Parallelization of Tau-Leap Coarse-Grained Monte Carlo Simulations on GPUs

Lifan Xu and Michela Taufer
Dept. of Computer & Inf. Sciences
University of Delaware
Email: {xulifan, taufer}@udel.edu

Stuart Collins and Dionisios G. Vlachos
Dept. of Chemical Engineering
University of Delaware
Email: vlachos@udel.edu

***Abstract*—The Coarse-Grained Monte Carlo (CGMC) method is a multi-scale stochastic mathematical and simulation framework for spatially distributed systems. CGMC simulations are important tools for studying phenomena such as catalysis, crystal growth, surface diffusion, phase transitions on single crystals, and cell membrane receptor dynamics. In parallel CGMC, the tau-leap method is used for parallel simulations that are executed on traditional CPU clusters in a master-slave setting. Unfortunately the communications between master and slaves negatively impact speedup and scalability.**

In this paper, we explore the potentials of GPUs for the tau-leap method and we present an extensive performance evaluation that leads to the most suitable degree of parallelism for this method under different simulation profiles. We show how the efficient parallelization of the tau-leap method for GPUs includes (1) the redefinition of its data structures, (2) the redesign of its algorithm, and (3) the selection of the most appropriate degree of parallelism (i.e., fine-grained or course-gained) on a single GPU or multiple GPUs. Exceptional performance improvements can thus be achieved for this method.

***Keywords*-Parallel programming; GPU programming; Monte Carlo methods; Data parallelism on GPUs.**

I. Introduction

The Coarse-Grained Monte Carlo (CGMC) method is a multi-scale stochastic mathematical and simulation framework for spatially distributed systems [4]. In the CGMC method, microscopic cells are grouped together into coarse cells, and this leads to tremendous acceleration in comparison to a microscopic MC simulation. In addition, the tau-leap CGMC method enables to take coarse time steps by executing multiple events at each time for all cells [3]. This is a key ingredient that can overcome the curse of one event at a time of the microscopic Monte Carlo (MC) method and allow time synchronization of multiple processors. CGMC simulations are important tools for studying phenomena such as catalysis, crystal growth, surface diffusion, phase transitions on single crystals, and cell membrane receptor dynamics.

Towards the goal of simulating larger, more complex phenomena more efficiently, it is desirable to accelerate MC simulations. Recent efforts have explored the potential of GPUs for density functional theory (DFT) and molecular dynamics (MD) simulations [12], [11]. To our knowledge, there is no previous work on parallelization of the CGMC method. Existing work targeting the microscopic MC method [9] and it may be limited to equilibrium rather than kinetic systems. In addition, the microscopic MC method is inefficient for large scale systems. In contrast, the CGMC method provides profound acceleration to enable simulations of large length and time scales. In this paper, we contribute to this effort by exploring the potential of GPU platforms for the tau-leap method supported by CUDA. To our knowledge, this is the first parallel implementation of this method. In particular, we present an extensive performance evaluation leading to the most suitable type and degree of parallelism for the tau-leap method under different application profiles. We show how the efficient parallelization of the tau-leap method for GPUs includes (1) the redefinition of the data structures, (2) the redesign of the algorithm, and (3) the selection of the most appropriate degree of parallelism (i.e., fine-grained or coarse-grained) on one single GPU or multiple GPUs.

The rest of this paper is organized as follows: Section II provides a short overview of GPU programming and the tau-leap method on CPUs; Section III describes different parallelization approaches for the tau-leap method on GPUs; Section IV presents the extensive performance evaluation of these approaches; and Section V concludes the paper and presents future work.

II. Background and Related Work

A. GPU Programming

GPUs are massively parallel multi-threaded devices capable of executing a large number of active threads concurrently. A GPU consists of multiple streaming multiprocessors, each of which contains multiple scalar processor cores. For example, NVIDIA's G80 GPU architecture contains 16 such multiprocessors, each of which contains 8 cores, for a total of 128 cores which can handle up to 12,288 active threads in parallel. Advanced GPU systems such as the Tesla S1070 include multiple GPUs that can be accessed and used simultaneously by an application using OpenMP pragma directives. In addition, the GPU has several types of memory, most notably the main device memory (global memory) and the on-chip memory shared between all cores of a single multiprocessor (shared memory) [8].

NVIDIA has introduced the CUDA language library, which facilitates the use of GPUs for general purpose programming by providing a minimal set of extensions to the C programming language. From the perspective of the CUDA programmer, the GPU is treated as a co-processor to the main CPU. A function that executes on the GPU, called a kernel, consists of multiple threads each executing the same code, but on different data, in a manner referred to as "single instruction, multiple data" (SIMD). Further, threads can be grouped into thread blocks, an abstraction that takes advantage of the fact that threads executing on the same multiprocessor can share data via the on-chip shared memory, allowing a limited degree of cooperation between threads in the same block [8].

B. Sequential and Parallel CGMC on CPUs

In microscopic MC simulations, a large number of microscopic sites, e.g., 10,000x10,000 sites, are occupied by different molecules with a maximum of one molecule per site. Neighboring sites are grouped into coarse-grained (CG) cells and a closure is applied at the stochastic level to resident molecules to describe their distribution in the cell (see Figure 1). The molecules are allowed to interact with, react with, and diffuse to nearby cells [5].

In the *sequential CGMC*, the probabilities (gamma) of every possible reaction (i.e., adsorption, desorption, and diffusion) are calculated sequentially and an event is executed based on probabilities [7]. This execution may change one molecule to another different molecule or diffuse from one cell to its neighbor cell, thus changing the population coverage (theta). The simulation updates theta, recalculate gamma according to theta and loop again (parameter synchronization). In the *parallel CGMC*, the tau-leap method can be used to assure scalability [3] and this approach extends the tau-leap method from well mixed systems (spatially homogeneous) [10], [6], [1], [2] to spatially distributed. The essence of the tau-leap method is that instead of executing one reaction in every microscopic time interval and changing the participating species by stoichiometric populations, the scientist selects a coarse time increment (tau), usually larger than the microscopic one. In this coarse time, each reaction is fired multiple times and the population is updated after each time step accordingly. The number of times each reaction is fired is selected randomly from a Poisson distribution. The parallel CGMC simulations based on the tau-leap method can be executed on high-end clusters of CPUs, one CPU for each cell in a master-slave setting. The master node defines and broadcasts tau. Slave nodes receive tau, compute gamma, select events, execute events, and send a package to the master with their new population changes. The master collects, changes, and broadcasts the new population and tau to all the slaves. Slaves receive changes, update event probabilities, calculate the local tau, and send it to the master that restarts the cycle. An analysis of the algorithm outlined how the communications between master and slaves negatively impact speedup and scalability on traditional clusters.

III. Parallelizations of the Tau-Leap Method on GPU

Since the GPU architecture is inherently different than traditional CPU architectures, code development and optimization for the GPU involve different approaches including the redesign of the data structures and the computation algorithms. Figure 2 shows the tau-leap method and those parts that can be executed on CPU and on GPU. This section presents different approaches to parallelize this method.

A. From Global to Shared Memory

Similarly as on CPUs, in a first parallelization of the tau-leap method on GPU, each thread is in charge of one cell and iteratively performs a given number of leaps, each with length tau (*GPU+global memory*). For each iteration, the thread first reads the probability of every event in its cell (gamma) and selects the events to execute based on a Poisson (or a binomial) distribution. Events include diffusion, adsorption, and desorption. When all the selected events are executed, the thread checks for any violation, i.e., whether there is a negative number of molecules or the number of molecules exceeds the number of microscopic sites in that cell. If there is no violation, the population coverage (theta) is updated; otherwise, the length of the leap, tau, is cut to half and the shorter leap including less events is re-executed. Once all the threads have updated their theta, each thread recalculates gamma for its events based on its own theta and that of its neighbors (synchronization).

This simple implementation has a major drawback, the frequent and expensive synchronization of the gamma and theta variables which cost is even higher if these variables are stored in global memory. Synchronizations through global memory cost 400-600 GPU cycles per access versus one cycle per read/write when the data is in shared memory. To recalculate gamma, each thread has to communicate with its neighbors. Because of the way the shared memory is organized, moving gamma into shared memory requires keeping all the threads in one block. The block maximum capacity of 512 threads becomes a limitation. On the other hand, we can move theta into shared memory, we can have threads selecting events based on gamma, and we can update theta locally in shared memory efficiently. After each leap, the threads have to write their own theta back to global memory and update their own gamma according to its neighbors' theta. This is the approach we used in our second parallelization of the tau-leap implementation for GPUs (*GPU+shared memory*).

B. From Cells to Macro-Cells

While using shared memory for the synchronization makes simulations definitely faster, when performing sim-

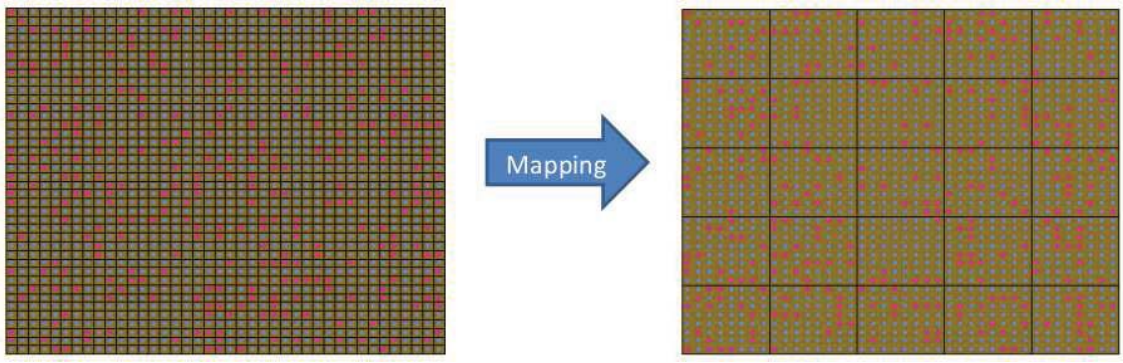


Figure 1. Mapping microscopic lattice sites into coarse-grained (CG) cells. Red dots indicate atoms residing on the lattice.

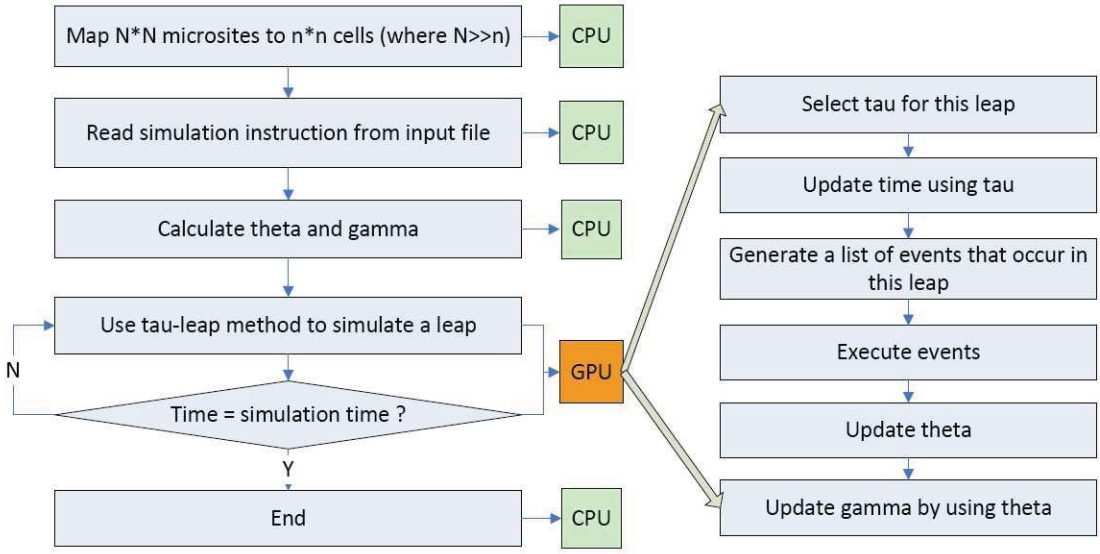


Figure 2. The parallel tau-leap method.

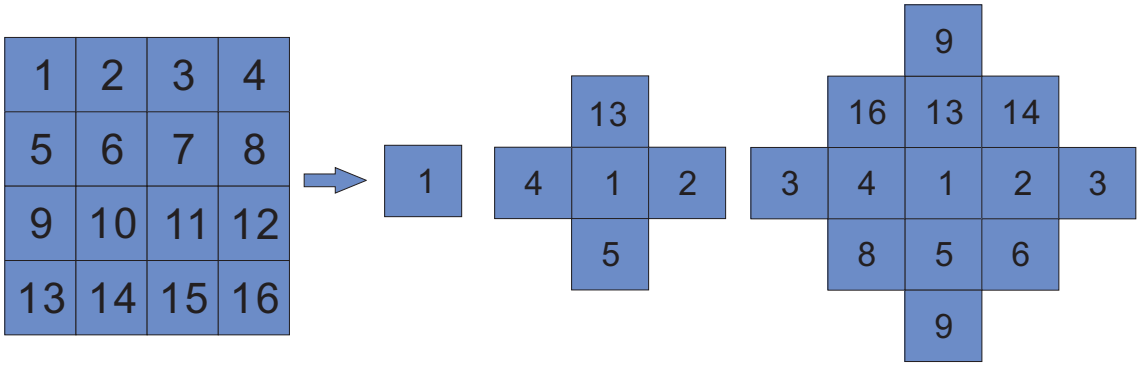


Figure 3. Example of macro-cells with 1-, 2-, and 3-layers centered on cell 1. The numbers indicate CG cells.

ulation steps, we have to consider that data in this memory is available only to the threads in the same block. Thus, both the data assigned to each single thread and the threads' execution have to be reorganized accordingly. We address this requirement with a multi-layer tau-leap method in which we redefine the data structures to be a collection of neighboring CG cells grouped into clusters of cells, termed hereafter macro-cells. In other words, each cell is replaced by a

macro-cell including its multiple neighbor cells organized in rhomboid shapes and each thread or block deals with one macro-cell. Macro-cells can have different sizes (or layers). Figure 3 shows three macro-cells with 1-, 2-, and 3-layers respectively where the 1-layer implementation corresponds to the GPU shared memory implementation described in the section above. We also redesign the way events are executed. Each thread simulates events in one macro-cell. Single cells are replicated across multiple macro-cells (see Figure 4) and, thus, events associated with a cell are also replicated across macro-cells (or threads).

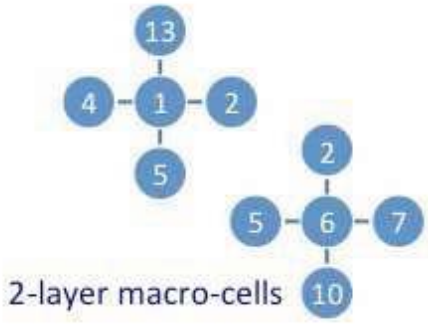


Figure 4. Example of two overlapping 2-layer macro-cells.

In a 2-layer implementation, each macro-cell includes a centered CG cell, like for the implementations described in Section II, and a centered CG cell's four neighbor CG cells (see Figure 3). In leap 1, threads execute the events for all the five cells in their assigned macro-cell; in leap 2, threads execute the events for the centered CG cell only, since the four neighbor CG cells may contain obsolete data at this point. This is not the case for the centered CG cell. Synchronizations across threads sharing cells are performed every two leaps. If we enlarge the macro-cell to 3 layers, then each macro-cell includes one centered cell, its four neighbors, and the neighbor cells of these four neighbors (see Figure 3). In a 3-layer implementation, the synchronization occurs every 3 leaps. By doing so, our tau-leap method uses small, adjacent portions of the global memory that fit in shared memory and are accessed simultaneously by multiple threads. One important aspect of our algorithm is the dynamic change of the macro-cell size as the simulation evolves. More in general, in an n-layer implementation, we start with an n-layer structure per thread in leap 1, an (n-1)-layer structure in leap 2, ..., and a 1-layer structure in leap n. At the end of each leap, the thread peals the macro-cell of the most external shell of cells removing them from the computation. The parameter synchronization among threads is performed every n leaps. This allows us to conserve the accuracy and, at the same time, massively parallelize the execution with communications among threads only every n leaps. On GPUs, the multi-layer tau-leap method can have different degrees of thread parallelism from course-grained to fine-grained parallelism. In a coarse-grained algorithm, one thread is in charge of one macro-cell. In a fine-grained algorithm, one block is in charge of one macro-cell and each block has as many threads as the number of cells in one macro-cell.

C. Multiple vs. Single GPUs

The maximum size (in number of cells) of a molecular system on a single GPU is limited by the maximum number of threads the GPU can serve, i.e., 65,536. In other words, systems with more than 65,536 cells cannot be simulated with a k-layer tau-leap method on a single GPU but can be simulated using multiple GPUs. Again, different parallelizations can be implemented. In a naive approach based on the simple example provided by NVIDIA in the CUDA SDK, we assign one GPU to each CPU thread. We also assign simulation regions to each GPU and use a 2-layer fine-grained algorithm to compute partial results for each region. The partial results are copied back to the CPU at the end of two leaps where all CPU threads write their data to a specified memory location and update it. At this point the next two leaps can start. This implementation has a major drawback: the communication between CPUs and GPUs as well as the communication among CPU threads.

To address this problem, we combine CUDA and OpenMP, and use the new memory features in CUDA2.2+ as a shared memory between CPU threads. The first feature is *portable pinned memory (ppm)* which is available to all host threads once allocated. Portable pinned memory can be used as a shared memory among CPU threads and it can be freed by any CPU thread. It works for contexts that predate the allocation and contexts that are created after the allocation. The second feature is *mapped pinned memory (mpm)* which is also called "zero-copy". Programmers can allocate mapped memory on the host side that can ultimately be mapped to a GPU's address space. By using API functions supplied by CUDA, we can get the address of this memory on the device side. With mapped pinned memory, there is no explicit memory copy between CPU and GPU. The copy is implicitly performed while the kernel function is launching. The third feature is *write combined memory (wcm)* which can free up L1 and L2 cache resources, making more cache available to the rest of the application. Since write-combined memory is not snooped during transfers across the PCI Express bus, performance can be improved by up to 40% [8].

In addition to the naive implementation described above (Multi-GPU), we implemented three other versions of our tau-leap method for multi-GPUs: (1) a version including all the three features in which we combine CUDA2.2, OpenMP, portable pinned memory, mapped pinned memory, and write-combined memory (Multi-GPU OpenMP+ppm+mpm+wcm); (2) a version in which we

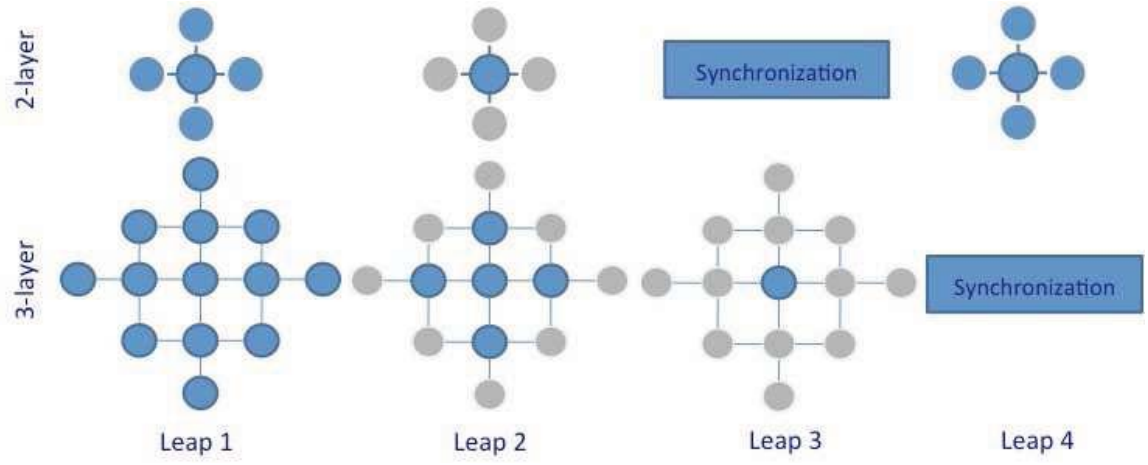


Figure 5. Synchronizations with 2- and 3-layer algorithms.

combine CUDA2.2, OpenMP, portable pinned memory, and mapped pinned memory (Multi-GPU OpenMP+ppm+mpm); and (3) a version in which we combine CUDA2.2, OpenMP, and portable pinned memory only (Multi-GPU OpenMP+ppm). The selection of the features in each implementation is based on information in the CUDA programming guide [8]. Some of these features are indeed beneficial only for special cases. For example, for GPUs integrated onto the motherboard, mapped pinned memory can eliminate superfluous memory copy. However, for discrete GPUs, mapped pinned memory is beneficial only if memory is read or written exactly once and the read or write is coalesced. The set of configurations chosen in this paper allows us to explore these special cases for our application.

IV. What Parallelism for What Molecular System?

The key question in this paper is how much parallelism is there in the tau-leap method. Different parallelizations are presented in Section III, each with strengths and weaknesses that can be correlated to the profile of the molecular system in general and its size in particular.

A. Performance Characterization

We quantify the impact of the different parallelization techniques presented above in terms of the number of events executed in one millisecond. We compare the performance of CGMC simulations of 128 leaps (each of 0.01s) on GPUs for three case studies:

- Case Study 1: sequential CPU vs. GPU+global memory vs. GPU+shared memory implementations
- Case Study 2: 1-layer vs. 2-layer implementations with different degrees of parallelism (i.e, coarse- vs. fine-grained parallelism) and using shared memory
- Case Study 3: Single-GPU implementations vs. multi-GPU implementations

We consider molecular systems with variable number of CG cells. Molecules within cells are assumed to be well mixed. The molecules of each cell are allowed to interact with and diffuse to nearby cells. For the sequential CGMC on CPU, we used an Intel(R) Core(TM)2 Extreme 3GHz desktop. For our GPU simulations, we used one and two GPUs from a Tesla S1070 system with thirty multiprocessors, 240 cores and 4 GB global memory. The performance values presented are the average values computed over three repeated simulations. The standard deviation is not reported because it is close to zero.

Figure 6 shows Case Study 1 where performance values are measured for the simulations of molecular systems with size ranging from 16 to 32,768 cells using a CPU implementation, a GPU implementation using global memory only (GPU+global memory), and a GPU implementation benefiting from shared memory (GPU+shared memory). The simulations are performed on a single GPU. Note that here the emphasis is on the benefits of efficiently using the memory organization on GPUs to gain performance compared to the sequential CPU implementation. As showed in the figure, the performance of the CPU implementation remains constant as the number of cells grows. For the GPU implementation using global memory, the performance increases, reaching its peak for a molecular system of 2,048 cells, and then decreases as the number of cells increases. The peak performance is about 16X faster than on CPUs when the system is making full use of all the GPU resources. For the GPU implementation using shared memory, the peak performance is approx. 100X faster than the CPU implementation due to the drastic reduction in global memory access.

Figure 7 shows Case Study 2 where performance values are measured for the simulations of molecular systems with size ranging from 16 to 12,288 cells using a 1-layer implementation (also called GPU+shared memory), a 2-layer implementation with coarse-grained thread parallelism

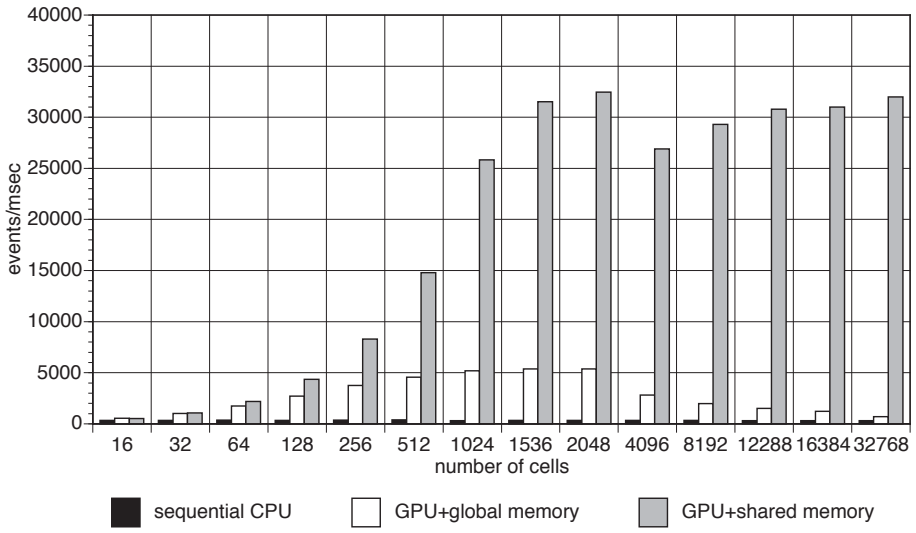


Figure 6. Performance comparison of global vs. shared memory implementations

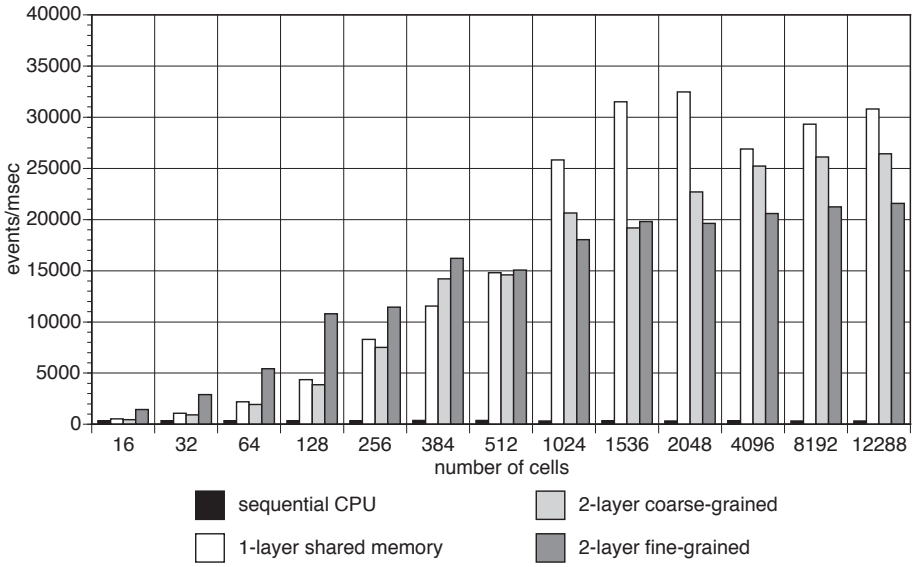


Figure 7. Performance comparison of 1-layer vs. 2-layer implementations.

(2-layer coarse-grained), and 2-layer implementation with fine-grained thread parallelism (2-layer fine-grained). Again the simulations are performed on a single GPU. This time the emphasis is on the size of the system and the more suitable degree of parallelism for a given system size. We observe that the 2-layer coarse-grained parallelism does not outperform the other two approaches. This is because in the first of the two leaps performed sequentially before a synchronization (see Figure 5), the thread has to deal with five cells and all the associated events. Even if, in the next leap, the same thread deals with only the events of the centered cell, the average number of cells considered per leap is still three times higher. The fine-grained thread parallelism addresses this challenge: instead of one single thread, in this implementation each block contains five threads and is in charge of one macro-cell. Each thread now refers to one single cell in the macro-cell. By doing this, the burden of each thread considerably reduces and the five threads can collaborate on the centered cell at every second leap. We are in average 2X faster compared to the 1-layer implementation and about 42X faster than CPUs when the system size is small. This is because 2-layer fine-grained parallelism uses the full GPU resources earlier and the synchronization penalty can be hidden by the massive number of threads. At 512 cells, we identify a sweet-spot when the performance of the 1-layer algorithm matches the

performance of the 2-layer fine-grained algorithm before to overcome the latter.

Figure 8 shows Case Study 3 where performance values are measured for the simulations of molecular systems with size ranging from 2,048 to 102,400 cells using a 1-layer implementation, a 2-layer implementation with fine-grained thread parallelism, and multi-GPU implementations using the 2-layer fine-grained algorithm. Note that for the first and second configurations the max size of the system that can be simulated is 32,768 cells. Here, the key aspect we want to address is the scaling of our simulations beyond traditionally simulated sizes, i.e., 32,000 cells. Note that this limit is dictated by hardware constraints rather than the scientist's interest. As emphasized in Section III, to overcome this limit, we have to use multiple GPUs efficiently. As showed in Figure 8, the naive multi-GPU implementation performs poorly due to the data copied between CPU and GPU at every leap and the expensive communication among CPU threads.

The implementation with CUDA2.2, OpenMP, portable pinned memory, mapped pinned memory, and write-combined memory (Multi-GPU OpenMP+ppm+mpm+wcm) shows poor performance as the number of cells grows because write-combined memory is beneficial for CPU write-only buffers. In our case, our implementation reads and writes multiple times. For the implementation with CUDA2.2, OpenMP, portable pinned memory, and mapped pinned memory (Multi-GPU OpenMP+ppm+mpm), the zero-copy approach is beneficial only if the memory is read or written exactly once and the read or write is coalesced. For our application, we read and write multiple times and we are not coalescing because the number of threads per block is only five (we are indeed using fine-grained thread parallelism). In the coalesced case, the number of threads per block should be 64 or 128 threads. Finally, in the implementation with CUDA, OpenMP, and portable pinned memory (Multi-GPU OpenMP+ppm), we observe excellent performance where OpenMP is beneficial for multi-threaded programming and portable memory serves as a shared memory between CPU threads. The combination of these two techniques can definitely assure scalability. Using multiple GPUs can provide scientists with a platform for simulating larger systems at an average 120X faster speedup than the sequential CPU implementation.

Overall, our analysis clearly identifies three application profiles that benefit from different parallelization methods. When the system is small (with less than 512 cells), it is recommended to run on a single GPU and use a 2-layer fine-grained algorithm. For averaged sized systems with the number of cell ranging between 512 and 4,096, simulations should still run on a single GPU but using 1-layer algorithms. Our multi-GPU implementation based on CUDA2.2, OpenMP, and portable pinned memory makes it feasible to run simulations of large and very large systems, guaranteeing significant gains in performance compared with a CPU implementation.

B. Accuracy Assessment

Clearly, even the fastest CGMC code would be useless if the results it produces are not accurate. Thus, we performed several tests of the accuracy of our implementation by performing simulations of three different species of molecules, A, B, and C, where A can change to B and vice-versa; two B molecules can change to C and vice-versa; and A, B, and C can diffuse. Figures 9(a) and 9(b) show the accuracy for a system of 18,000 molecules as the simulation evolves on both CPU and GPU, respectively. The GPU implementation uses 2-layer coarse-grained parallelism. In both cases the system reaches the equilibrium after approx. 50 leaps and presents the same behavior. Similar results were observed for larger systems and the other methods described in Section III.

V. Conclusion

In this paper we presented an extensive performance evaluation of different parallel implementations of the tau-leap method for CGMC simulations on single and multi-GPUs. Our analysis leads to the identification of the most suitable parallel implementation for CGMC simulations given a molecular system with a certain size. We observed how for small systems (with less than 512 CGMC cells), a GPU implementation of the tau-leap method based on 2-layer fine-grained thread parallelism can achieve up to 42 times speedup in terms of numbers of events per millisecond, compared to the sequential implementation on a CPU. For averaged sized systems (with the number of cells ranging between 512 and 4,096), an implementation based on a 1-layer algorithm can be 100 faster than a CPU implementation. Large systems (i.e., larger than 13,000 cells with a 2-layer fine-grained implementation and 65,536 cells with a 1-layer implementation) cannot be simulated with a single GPU because they require a number of threads higher than the maximum number of threads the GPU can serve. Our multiple GPUs implementation based on CUDA2.2, OpenMP, and portable pinned memory allows us to overcome this limit and is 120X faster than on a CPU. Future work includes the extension of the performance evaluation to molecular systems presenting heterogeneity in their molecular distribution in the simulated region and the performance comparison of our GPU implementations with multi-core CPU implementations.

Acknowledgment

This work was supported by the National Science Foundation grant #0941318 "CDI-Type I: Bridging the Gap Between Next-Generation High Performance Hybrid Computers and Physics Based Computational Models for Quantitative Description of Molecular Recognition", and grant

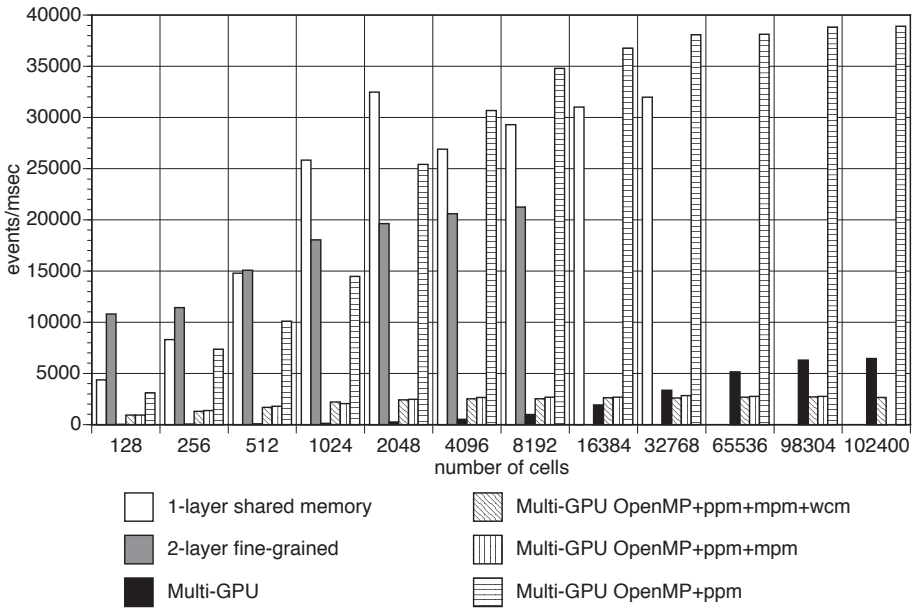
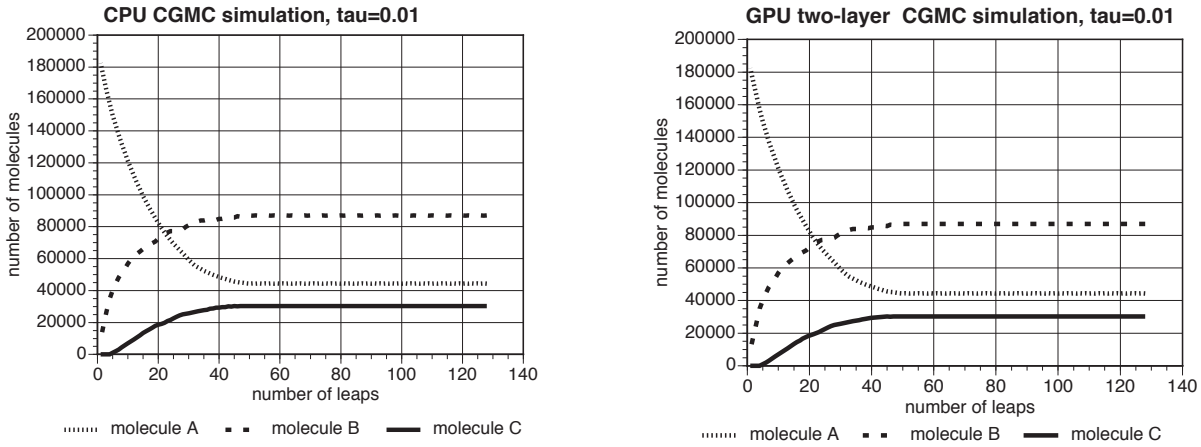


Figure 8. Performance comparison of single vs. multi GPU implementations.



(a) Results on CPU

(b) Results on GPU

Figure 9. Accuracy comparison of simulations on CPU and GPU.

#0922657 "MRI: Acquisition of a Facility for Computational Approaches to Molecular-Scale Problems"; by the U.S. Army, grant #YIP54723-CS "Computer-Aided Design of Drugs on Emerging Hybrid High Performance Computers", by DOE, grant #DE-FG02-05ER25702, and by the NVIDIA University Professor Partnership Program. The authors want to thank Sumit Gupta from NVIDIA for his valuable advice with multi-GPUs.

REFERENCES

[1] Y. Cao, D. Gillespie, and L. Petzold. Avoiding Negative Populations in Explicit Poisson Tau-leaping. *Journal of Chemical Physics*, 123(5):541041–0541048, 2005.

[2] A. Chatterjee, D. Vlachos, and M. Katsoulakis. Binomial Distribution based Tau-leap Accelerated Stochastic Simulation. *Journal of Chemical Physics*, 122:0241121–0241126, 2005.

[3] A. Chatterjee and D. G. Vlachos. Temporal Acceleration of Spatially Distributed Kinetic Monte Carlo Simulations. *Journal of Computational Physics*, 211:596615, 2006.

[4] A. Chatterjee and D. G. Vlachos. An Overview of Spatial Microscopic and Accelerated Kinetic Monte Carlo Methods. *J. of Computer-aided Materials Design*, 14(2):253–308, 2007.

[5] S. D. Collins, A. Chatterjee, and D. G. Vlachos. Coarse-grained Kinetic Monte Carlo Models: Complex Lattices, Multicomponent Aystems, and Homogenization at the Stochastic Level. *J. Chem. Phys.*, 129(18), 2008.

[6] D. Gillespie. Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems. *Journal of Chemical Physics*, 115(4):1716–1733, 2001.

[7] E. Martínez, J. Marian, M. H. Kalos, and J. M. Perlado. Synchronous Parallel Kinetic Monte Carlo for Continuum Diffusion-reaction Systems. *Journal of Computational Physics*, 227:3804–3823, Apr. 2008.

[8] NVIDIA. NVIDIA CUDA Programming Guide Version 2.2. 2009.

[9] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. GPU Accelerated Monte Carlo Simulation of the 2D and 3D Ising Model. *J. of Computational Phys.*, 228(12):4468–4477, 2009.

[10] T.T. Marquez-Lago and K. Burrage. Binomial Tau-leap Spatial Stochastic Simulation Algorithm for Applications in Chemical Kinetics. *J. of Chem. Phys.*, 127(10), 2007.

[11] J.A. van Meel, A. Arnold, D. Frenkel, S.F. Portegies-Zwart, and R.G. Belleman. Harvesting Graphics Power for MD Simulations. *Molecular Simulation*, 2008.

[12] K. Yasuda. Accelerating Density Functional Calculations with Graphics Processing Unit. *J. of Chem. Theory Comput.*, 4(8), 1230-1236 2008.