A Hybrid Software Framework for the GPU Acceleration of Multi-Threaded Monte Carlo Applications

Joo Hong Lee, Mark T. Jones and Paul E. Plassmann
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia 24061

Abstract – *Monte Carlo simulations are extensively used in wide of application areas. Although the basic framework of these is simple, they can be extremely computationally intensive. In this paper we present a software framework partitions a generic Monte Carlo simulation 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 in most any Monte Carlo application without requiring application-specific programming of the GPU. We present an analysis of the performance of this software framework. Finally, we compare this analysis to experimental results obtained from our implementation of this software framework.*

Keywords: Parallel Monte Carlo algorithms, GPU acceleration, Hybrid algorithms, Scientific computing

1. Introduction

Recently there has been a tremendous amount of interest in using Graphics Processing Units (GPUs) to perform computationally intensive tasks traditionally performed by CPUs. This interest is motivated by the raw peak performance numbers available on current GPUs—these peak performance numbers can 1,000 times that of the associated CPU. However, there are a number of problems in achieving a high percentage of this peak performance. These problems include the parallelization of complex algorithms into large numbers of lightweight threads, the overhead of copying data between CPU and GPU memories, and the difficulties of developing GPU programs.

GPU programming has been done with the CUDA API of NVIDIA [1] or Brook+ of AMD [2]. However, each programming API is compatible with only its own hardware. Recently, the portable API OpenCL [3] has been developed for GPUs and as a result, the prospect for an over-arching portable approach for a hybrid-computing model has begun to get more attention [4]. The idea behind the portable, hybrid approach is the use of multiple threads to exploit the multiple cores on GPU.

An example of where this Monte Carlo framework can be used is PathSim2, a software environment developed to simulate biological systems at the cellular level focused on Germinal Center (GCs) [5]. The goal of these simulations is to model an adaptive immune response of the human tonsil [6-7]. As this biological system represents the motion, interaction and aging of large number of agents, it requires significant computing power. In particular, the efficient usage of computing power of GPU must be coupled with the whole simulation model to increase the performance of the system simulation. The key to make the simulation faster depends on how to transfer the CPU work to the GPU side in an efficient manner. As PathSim2 requires a parallelization strategy that can speed up the simulation using GPUs, we use an OpenCL-based solver.

Programmers in Biological System Simulation (BSS) area have started to model their program working on parallel architectures since parallel architecture have appeared and shown good performance [8]. The current trend is combining shared- and distributed-memory programming models together [9-10]. The parallel-programming techniques have evolved to take advantage of the emergence of multi-core, distributed memory computer architectures with GPUs [11]. The parallelization approach developing for PathSim2 also follows parallelization strategies in current BSS trend.

The remainder of this paper is structured as follows. In section 2 we present an overview of the simulation model and our approach to parallelization. A theoretical analysis of the performance of this proposed approach is described in section 3. In section 4 we compare experimental results of the performance of the simulation framework with the theoretical model. We present our conclusion in section 5.

2. Simulation Model Overview

PathSim2 is a software framework that simulates the motion and interaction of biological agents within a discretized three-dimensional spatial region. 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 three-dimensional illustration of the elements and agents is displayed in the top image in Figure 1. In the top image of Figure 1, 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 1, we show a cropped, two-dimensional cross-section from a PathSim2 simulation [12]. This image shows a rendering of cells (agents) and elements (indicated by the size of the colored squares).

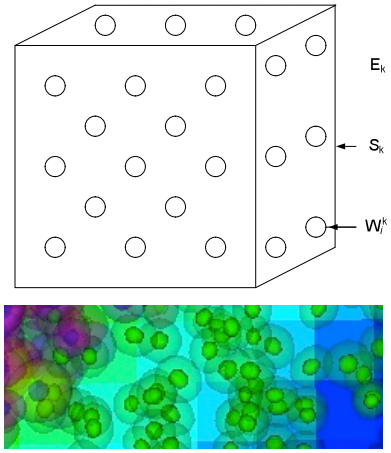

Figure 1. (Above) A simplified three-dimensional model of agents with elements, E_k : Element, W_i^k : Internal work of interaction and movement of agents, S_k : Summation of internal work of each agent; (Below) A close up of a two-dimensional cross-section from a PathSim2 simulation showing cells (agents) and elements (indicated by the colored squares [12].

Since these element-based calculations are independent, they can be executed in separate threads. Each separate thread is distributed to each core on CPU then the computational part is calculated on main memory. However, this computational part also requires data and it can be transferred from GPU. The required data is generated on GPU then copied back to main memory for computing. Above process is described in the memory model of CPU and GPU in Figure 2.

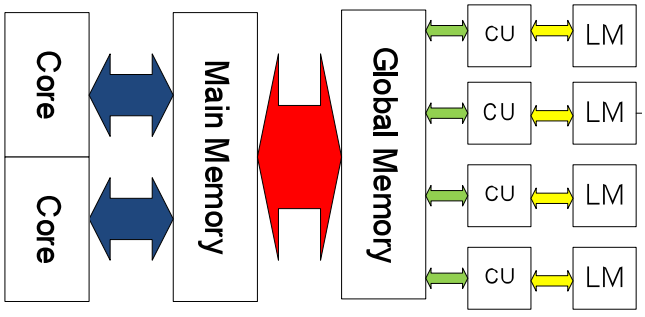

Figure 2. A Memory Model showing the CPU and GPU Architectures. CU: Compute Unit, LM: Local Memory.

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, for example randomly generated agents, can be accomplished on the GPU. This allocation tasks to a multi-core system with an attached GPU is illustrated in Figure 3.

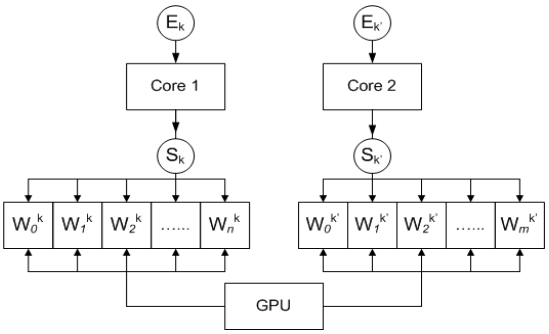

Figure 3. The assignment of element workloads to multiple cores and the GPU.

In Figure 3, 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 (this could, for example, involve the solutions of ODEs) is 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 shared memory. Then the individual work tasks are accomplished in parallel on the multiple stream processors on the GPU.

To do this, a "managing" thread can generate a new random number block as needed. Monte Carlo threads access memory blocks that are ready to be used, and when a block is used up, the managing thread swaps the memory block to another full memory blocks. Empty memory blocks are filled again on the GPU by calls from the managing thread. This overall process is illustrated in Figure 4.

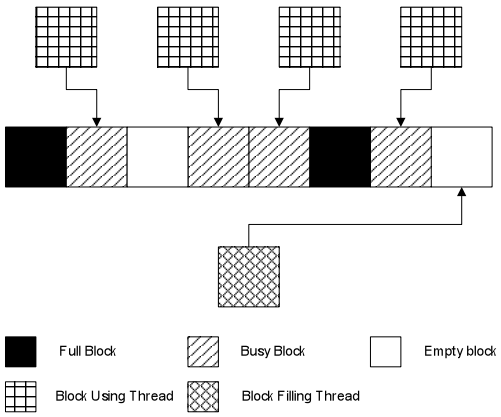

Figure 4. An illustration of how the random number blocks are managed between the GPU managing thread and the Monte Carlo application threads. The thread that manages GPU kernel fills one memory block at a time, while multiple Monte Carlo threads have access to other memory blocks that have been previously filled by the GPU managing thread.

3. Theoretical Analysis

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.

3.1. GPU 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 5. In Figure 6 we give a high-level view of the OpenCL kernel code that is executed by each thread on the GPU.

```
// 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[i*offset], PRNs_Size, GPU_PRNs);
}

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

Figure 5. 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.

From the OpenCL pseudo-code shown in Figure 6, one can see that the required computation time is comprised of the time required to complete three types of tasks. 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 6. A high-level view of the kernel code run on the GPU. The arguments passed to the kernel include the number of "kernel cycles" and pointers to the pseudo-random number generator state tables (input and output) and 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.

3.2. Speedup

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 [13], a linear model can accurately represent the time required to transfer data between these memories as a function of the amount of data transferred. In Figure 7, 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.

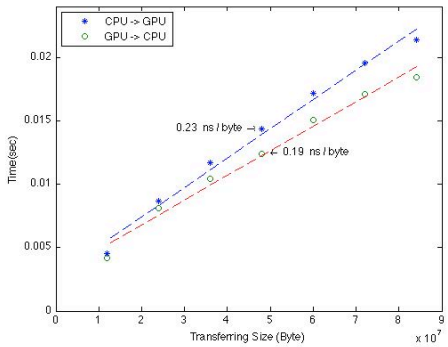

Figure 7. Data transferring time between the CPU and the GPU as a function of the number of bytes transferred. Note the different transfer rates to and from the GPU.

We use the following expression to model the time, $T_{CPU \rightarrow GPU}(m)$, to write m bytes of data from the CPU to the GPU as

$$T_{CPU \rightarrow GPU}(m) = t_{CG}\, m + t_s \,, \quad (1)$$

where t_s is a "start up" time for the write, and t_{CG} is the incremental time required to write each addition byte of data. Based on a least squares fit to experimental results from a simple program that writes data between the CPU and the GPU (as shown in Figure 7), values for these constants were obtained and are presented in Table 1.

Likewise, we can use a linear model for the time, $T_{GPU \rightarrow CPU}(m)$, required to read m bytes of data from the GPU to the CPU. Thus, we use the expression

$$T_{GPU \rightarrow CPU}(m) = t_{GC}\, m + t_s \,, \quad (2)$$

where t_{GC} is the incremental time required to read each addition byte of data. Note that t_s for both the write and read are nearly equal, so we model them as the same. The incremental times for the read and write are different enough that we use two different constants as shown in Table 1.

Table 1. Constants for t_s, t_{CG} *and* t_{GC} as computed for the machine architecture used for the experimental tests.

Constant	Time
t_s	*2.9ms/copy*
t_{CG}	0.23*ns/byte*
t_{GC}	0.19*ns/byte*

On the first line of the program outline given in Figure 5, 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. The number of GPU threads is the number of work items, n_{wi}. The number of work group items is the product of the number of work groups, n_{wg}, and the work group size, n_{wgs}. Thus, the time required to write the state tables to the GPU is given by the expression

$$T_{seedUp} = n_{wi}\, n_{tab}\, t_{CG} + t_s \,. \quad (3)$$

The time required to read the state tables back from the GPU (the last line of the program segment in Figure 5) is given by the expression

$$T_{seedDown} = n_{wi}\, n_{tab}\, t_{GC} + t_s \,. \quad (4)$$

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 CPU. Let n_k be the number of kernel cycles (the number of iterations in the loop in Figure 6), then the time required to read these numbers back to the CPU memory would be given by the expression

$$T_{numDown} = n_k\, n_{wi}\, t_{GC} + t_s \,. \quad (5)$$

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 Radeon HD 5750 used for these experiments, 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.

The effect of the GPU architecture is illustrated in Figure 8. In this figure the number of kernel cycles is fixed at 10,000; we then measure the time it takes for the kernel to execute (the call to *queue.enqueueNDRangeKernel* in Figure 5). The measured times are shown as the green '+' symbols in this graph. As we vary the size of the work group, we can observe the effect of having a limited number of stream processing units within a compute unit on which to schedule threads to execute. As the work group size increases beyond multiples of 80 (e.g., 80, 160, and 240) we observe discrete jumps in the measured times. We denote the number of stream processing units per compute unit as p_{wgs}. By examining experimental results (similar to those in Figure 7), we empirically determined that the execution time depends on two terms, a "kernel start up time" T_s and a "kernel execute time" which we denote by T_e. The first term can be modeled as

$$T_s = a\, n_k + b \,, \quad (6)$$

where a is an incremental rate, *200ns/kernel-cyle*, and b is fixed setup time, 0.9*ms*. The second term can be modeled as

$$T_e = t_c^{GPU} \lceil n_{wgs} / p_{wgs} \rceil, \quad (7)$$

where t_c^{GPU} was measured to be 160*ns*. Using this model for the execution time, we obtain the black '*' points shown in Figure 8. The total time for the kernel to execute can then be modeled as

$$T_k = T_e + T_s \,. \quad (8)$$

To use the memory available on the GPU efficiently, the program given in Figure 5 iteratively generates random number blocks and reads these values back to the CPU as they are generated. The random number state tables only have to be copied to and read from the GPU outside of this iteration loop. We denote the number of iterations for this loop as n_i. We can then combine all the terms in our model to obtain an overall model for the time for the program given in Figure 5 to execute as

$$T_{GPU} = T_{seedUp} + T_{seedDown} + n_i\,(T_k + T_{numDown}) \,. \quad (9)$$

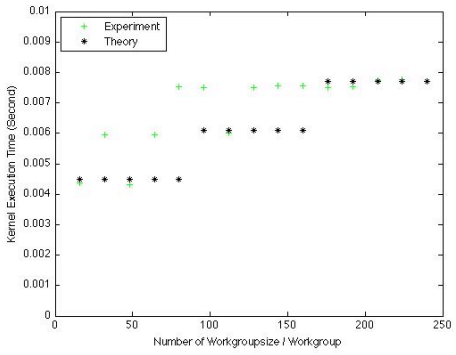

Figure 8. The time measured for the kernel to execute as a function of the work group size. For this data we fixed the number of work groups to one. The experimentally measured data is shown as the green '+' points, the modeled times (as explained in the text) is shown as black '*' points in this graph.

We can compare the time required to generate the pseudo-random numbers on the GPU to the time to generate these numbers using the CPU. If we denote the pseudo-random number generating rate on the CPU by t_c^{CPU}, the total required time on the CPU would be

$$T_{CPU} = n_k \, n_{wi} \, n_i \, t_c^{CPU} . \quad (10)$$

We can compute a speedup for using the GPU relative to using the CPU by taking the ratio of these times as

$$S = T_{CPU} / T_{GPU} . \quad (11)$$

In the subsequent section of our paper we present experimental results and compare these results to the above model. One quick observation from the model is the relatively large start up cost for reading and writing data to the GPU. It is clear that in order to amortize this start up cost, a relatively large block of pseudo-random numbers must be generated at each iteration in order to have any chance of obtaining a good speed up.

4. Experimental Results

4.1. GPU Speedup Measurement

In measuring the performance of our model, we generate a fixed number of 245,760,000 pseudo-random numbers using work group sizes of 80, 160, and 240. For the GPU we are using (a Radeon HD 5750), one compute unit consists of 80 processing elements. Thus, we increase the work group size in increments proportional to this number to allow threads to be scheduled efficiently on the GPU. The time to generate random numbers using the GPU, as compared to the CPU, is presented as a speed up in Figure 9. The experimentally measured values for t_c^{CPU} is 65*ns/number* and for t_c^{GPU} is 160*ns/number*. These experimental values are used in the theoretical model presented in the previous section, and are shown in Figure 9. We also include the model where we limit the number of compute units to 9 (and as a result the number of stream processing units to 720) and this model is also shown as the solid curves in the figure. Note that these results agree well with the theoretical model that includes the limited number of processing units available on the GPU. It is interesting to note the weak dependence of the speedup with respect to the work group size (the difference between the three curves shown in the graph). This weak dependence is due to the kernel "set up time" T_s.

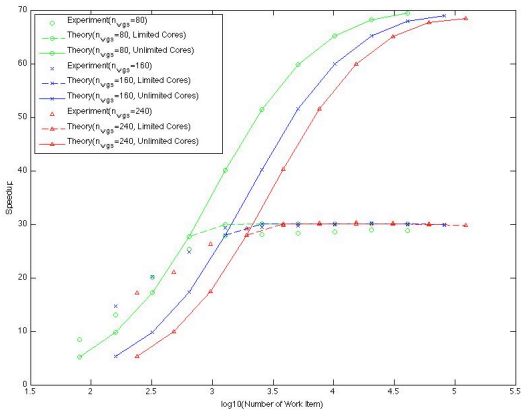

Figure 9. Speedup plots comparing the GPU execution time to the CPU execution time. Three different work group sizes (80, 160 and 240) are used. The number of work groups is increased from 1 to 512 in order to vary the number of work items.

To observe the performance improvement for using the GPU to generate the pseudo-random numbers in a simple Monte Carlo application, we consider a numerical integration scheme to estimate the value of π. The overall software framework is illustrated in Figure 4. 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 [14]. When using the GPU, memory blocks are filled with the method described in Figures 5 and 6. The simulation times for the CPU and the GPU are compared, and the resulting speedup is shown in Figure 10. 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.

4.2. Randomness Check

Before using generated random numbers, we need to know that the random numbers are statistically independent. The standard way to check the randomness is as follows. First, we subdivide the samples into some

number of independent subsamples and obtain each sub-sample mean. Then, the standard deviation of for these subsample means should be decreasing as we increase the number of samples (a result of Central Limit Theorem). This approach can be used as a sanity check for the pseudo-random numbers generated on the independent threads on the GPU. In Figure 11, we present the measured standard deviation of pseudo-random numbers generated using the CPU and the GPU. As expected, the standard deviation decreases as the square root of the number of samples with increasing numbers of samples for both cases.

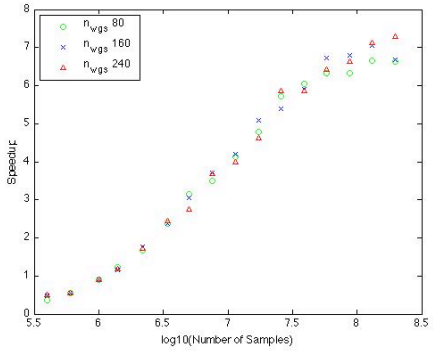

Figure 10. Speedup of a simple Monte Carlo simulation using the GPU acceleration scheme with work group sizes of 80, 160 and 240.

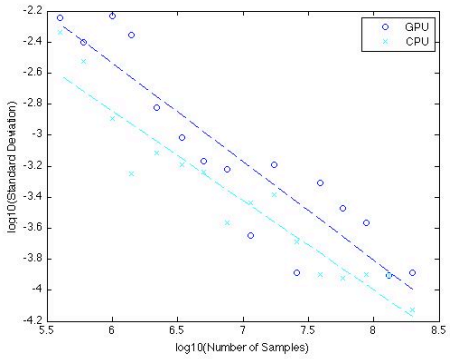

Figure 11. The standard deviation of the means computed for the Monte Carlo application as a function of the number of samples used to compute the means. As expected from the Central Limit Theorem, these standard deviations should decrease as the square root of the number of samples (the dashed lines in the figure).

As a more rigorous test, we also used the empirical tests of TestU01 to check the theoretical quality of our RANLUX pseudo-random numbers [15]. This package contains three sets of test batteries. The tests are SmallCrush, Crush, and BigCrush. These tests apply a variety of statistical tests to large sequences of random numbers. We tested the GPU implementation of RANLUX using these three tests.

One fine point in using these tests is that the GPU implementation of RANLUX has only 24 bits of resolution as it is computed in single precision. This means that when converted to a double precision value for the tests, the additional mantissa bits in the double have to be filled with statistically independent values for the test. Once this was done, then the random numbers passed the SmallCrush battery. In the case of the Crush and BigCrush battery, except only one battery, it passes all of the tests. The tests included in SmallCrush, Crush and BigCrush respectively include 15, 144, and 160 independent statistical tests.

4.3. Monte Carlo Simulation Results

To verify the adequacy of our theoretical model, we present the estimation of π using Monte Carlo method. The estimated value of π is presented with error and mean in Figure 12. The theoretical value is within the error range of experimental result. Also with more samples, the experimental result approximates to π. More importantly, the convergence depends on the square root of the number of samples as shown in Figure 11 (which agrees with what one expects from the Central Limit Theorem).

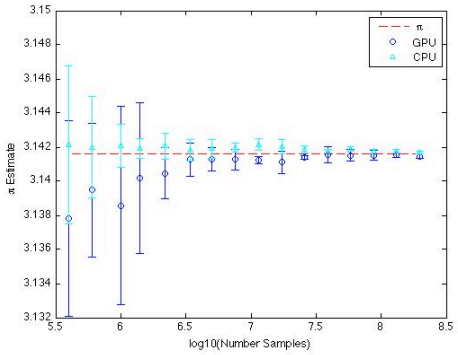

Figure 12. Convergence of the theoretical and experimental estimation of π by numerical integration with the Monte Carlo framework as a function of the number of samples used.

5. Conclusions

In this paper we have introduced a theoretical model of the efficiency of a multi-threaded Monte Carlo application framework using GPU acceleration. Experimental results are obtained by measuring the running time of the simulation framework and these running times are well explained by the theoretical analysis.

Our approach demonstrates an efficient way of mixing multi-threading with GPU acceleration. We observe that generating as much data as possible from the GPU at a time (through blocking) 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. These limits need to be considered when considering the overall benefits possible for a GPU-accelerated software application framework.

6. ACKNOWLEDGEMENTS

This work was supported by NSF grant CCF-0728901.

7. REFERENCES

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," ACM Queue, vol 6, pp. 40-53, 2008.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," ACM Transactions on Graphics, vol 23, pp 777-786, 2004.

[3] J. E. Stone, D. Gohara, and S. Guochun, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," Computing in Science & Engineering, vol 12, pp 66-73, 2010.

[4] G. Jost, J. Jin, D. Mey, and F. Hatay, "Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster," presented at European Workshop on OpenMP(EWOMP), Germany, 2003.

[5] "PathSim2"[online].Available:http://pathsim2.ece.vt.edu/. [Accessed: Feb. 1, 2010].

[6] "PathSim"[online].Available:http://pathsim.vbi.vt.edu/. [Accessed: Feb. 1, 2010].

[7] N. F. Polys, D. A. Bowman, C. North, R. Laubenbacher, and K. Duca, "PathSim visualizer: an Information-Rich Virtual Environment framework for systesm biology," International conference on 3D Web Technology, pp. 7-14, 2004.

[8] K. Stuben, "Europort-D: commercial benefits of using parallel technology," Parallel Computing: Fundamentals, Applications and New Directions, Advances in Parallel Computing, vol. 12, pp. 61-78, 1998.

[9] A. Moerschell and J. D. Owens, "Distributed Texture Memory in a Multi-GPU Environment" Computer Graphics Forum, vol. 27, pp. 130-151, 2008.

[10] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," presented at Conference on High Performance Computing Networking, Storage and Analysis, SC, Portland, OR, 2009.

[11] S. Shah, and E. Gabriel, "Image computing for digital pathology," presented at International Conference on Pattern Recognition (ICPR) , Tampa, FL, 2008.

[12] J. H. Lee, M. T. Jones, Paul E. Plassmann: A scalable distributed memory programming model for large-scale biological systems simulation. International Conference on Scientific Computing (CSC), pp. 251-256, 2010.

[13] O. S. Lawlor: Message passing for GPGPU clusters: cudaMPI. Cluster Computing and Workshops, pp. 1-8, 2009.

[14] "GSL – GNU Scientific Library" [online]. Available: http://www.gnu.org/software/ [Accessed:Feb.1,2011]

[15] "TestU01"[online].Available:http://www.iro.umontreal.ca/~simardr/testu01/tu01.html[Accessed: Mar.1, 2011]