesh. A simplified three-dimensional illustration of the elements and agents is displayed in the top image in Figure 4.34. In the top image of Figure 4.34, an element is indicated as E_k, the internal work of interaction and movement of agents is indicated as W_i^k and the summation of internal work of each agent is S_k. In the bottom image of Figure 4.34, we show a cropped, two-dimensional cross-section from a PathSim2 simulation. This image shows a rendering of cells (agents) and elements (indicated by the size of the colored squares).

Many of the computationally intensive element-based computations can be allocated to multiple threads and be distributed to multiple CPU cores. Certain other parts of the calculation, an element is indicated as E_k, the internal work of interaction and movement of agents is indicated as W_i^k and the summation of internal work of each agent is S_k. For example randomly generated agents, can be accomplished on the GPU. This allocation of tasks to a multi-core system with an attached GPU is illustrated in Figure 4.34.

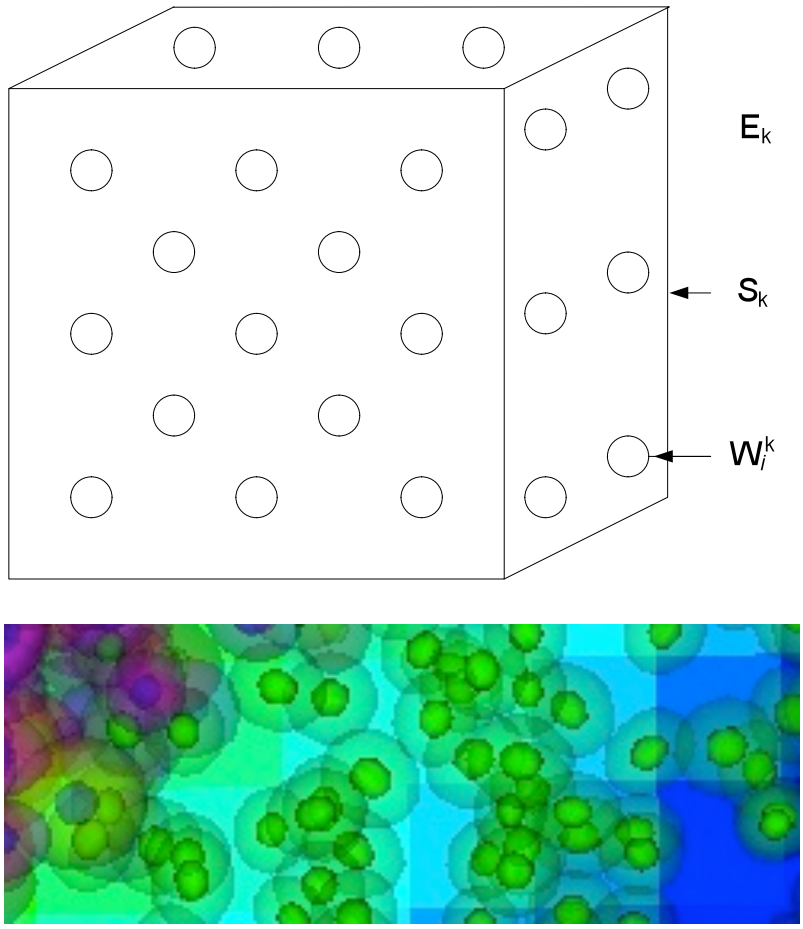

Figure 4.34 (Above) A simplified three-dimensional model of agents with elements, E_k : Element, W_i^k : Internal work of interaction and movement of agents, S_k : Summation of internal work of each agent; (Below) A close up of a two-dimensional cross-section from a PathSim2 simulation showing cells (agents) and elements (indicated by the colored squares) [32].

In Figure 4.35, the element sets assigned to the two cores are indicated as E_k and $E_{k'}$. The sets of agents within the element sets are denoted by S_k and $S_{k'}$. For the agents in these sets, the updating of the individual states involves the solutions of ODEs represented by the work W_0^k, W_1^k, …, W_n^k and $W_0^{k'}$, $W_1^{k'}$, …, $W_m^{k'}$. These work sets must be coordinated through the shared memory. Then they are prepared in parallel on the multiple stream

processors on the GPU. By using this scheme, PathSim2 simulation can achieve better performance.

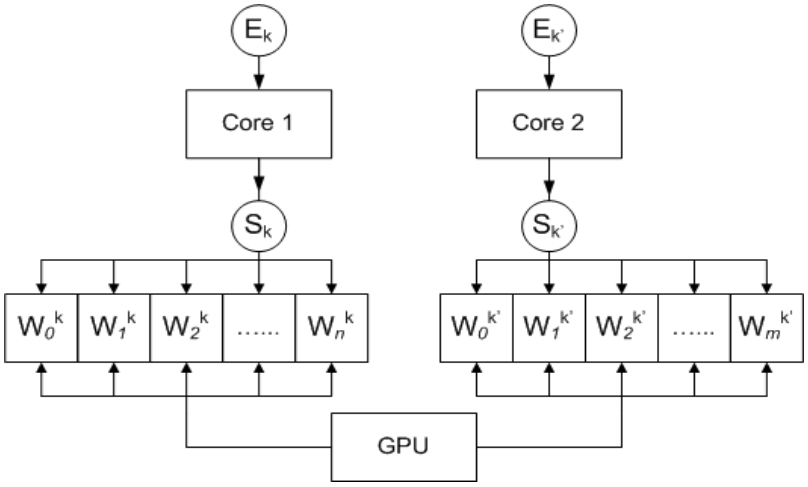

Figure 4.35 The assignment of element workload to multiple cores and the GPU [32].

5 Conclusions

This dissertation presents the characteristics of the complex architectural picture of hybrid computing. The advantage of having a clear characterization is that it can present to the applications programmer a model for how to use this architecture in the most effective manner for their particular application. This characterization is developed through the detailed analysis and implementation of several representative scientific computing applications. These applications include simulations of biological systems at the cellular level and a general-purpose software framework for Monte Carlo algorithms. The ultimate goal of the analysis of these software implementations is to develop an over-arching approach to "hybrid computing," that is a software approach that takes the best advantage of the various architectural resources available to the applications programmer. We now summarize the dissertation's contributions and identify future work.

5.1 Summary of Contributions

Implement and analyze a multiple GC simulation on a distributed memory architecture [completed and presented in Chapter 2].

Summary: We have developed a theoretical model for the performance of simulating multiple GCs. Message-passing implementation of this approach is developed using PathSim2. To verify this model, experimental results were obtained by measuring the running time of our implementation on System X. The scaled efficiencies computed from

these results agree with our theoretical analysis. A significant advantage of our approach is that it introduces a longer, global time-step for updating the part of the simulation that requires inter-processor communication and synchronization. This longer time-step significantly improves the overall performance of parallel simulation without adversely affecting the accuracy of the computed results.

Develop an analysis of a multi-threading approach for an element-based processing scheme for Pathsim2 on a multi-core architecture [completed and presented in Chapter 3]. Summary: We compared two standard APIs for multi-threading, Pthreads and OpenMP, using the PathSim2. Although a pthread implementation has advantages in terms of detailed control (e.g., setting priorities), its measured performance was not as good as an OpenMP implementation. This difference is due to different system overheads and can be characterized as a difference in thread startup times. For the SGI ALTIX 3700 that our tests were run on, the OpenMP startup time was measured to be 2/3 that of the pthread startup time. An additional advantage of the OpenMP implementation is that its coding is slightly simpler. When the two APIs were used with the PathSim2 simulation code, similar relative performances were measured, and good speedups were achieved for a representative biological simulation.

Implement and analyze a GPU accelerated Monte Carlo software framework suitable for hybrid architectures to solve π estimation and Radiative Heat Transfer (RHT) simulation. [completed and presented in Chapter 4].

Summary: We have introduced and implemented a theoretical model of a multi-threaded Monte Carlo application framework using GPU acceleration on a hybrid computer architecture. In this framework the GPU acceleration is used for generating large blocks of pseudo-random numbers asynchronously. The target application is π value calculation and a baseline Radiative Heat Transfer (RHT) simulation. Experimental results are obtained by measuring the running time of the simulation and these running times are well explained by the theoretical analysis. We presented experimental results that confirm our analysis about the scaled speedup of this approach. Overall, this approach can be very effective and achieve nearly a 1,000 times speedup on a modestly sized hybrid machine.

Our approach demonstrates an efficient way of mixing multi-threading with GPU acceleration that can be used in π value calculation or in real world applications such as RHT. We observe that generating as much data as possible from the GPU at a time improves the overall simulation time relative to a CPU-based scheme. However, the time required transferring data between the CPU and GPU memories and hardware setup times ultimately limit the efficiencies of these algorithms. In case of transferring small date from the CPU to GPU and big data from the GPU to CPU, we observe that transferring time from the CPU to GPU can significantly improve the simulation efficiency. These limits need to be considered when considering the overall benefits possible for a GPU-accelerated software application framework. The overhead of memory copies to and from the CPU and GPU must be amortized through the use of large data block transfers and significant data reuse on the GPU.

5.2 Future Work

A direction for future research is to extend an overall characterization of hybrid computing architectures and uses the properties of applications effectively. In Chapter 1 of this dissertation, in Table 1.1, the basic architectural parameters of the computing layers of a hybrid computing system are listed. The goal of this section of my dissertation would be to develop an overall characterization of what properties a scientific computing application must have in order to take full advantage of hybrid, parallel computing. Given such a characterization, I think that various applications can be analyzed and developed for a more comprehensive picture of the potential advantages and challenges of hybrid, parallel computing.

Bibliography

[1] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing,* vol. 22, pp. 789-828, 1996.

[2] Surendra Byna, William Gropp, Xian-He Sun, and Rajeev Thakur, "Improving the performance of MPI derived datatypes by optimizing memory-access cost," in *Proceedings. IEEE International Conference on Cluster Computing*, pp. 412-19, 2003.

[3] Michael Ott, Tobias Klug, Josef Weidendorfer, and Carsten Trinitis, "Aautopin - Automated optimization of thread-to-core pinning on multicore systems," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6590, pp. 219-235, 2011.

[4] Josh Simons, *The OpenMP® API specification for parallel programming*. Available: http://openmp.org/wp/, Accessed: July.1.2011.

[5] Blaise Barney, *POSIX Threads Programming*. Available: https://computing.llnl.gov/tutorials/pthreads/, Accessed: July.1.2011.

[6] W. B. Langdon and Wolfgang Banzhaf, "A SIMD interpreter for genetic programming on GPU graphics cards," in *Genetic Programming. 11th European Conference,* pp. 73-85, 2008.

[7] Everett H. Phillips, Roger L. Davis, and John D. Owens, "Unsteady turbulent simulations on a cluster of graphics processors," in *40th AIAA Fluid Dynamics Conference,* 2010.

[8] John E. Stone, David Gohara, and Guochun Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering,* vol. 12, pp. 66-73, 2010.

[9] Jen-Hsung Huang, *CUDA*. Available: http://developer.nvidia.com/category/zone/cuda-zone, Access: March.1.2011

[10] Hans Meuer, *Infiniband*. Available: http://www.top500.org/ 2007_overview_recent_ supercomputers/infiniband, Accessed: May.1.2007.

[11] Brice Goglin and Nathalie Furmento, "Finding a tradeoff between host interrupt load and MPI latency over Ethernet," in *2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER),* 2009.

[12] Matthias S. Muller, "A shared memory benchmark in OpenMP," in *High Performance Computing. 4th International Symposium, ISHPC 2002. Proceedings,* pp. 380-9, 2002.

[13] *HT Connectors and Cable*s, Available: http://www.hypertransport.org/default.cfm?page=HTConnectorsAndCables, Accessed: March.1.2009.

[14] Jen-Hsung Huang, *NVIDIA OpenCL Best Practices Guide*. Available: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, Accessed: June.1.2009.

[15] Jim Brewer, *PCI EXPRESS TECHNOLOGY*. Available: http://i.dell.com/sites/ content/ business/solutions/whitepapers/en/Documents/wp-2004_pciexpress.pdf, Accessed: May.1.2004.

[16] *PathSim2*. Available: http://pathsim2.ece.vt.edu/, Accessed: July.1.2008.

[17] Marvin H. Kalos and Paula A. Whitlock, *Monte Carlo methods*. New York: J. Wiley & Sons, 1986.

[18] *PathSim*. Available: http://pathsim.vbi.vt.edu/, Accessed: July.1.2007.

[19] Nicolas F. Polys, Doug A. Bowman, Chris North, Reinhard Laubenbacher and Karen Duca, "PathSim visualizer: An information-rich virtual environment framework for systems biology," in *Proceedings - 9th International Conference on 3D Web Technology*, pp. 7-14, 2004.

[20] Joo Hong Lee, Mark T. Jones, and Paul E. Plassmann, "An efficient shared memory programming model for biological systesm simulation," in *2010 International conference on Parallel and Distributed Processing Techniques and Applications*, pp. 315-319, 2010.

[21] K. Stuben, "Europort-D: commercial benefits of using parallel technology," in *Proceedings of ParCo 97 Parallel Computing 97,* pp. 61-78, 1998.

[22] Frank Baetke, "Trends in high performance computing for industry and research," in *Parallel and Distributed Processing and Applications. 4th International Symposium, ISPA 2006. Proceedings,* pp.4, 2006.

[23] Jack Dongarra, "Trends in high performance computing: a historical overview and examination of future developments," *IEEE Circuits and Devices Magazine,* vol. 22, pp. 22-7, 2006.

[24] Vaidy Sunderam, "Current trends in high performance parallel and distributed computing," in *International Parallel and Distributed Processing Symposium (IPDPS 2003), pp. 22-26, 2003*

[25] Joo Hong Lee, Mark T. Jones, and Paul E. Plassmann, "A scalable distributed memory programming model for large-scale biological systems simulation," in *2010 International conference on Scientific Computing*, pp. 251-256.

[26] M. Shapiro, K. A. Duca, K. Lee, E. Delgado-Eckert, J. Hawkins, A.S. Jarrah, R. Laubenbacher, N. F. Polyc, V. Hadinoto, and D. A. Thorley-Lawson, "A virtual look at Epstein-Barr virus infection: Simulation mechanism," *Journal of Theoretical Biology,* vol. 252, pp. 633-648, 2008.

[27] Filippo Castiglione, Karen Duca, Abdul Jarrah, Reinhard Laubenbacher, Donna Hochberg, and David A. Thorley-Lawson, "Simulating Epstein-Barr virus infection with C-ImmSim," *Bioinformatics,* vol. 23, pp. 1371-1377, 2007.

[28] David A. Thorley-Lawson, Karen Duca, Michael Shapiro, "Epstein-Barr virus: a paradigm for persistent infection - for real and in virtual reality," *Trends in Immunology,* vol. 29, pp. 195-201, 2008.

[29] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, "Scalable Parallel Programming with CUDA," *Queue,* vol. 6, pp. 40-53, 2008.

[30] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2004,* pp. 777-786, 2004.

[31] Gabriele Jost, Haoquang Jin, Dieter an Mey, and Ferhat F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigm on an SMP Cluster," 2003.

[32] Joo Hong Lee, Mark T. Jones, and Paul E. Plassmann, "A Hybrid software framework for the GPU acceleration of multi-threaded Monte Carlo Applications," International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol. 1, pp. 70-76, 2011

[33] Orion S. Lawlor, "Message passing for GPGPU clusters: cudaMPI," in *2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER), 2009.*

[34] Ricahrd Stallman. *GNU Operating System*. Available: http://www.gnu.org/, Accessed : Aug.1.2011.

[35] Richard Simard, *TestU01*. Available:http://www.iro.umontreal.ca/~simardr/testu01/tu01.html, Accessed: March.1.2011.

[36] R. Goncalves dos Santos, M. Lecanu, S. Ducruix, O. Gicquel, E. Iacona, and D. Veynante, "Coupled large eddy simulations of turbulent combustion and radiative heat transfer," *Combustion and Flame,* vol. 152, pp. 387-400, 2008.

[37] Robert D. Boudreau, "A solution to the integral equations for radiative transfer of heat in the atmosphere," Ph.D. thesis, Texas A&M University, College Station, 1968.

[38] Brian G. Wiedner and Cengiz Camci, "Technique for the determination of local heat flux on steady state heat transfer surfaces with arbitrarily specified external and internal boundaries," 29th National Heat Transfer Conference, pp. 21-31, 1993.

[39] Michael F. Modest, Radiative heat transfer, second edition. Academic Press, 2003.

[40] Bruce J. Palmer, M. Kevin Drost, James R. Welty, "Monte Carlo simulation of radiation heat transfer in arrays of fixed discrete surfaces using cell-to-cell photon transport," 28th National Heat Transfer Conference and Exhibition, pp. 85-91, 1992.

[41] Ivana Veljkovic and Paul E. Plassmann, "Scalable Photon Monte Carlo Algorithms and Software for the Solution of Radiative Heat Transfer Problems," High Performance Computing and Communication (HPCC), vol. 3726, pp. 928-937, 2005.

[42] F. James, "RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Lüscher", *Computer Physics Communications*, vol. pp. 111–114, 1994.

[43] "GNU Operating System GSL(GNU Scientific Library) [online]. http://www.gnu.org/software/gsl/, Accessed:Aug.1.2011

[44] "Athena"[online] http://www.arc.vt.edu/arc/Athena/index.php, Accessed: Dec.1.2011

[45] "TESLA™ S2050 GPU Computing System"[online]. http://www.nvidia.com/docs/IO/105880/NV-DS-Tesla-S2050-Aug11.pdf, Accessed: Dec. 1. 2011.

[46] Jacques G. Amar, "The Monte Carlo method in science and engineering," *Computing in Science & Engineering,* vol. 8, pp. 9-19, 2006.

[47] Eugen Nisipeanu and Peter D. Jones, "Monte Carlo simulation of radiative heat transfer in coarse fibrous media," *Journal of Heat Transfer,* vol. 125, pp. 748-52, 2003.

[48] David Middleton, Piyush Mehrotra, and John Van Rosendale, "Expressing direct simulation Monte Carlo methods in High Performance Fortran," *SIAM Conference on Parallel Processing for Scientific Computing*, pp. 698-703, 1995.

[49] Vadim Demchik, "Pseudo-random number generators for Monte Carlo simulations on ATI Graphics Processing Units," *Computer Physics Communications,* vol. 182, pp. 692-705, 2011.

[50] Kenta Hongo, Ryo Maezono, and Kenichi Miura, "Random number generators tested on quantum monte carlo simulations," *Journal of Computational Chemistry,* vol. 31, pp. 2186-2194, 2010.

[51] Umesh V. Vazirani and Vijay V. Vazirani, "Efficient and secure pseudo-random number generation," 25th Annual Symposium on Foundations of Computer Science, pp. 458-63, 1984

[52] S. Hong, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *Proceedings - International Symposium on Computer Architecture,* pp. 152-163, 2009.

[53] G. T. Byrd, "Producer-consumer communication in distributed shared memory multiprocessors." *Proceedings of the IEEE* 87(3): pp. 456-466, 1999.

[54] K. Jeffay, "The real-time producer/consumer paradigm: a paradigm for the construction of efficient, predictable real-time systems," Proceedings of 8th SIGAPP Symposium on Applied Computing, pp. 796-804, 1993.

[55] Y. Smaragdakis, "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs." ACM Transactions on Software Engineering and Methodology 11(2): pp. 215-255, 2002.

[56] "CUDA Toolkit 4.0 Readiness for CUDA Applications" [online] http://cora.gridlab.univie.ac.at/docs/CUDA/CUDA_4.0_Readiness_Tech_Brief.pdf, Accessed: Aug.1.2012

[57] Harty, K. and D. R. Cheriton, "Application-controlled physical memory using external page-cache management." SIGPLAN Not. 27(9): pp. 187-197, 1992.

[58] I. A. Ibragimov, "A note on the central limit theorem for dependent random variables." Theory of probability and its applications 20(1): pp 135-141, 1975.

[59] J. Nickolls and W. J. Dally, "The GPU Computing Era," *Micro, IEEE,* vol. 30, pp. 56-69, 2010.