Fat versus Thin Threading Approach on GPUs: Application to Stochastic Simulation of Chemical Reactions

Guido Klingbeil, Radek Erban, Mike Giles, and Philip K. Maini

Abstract—We explore two different threading approaches on a graphics processing unit (GPU) exploiting two different characteristics of the current GPU architecture. The fat thread approach tries to minimize data access time by relying on shared memory and registers potentially sacrificing parallelism. The thin thread approach maximizes parallelism and tries to hide access latencies. We apply these two approaches to the parallel stochastic simulation of chemical reaction systems using the stochastic simulation algorithm (SSA) by Gillespie [14]. In these cases, the proposed thin thread approach shows comparable performance while eliminating the limitation of the reaction system's size.

Index Terms—Parallel processing, compute unified device architecture (CUDA), graphics processing unit (GPU).

1 Introduction

Graphics processing units (GPUs) using the compute unified device architecture (CUDA) have distinct exploitable architectural characteristics [1]. One is the memory hierarchy consisting of fast on-chip memory and device memory which needs to be managed explicitly by the programmer, and another is the computational grid assigning threads to groups of compute cores. With CUDA GPUs, it is common to use fast on-chip shared memory to avoid high data access latencies. However, shared memory is limited and its size may actually limit the number of threads per block. Another approach is to simply accept the high latency when accessing device, also called global memory, but try to hide it using many parallel threads. This paper compares these two approaches by applying them to the parallel stochastic simulation of chemical reaction systems.

The time evolution of a chemical reaction system is neither continuous since reactions are atomic nor deterministic since a reaction may, but does not have to, occur whenever the appropriate molecules collide. Nevertheless, assuming a sufficiently large molecular population, a well-stirred chemical reaction system can be treated as a deterministic continuous process which can be modeled using ordinary differential equations (ODEs). But in many biological systems, small molecular abundances make deterministic ODE-based simulations inaccurate [2]. As an example, the average copy numbers of some mRNA molecules relevant to the budding yeast cell cycle per cell may be less than 1. Proteins, the actual work horses of the cell, are translated from mRNA [3]. In this scenario, all protein of a certain kind may be translated from a single mRNA molecule [4]. This means depending on even a single event occurring, the transcription of an mRNA molecule, the system may take different paths. The behavior of the reaction system in a single cell, like switching between two states, may depend on stochastic fluctuations [5], [6].

Stochastic simulations compute every molecular reaction event. While ODE-based simulations are computationally inexpensive, the computational time of a stochastic simulation increases with the molecular population and the complexity of the system such that the computational effort for realistic biochemical systems may be prohibitive. Parallel simulations on GPUs can successfully address this problem. In this paper, we show that the thin threading approach is superior to fat threading for larger biochemical reaction networks.

In general, there are two approaches to parallel stochastic simulations of chemical reaction systems: computing multiple independent replications in parallel (MRIP) or a single replication in parallel (SRIP) [7], [8]. The MRIP spawns multiple independent simulations on multiple processors. It is important to use different random seeds such that the processes are uncorrelated. SRIP relies on cooperating threads, either by dividing the simulated model into submodels and computing each of them on a different processor [7], or by computing parts of a single SSA step in parallel like updating the propensities or computing the tentative reaction times in parallel [9]. We are computing many realizations independently in an MRIP approach. No synchronization between the threads is required during the computation.

- *G. Klingbeil is with the Centre for Mathematical Biology, Mathematical Institute, University of Oxford, Oxford OX1 3LB. E-mail: klingbeil@maths.ox.ac.uk.*
- *R. Erban is with the Oxford Centre for Collaborative Applied Mathematics and the Centre for Mathematical Biology, Mathematical Institute, University of Oxford, Oxford OX1 3LB. E-mail: erban@maths.ox.ac.uk.*
- *M. Giles is with the Oxford-Man Institute of Quantitative Finance, and the Mathematical Institute, University of Oxford, Oxford OX1 3LB. E-mail: mike.giles@maths.ox.ac.uk.*
- *P.K. Maini is with the Centre for Mathematical Biology and the Oxford Centre for Integrative Systems Biology, University of Oxford, Oxford OX1 3QU. E-mail: maini@maths.ox.ac.uk.*

Manuscript received 27 May 2010; revised 21 Aug. 2010; accepted 29 Oct. 2010; published online 25 May 2011.
Recommended for acceptance by P. Stenstrom.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-05-0318.
Digital Object Identifier no. 10.1109/TPDS.2011.157.

Thin threading uses multithreading and data locality optimization to hide memory access latency. The idea is to perform a context switch whenever memory is accessed. Latency hiding by multithreading has been used extensively by the Cray MTA-2 system and the Sparcle multiprocessor design [10], [11]. The Cray MTA-2 requires at least 21 threads per processor to hide memory access latency. Data locality optimization exploits the memory hierarchy to improve memory access. One technique is data *tiling*, sorting or rescheduling data access such that they suit the underlying architecture best, sequential memory access operation accessing locations fitting into a cache line [12].

This paper is organized as follows. Section 2 introduces the stochastic simulation of chemical reaction systems. In Section 3, parallel computing on a GPU is briefly reviewed. Section 4 introduces the idea of fat and thin threads on a GPU. Section 5 presents the benchmark results and Section 6 concludes this paper.

2 Stochastic Simulation Algorithm

The Gillespie stochastic simulation (SSA) of chemical reaction systems answers two questions [13], [14]. First, when does the next reaction event occur? Second, of which kind is this event? The SSA is an exact algorithm to simulate the time evolution of realizations consistent with the chemical master equation of a well-mixed chemical reaction system. One time step of the Gillespie SSA is given in Listing 1. The SSA considers a spatially homogeneous (well mixed) molecular system of M reactions $R_i, i = 1, \ldots, M$, in thermal equilibrium of the N molecular species $S_j, j = 1, \ldots, N$. The current state vector

$$\mathbf{X}(\mathbf{t}) = (x_1(t), \ldots, x_N(t)),$$

where $x_j(t)$ is the number of molecules of chemical species $S_j(t), j = 1, 2, \ldots, N$.

The M reactions $R_i, i = 1, \ldots, M$, are accompanied by the vector of propensity vector

$$\boldsymbol{a}(\mathbf{X}(\mathbf{t})) = (a_i(\mathbf{X}(t)), \ldots, a_M(\mathbf{X}(t)))$$

and the state change vectors $\boldsymbol{\nu}_i$ which compose the $N \times M$ stoichiometric reaction matrix ν. The N components

$$\nu_{ij}, j = 1, \ldots, N,$$

of the change vector $\boldsymbol{\nu}_i$ describe the change in the number of molecules of species S_j due to an R_i reaction event. The term $a_i(\mathbf{X}(t))dt$ is the probability, given the current state $\mathbf{X}(t)$, that an R_i reaction event will occur in the next infinitesimal time interval $[t, t + dt)$. The total propensity function $a_0(\mathbf{X}(t))$ is the sum of all propensity functions [13], [14]

$$a_0(\mathbf{X}(t)) = \sum_{i=1}^{M} a_i(\mathbf{X}(t)).$$

The core of the SSA is to use random numbers to choose state transitions. The time τ at any given state of the system $\mathbf{X}(t)$ until the next reaction fires at $t + \tau$ is exponentially distributed with mean $a_0(\mathbf{X}(t))$. Then, $a_i(\mathbf{X}(t))/a_0(\mathbf{X}(t))$ is the probability that the i-th reaction occurs.

```
1  while t < T do:
2    Calculate Propensity functions: a_i.
3    Calculate a_0(X(t)) = sum_{i=1}^{M} a_i(X(t)).
4
5    Sample two random numbers r_1, r_2
       from a uniform distribution.
6
7    Select tau = -(1/a_0) ln(r_1).
8
9    Select the reaction k to fire such
       that sum_{i=1}^{k-1} a_i < r_2 a_0 < sum_{i=1}^{k} a_i.
10
11   Update X according to nu_k:
12   Let X(t + tau) = X(t) + nu_k.
13   Let t = t + tau.
14 end;
```

Listing 1. *Main loop pseudocode of the SSA algorithm.*

Several more efficient reformulations of the exact SSA have been proposed like the next reaction method (NRM) by Gibson and Bruck [15], the logarithmic direct method (LDM) by Li and Petzold [16], or the optimized direct method (ODM) by Cao et al. [17]. The aim of this paper is not to compare the algorithms, but to investigate two threading approaches on GPUs; thus, the SSA is used.

3 Parallel Computing on a GPU

NVIDIA's compute unified device architecture enables efficient stochastic simulations in parallel using GPUs. While GPUs emerged as dedicated graphics boards to accelerate 3D graphics, these devices have recently been used for general purpose parallel computations. NVIDIA introduced CUDA in November 2006 in its GeForce 8800 GPU [18], [19]. GPUs are especially well suited for problems where the same set of instructions can be applied to several data sets simultaneously. This is called single-instruction multiple data (SIMD). Even if the problem itself cannot be well parallelized, several instances can be executed in parallel. The stochastic simulation of chemical reaction systems is not well parallelizable due to the requirement of a single chain of random numbers, but several instances of the simulation can be concurrently computed.

CUDA enables one to apply GPUs to parallel general purpose computations. A NVIDIA CUDA enabled GPU consists of a set of streaming multiprocessors (SMs). Each SM currently aggregates eight single-precision and one double-precision floating point processing cores called stream processors (SPs) accompanied by two special function units (SFUs). The SM is the smallest independent unit of scheduling with its own instruction cache, thread select logic, 64 kB of register file, 16 kB of shared memory, and 8 kB of constant cache.[1] Each SM is a threaded single-issue processor with SIMD. Each SM can execute up to eight thread blocks with a total of 1,024 threads; this is 128 threads per SP, concurrently. So, each of the SM's eight SP has a 2 K 32-bit entry register file. The register's usage is assigned at compile time. The graphics card's memory is called global memory. It is accessible by all

1. Based on NVIDIA GTX 200 generation GPUs [1].

threads of all SMs, but it has a high access latency of 400 to 600 clock cycles [1].

Each SP can perform two floating point operations per cycle by its multiply add unit. Each SFU can perform four instructions per cycle. This gives a total per SM of 16 operations per cycle for the eight SPs and eight for the two SFUs [20].

CUDA uses a variation of SIMD, as used in vector computers, called single-instruction multiple thread (SIMT). The SIMT architecture applies one instruction to multiple independent threads in parallel achieving data level parallelism. In contrast to vector computers using SIMD, where a single instruction is applied to all data lanes, the threads are scheduled in groups, called a "warp," of 32 threads. Threads of a warp either execute the same instruction or remain idle. This allows threads of the warp to branch and take other execution paths. The execution of threads taking different branches is serialized decreasing the overall performance. The advantage of SIMT over SIMD is the independent scheduling of thread warps. This gives a higher flexibility and efficiency when branches occur, since only diverging threads within a warp need to be serialized. Obviously, full efficiency is gained when all threads of a warp take the same execution path [18].

Multiple warps of threads can be grouped into blocks which again form a computational grid. Blocks are assigned to the SMs and the threads of a block can cooperate and share cached memory areas and shared memory. Each thread is able to access its own set of local variables in local memory. Variables in local memory are stored in registers (fast access, but read after write penalty of two cycles). If register space is exceeded, local variables spill into global memory.

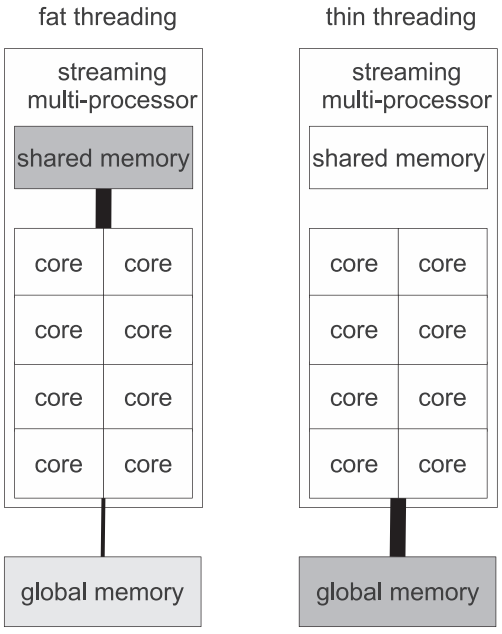


Fig. 1. Outline of the fat threading approach on the left and the thin threading approach on the right. Each streaming multiprocessor contains eight processing cores and 16 kB of shared memory. The thickness of the lines between the computational cores and shared/global memory as well as the shading of the shared/global memory indicates the usage. The fat thread approach uses the shared memory to avoid the global memory access latency, while the thin thread approach does not use shared memory and hides latency by using large numbers of parallel threads.

4 Fat versus Thin Threads

The idea to distinguish a fat threading approach from a thin threading approach is based on the characteristics of modern GPUs. In contrast to a central processing unit (CPU), where many details, such as caching or paging mechanisms, are hidden and hence transparent to the programmer, the GPU exhibits different types of memory to the user and determines the interoperability of threads.[2] The main loop of Gillespie's SSA as given in Listing 1 is executed as a CUDA kernel on the GPU. Regarding both threading approaches, the stoichiometric reaction matrix ν, the reaction rate constants $c_i, i = 1, \ldots, M$, and the seeds of the pseudorandom number generator (PRNG) are initialized on the host and copied and stored in the GPU's cached constant memory. The current time t, final time T, index j of the reaction which occurs in the next time step, the total propensity function $a_0(\mathbf{X}(t))$, and the uniform random numbers r_1 and r_2 are kept in registers.

4.1 Fat Threads

The common approach is to maximize the usage of shared memory and registers to keep the data required by the threads as "close" (in terms of memory access time and latency) as possible. This means to copy a working set of data into shared memory and process it by as many threads as possible in parallel [1].

2. The term transparent describes features such as a cache which are beneficial, but do not require any attention from or work by the programmer.

1. Load data from device memory to shared memory.
2. Synchronize with all the other threads of the block so that each thread can safely execute the chosen algorithm in shared memory.
3. Synchronize with all the other threads of the block again if necessary to make sure that shared memory has been updated with the results.
4. Write the results back to device memory.

The number of threads per block may be restricted by the available shared memory which may again hamper the overall performance.

In order to minimize global memory transactions, the fat thread stochastic simulation software keeps the vector of propensity functions $\mathbf{a}$ and current molecular population vector $\mathbf{X}(t)$ in shared memory. Only when a sample is permanently stored, is it written to global memory as shown in Fig. 1. However, both the propensity function vector $\mathbf{a}$ and the current molecular population vector $\mathbf{X}(t)$ depend on the size of the reaction system and with it the maximum number of possible threads per block. The PRNG state vector is stored in registers.

For each thread, the state vector $\mathbf{X}(t)$ of length N and the propensity function vector $\boldsymbol{a}(\boldsymbol{X}(t))$ of length M need to be stored in shared memory. Additionally, the 128 bytes (32 32-bit wide entries) of shared memory are statically allocated by the compiler. This gives a per block shared memory requirement in bytes of

$$\text{memory}_{\text{shared}} = ((M + N) \times \text{ threads per block} + 32) \times 4.$$

TABLE 1
Number of Reactions and Molecular Species

System	Reactions	Species	Maximum fat thread block size (warps)
Decay-dimerisation	4	3	512 (16)
Oregonator	5	8	288 (9)
Circadian cycle	16	9	160 (5)
lac-operon	16	11	128 (4)
Fully connected network	30	6	96 (3)

The right-most column gives the maximum feasible block sizes using the fat thread approach.

Rearranging this for the number of threads gives the upper bound

$$\text{threads per block} \leq \left\lfloor \frac{4096 - 32}{M + N} \right\rfloor.$$

The maximum block size for the used example reaction systems is given in Table 1. Given the minimum efficient block size is 64 threads or 2 warps [1], the maximum size of the reaction system is given by $\lfloor N + M \rfloor \leq 63$.

4.2 Thin Threads

As discussed earlier, one characteristic of a GPU is the elaborated memory hierarchy, but in contrast to the CPU which relies heavily on caching, the programmer has to manage different memory types to maximize performance. The second characteristic of a GPU is the ability to spawn large numbers of threads. The thin thread approach is to maximize the number of parallel threads per thread block. As a consequence, it is no longer feasible to store the propensity function vector $\boldsymbol{a}(\boldsymbol{X}(t))$ and current molecular population vector $\mathbf{X}(t)$ in shared memory. They are stored in global memory. The high global memory access latency is hidden by the large number of threads and the total number of global memory transactions is minimized by coalesced transfers. The thin thread does not use the above protocol, but works in global memory similarly as one would do using a CPU. This is also illustrated in Fig. 1. The PRNG state vector is moved from registers to share memory. This is not contradictory to the aim of this paper, since registers are the fastest storage available [21]. The second motivation for the thin thread approach is hiding register dependencies or arithmetic latency. When an instruction within a thread writes a result into a register, there is an approximately 24 clock cycle latency until the thread is able to use this result. To hide this arithmetic latency, multiprocessors should be running at least 192 threads or 6 warps [21].

The maximum block size of the thin threading approach is independent of the reaction system 512 threads per block, which is the maximum block size using current CUDA capable GPUs [1].

4.3 Coalesced Memory Access

If the threads of a block access global memory in the right pattern, the GPU is able to bundle a number of these accesses into one memory transaction. This is called a coalesced memory transaction. The underlying idea of memory coalescing is similar to a cache line. A cache line is either completely replaced or not at all. Even if only a single data item is requested, the entire line is read. If now a neighboring item is subsequently requested, it is already in the cache.

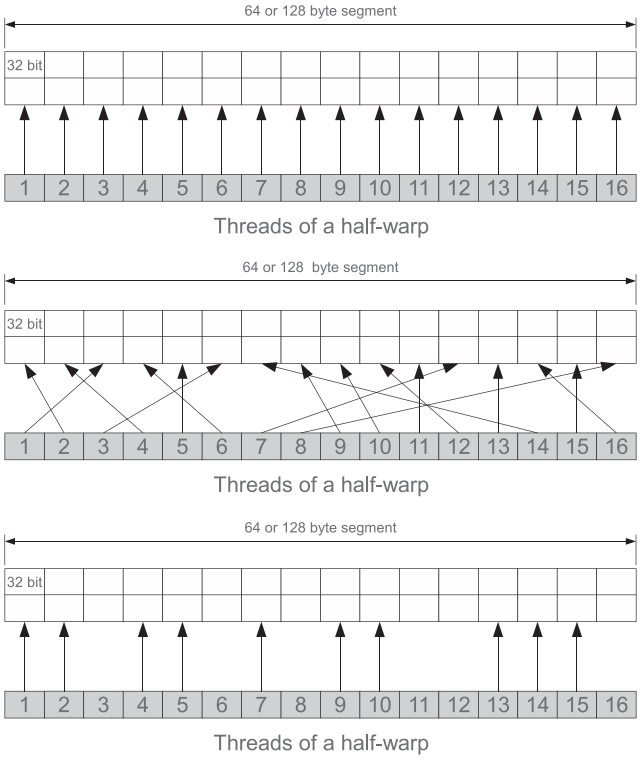


Fig. 2. Examples of coalesced global memory access patterns using a GPU with compute capability 1.2 and beyond grouping all access of 16 threads into one transaction. The top panel shows a consecutive access. The middle shows a random access onto a 64- or 128-byte segment. The bottom panel shows that not all threads participate in the transaction. Even though the achieved bandwidth is not increased 16 times, it is still much higher than using noncoalesced global memory accesses.

The stochastic simulation software only uses 32-bit, or 4 byte word, values. Using a GPU with compute capability greater than or equal to 1.2, the global memory access of all threads of a half-warp is coalesced into a single memory transaction if certain requirements are met. If the words accessed by the threads of a half-warp are in a 128-byte segment, the memory transactions are coalesced as shown in Fig. 2.

Memory transactions are issued per half-warp, either the lower or upper 16 threads of a warp. The following protocol is used to issue a memory transaction and to determine the number of bytes transferred [1]:

1. Find the 128-byte memory segment that contains the address requested by the lowest numbered active thread.
2. Find all other active threads whose requested address lies in the same segment and reduce the transferred number of bytes if possible:
   a. If the transaction is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes.
   b. If the transaction is 64 bytes and only the lower or upper half is used, reduce the transaction size to 32 bytes.

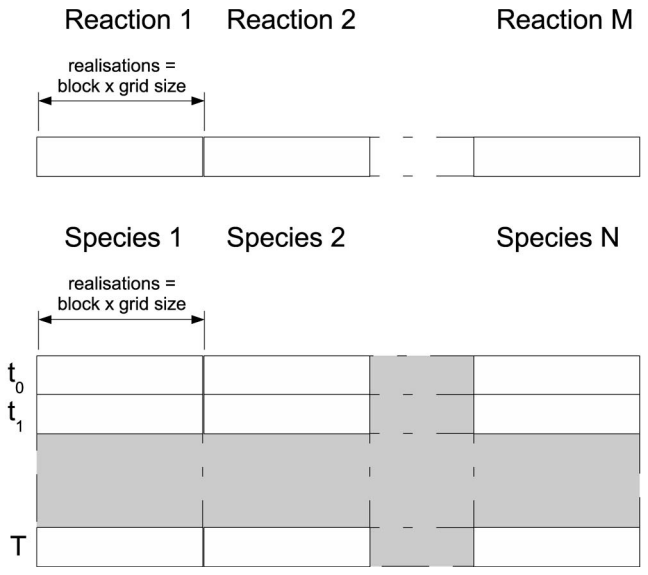


Fig. 3. The coalesced global memory layout used. Storage of the propensity functions of the M reactions of the reaction system is shown on the top, of the time evolution data at the bottom. The time evolution data were stored row wise. Each row corresponds to one sampled time point. Each row consists of N blocks of size equal to $\text{block size} \times \text{grid size}$ holding the per realization molecular populations of one molecular species.

3. Carry out the transaction and mark the serviced threads as inactive.
4. Repeat until all threads in the half-warp are serviced.

4.4 Minimizing the Number of Memory Transactions

The key to minimizing memory transactions by coalescing is an appropriate storage layout taking the characteristics of the SSA into account. The global memory access scheme used to store the time evolution data is shown in Fig. 3. Each thread is computing a single realization. The storage addresses of the molecular populations and propensity functions are grouped by species.

This scheme works perfectly for the propensity functions if all threads update the same molecular species. If the threads do not fire the same reaction, they do not necessarily update the same species. But due to the large block size, there will still be updates in the same 128-byte segment which can be coalesced, thus minimizing the total global memory transactions. Whenever multiple threads update the same molecular species, memory access is coalesced.

5 Results

The performance of the thin thread approach implementation of the SSA algorithm on GPUs is assessed in two steps. First, we assess the optimum block size using a small example system. Second, the fat and thin thread approaches to parallel stochastic simulation using GPUs are compared to the sequential implementation on a CPU.[3] The results are given as a speedup (ratio of sequential runtime on a CPU to parallel runtime on the GPU) as well as in runtime in milliseconds. The base case computes a realization of the stochastic simulation on a single CPU core. This is a reasonable choice using an MRIP approach. This is the sequential runtime per realization and assumes a linear scaling on the CPU which is, by Amdahl's law, the best case and a conservative estimate [22]. Our results do not include transfers between the host and the GPU.

3. To ensure an unbiased comparison, all used algorithms are implemented in C.

5.1 Example Systems

We use five example reaction systems of varying size. They are described in detail in the Supplemental Online Material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.157. The first example system is a decay-dimerization model used by Gillespie [23], and the second is the Oregonator, a simulation of the oscillating Belousov-Zhabotinskii reaction discovered in 1951 [24]. The third example is a simple model of a circadian oscillator. Many organisms use a circadian clock to keep internal sense of daily time and regulate their behavior accordingly [25]. The fourth is a simplified simulation of the *lac*-operon presented in [26]. The *lac*-operon consists of three genes and is required for the transport and metabolism of lactose in some bacteria, e.g., *Escherichia coli*. It is regulated by several factors including the availability of glucose and of lactose. The *lac*-operon is one of the foremost examples of prokaryotic gene regulation. The final example is a fully connected reaction network consisting of six chemical species. The size of these reaction systems in terms of the number of reactions and molecular species involved is given in Table 1.

5.2 Latency Hiding and Register Pressure

The thin threading approach is based on minimizing memory transactions by coalescing and hiding memory and arithmetic latency by large block sizes. It is now obvious to ask whether there is an optimal block size. The fat thread approach uses up to 128 threads or 4 warps per block, so we use 160 up to 512 threads or 5 to 16 warps per block with the thin threading. We investigate the influence of the block size by simulating the time evolution of the Oregonator system. The runtime per realization across the increasing block sizes is given in Table 2. Since threads are scheduled in warps of 32 threads, the block size is incremented by 32 threads. Against intuition, the best performance is at block sizes of 192 (6 warps) or 384 (12 warps) threads per block, respectively. All further calculations are performed with a block size of 384 threads or 12 warps per block.

It has to be noted that our SSA implementation requires 33 32-bit register entries per thread. At 16384 32-bit register entries available per SM, this allows for up to 455 threads per block. To allow a block size of 512 threads per block, the register usage has to be limited to 32 registers per thread. This may potentially hamper the performance at this block size [27].

5.3 Comparing Fat and Thin Threading

Using the fat thread approach, the block size computing the simulations is set according to Table 1 and set to 384 using the thin thread approach.

TABLE 2
Determining the Optimal Block Size in Latency Hiding

Threads (warps)	Blocks	Realisations	Time per realisation $[ms]$
160 (5)	126	20160	11.7
192 (6)	105	20160	9.95
224 (7)	90	20160	13.81
256 (8)	79	20224	10.9
288 (9)	70	20160	11.8
320 (10)	63	20160	12.7
352 (11)	57	20064	13.8
384 (12)	53	20352	9.9
416 (13)	49	20384	10.64
448 (14)	45	20160	11.5
512 (16) [4] [5]	40	20480	13.0

Runtime per realization in milliseconds while varying the block size from 160 (5 warps) to 512 (16 warps) threads per block. Counterintuitive to latency hiding, the largest block size of 512 threads does not yield the best performance. Both block sizes of 192 and 384 threads, respectively, yield the best performance.
[4]. Limited to 32 registers per thread.
[5]. The maximum block size for GPUs with compute capability greater or equal than 1.3 is 512 threads [1].

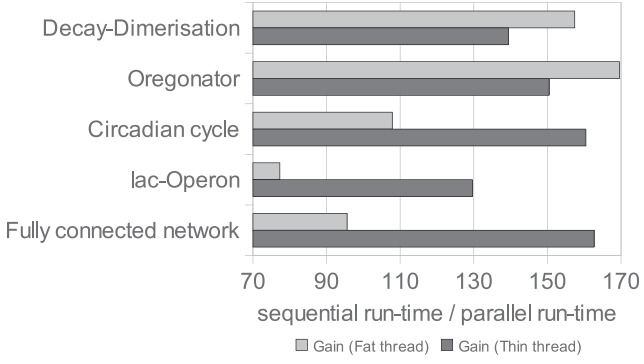


Fig. 4. Comparing the gain, ratio of the sequential runtime of the CPU implementation and the parallel GPU implementation, of the fat and thin thread approaches.

Since the actual runtime depends on the chosen hardware, the results are stated in terms of gain or speed-up comparing the parallel GPU implementation to the sequential CPU one. The speedup for the example systems used is given in Fig. 4. While the fat thread approach outperforms the thin thread one on the dimerization-decay and Oregonator system, the thin thread approach performs better on the *Lac*-operon, circadian cycle, and the fully connected network. The fat thread approach seems to be favorable for smaller reaction systems when its small size allows large block sizes.

To investigate whether the performance of the fat thread approach depends on the block size and with it on the size of the reaction system, we simulated 1, 2, 4, and 6 decay-dimerization systems at once. The details of these models are given in the Supplemental Online Material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.157. The runtime ratio of the fat thread and thin thread approaches is shown in Fig. 5. A ratio smaller than 1 means that the fat thread approach is faster.

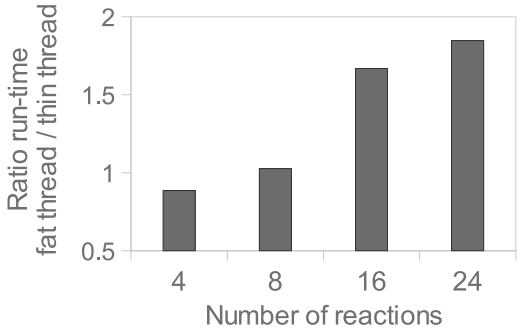


Fig. 5. Runtime ratio of the fat thread and thin thread approaches. A ratio smaller than 1 means that the fat thread approach is faster, and greater than 1 means that the thin thread is faster. To create larger reaction systems with similar characteristics, multiple (1, 2, 4, and 6) dimerization-decay reaction systems are concatenated to create reaction systems with 4, 8, 16, and 24 reactions, respectively.[6] Even for eight reactions, the thin thread approach is slightly faster than the fat thread. For 16 and 24 reactions, the thin thread approach is clearly beneficial.

6 Discussion and Conclusion

We compared two threading strategies on GPUs with the sequential computation on a CPU. We applied both strategies to the stochastic simulation of four biologically meaningful and one artificially constructed example systems. The fat threading follows the recommended approach using the fast on-chip shared memory to minimize global memory accesses and their latency. We propose a thin threading strategy not using the shared memory, but hiding the arithmetic and access latencies by large block sizes. The thin thread approach basically ignores the GPU memory hierarchy, especially shared memory, and uses memory in a more CPU-like fashion. However, the programmer still has to ensure coalesced memory transfers to hide access latency.

Thin and fat threading approaches show comparable performance. However, the thin threading approach relieves the stochastic simulation software of one of its major limitations, the size of the reaction system to be simulated by the size of shared memory.[6]

The optimal block size in latency hiding is not, as one might expect, at 512 threads per block, but either at 192 (6 warps) or 384 (12 warps) threads per block.

Fat threading uses shared memory as a CPU-style cache. It is not used for thread synchronization or cooperation. Thin threading neglects shared memory and uses an easier more CPU-like programming model without a performance loss. If thread synchronization and caching compete for the limited shared memory, thread synchronization should take priority since memory latency can be successfully hidden by using a thin thread approach.

6.1 Outlook

The stochastic simulation of ensembles of chemical reaction systems is trivially parallel. All simulations are independent. The shared memory is solely used to accelerate computation, but not for thread cooperation. It has to be verified whether the thin threading approach is applicable to problems requiring thread cooperation. Since

6. Details of the example reaction systems used are given in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.157.

the global memory is shared by all threads, it is not a question of feasibility but of performance.

The lately released NVIDIA Fermi architecture increases the amount of shared memory and introduces caches to the GPU. Each SM has a combined 64 kB shared memory and cache area. It can be divided either into 48 kB of shared memory and 16 kB of cache or *vice versa* [28]. The fat thread strategy benefits from the increased shared memory by relieving the pressure on the maximum size of the reaction system. The thin thread approach may benefit from the introduction of caches when different threads update different molecular populations in global memory.

APPENDIX

HARDWARE AND SOFTWARE ENVIRONMENT

All simulations were computed using a NVIDIA GTX 260 GPU with 216 cores and 896 Mb DDR3 RAM. The Host CPU is an Intel Core 2 Duo 6420 at 2.13 GHz and 4 GB RAM. The software environment is OpenSuse 11.1 with gcc 4.2.3, NVIDIA CUDA 3.0, and MATLAB 2010a. The stochastic simulation software requires a GPU with compute capability 1.2 or higher. The GPU code is integrated into MATLAB using the MEX C interface.

ACKNOWLEDGMENTS

GK was supported by the Systems biology Doctoral Training Center (DTC) and the Engineering and Physical Sciences Research Council (EPSRC). This publication was based on work supported in part by Award No. KUK-C1-013-04, made by King Abdullah University of Science and Technology (KAUST). The research leading to these results has received funding from the European Research Council under the *European Community's* Seventh Framework Programme *(FP7/2007-2013)* / ERC *grant agreement* No. 239870. RE would also like to thank Somerville College, University of Oxford for Fulford Junior Research Fellowship. MG was supported in part by the Oxford-Man Institute of Quantitative Finance, and by the United Kingdom Engineering and Physical Sciences Research Council under research grant EP/G00210X/. PM was partially supported by a Royal Wolfson Merit Award.

REFERENCES

[1] *Nvidia CUDA Programming Guide, Version 2.1,* NVIDIA Corporation, 2701 San Tomas Expressway, vol. 12, 2008.

[2] R. Erban, S. Chapman, I. Kevrekidis, and T. Vejchodsky, "Analysis of a Stochastic Chemical System Close to a Sniper Bifurcation of Its Mean-Field Model," *SIAM J. Applied Math.,* vol. 70, no. 3, pp. 984-1016, 2009.

[3] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter, *Molecular Biology of the Cell.* Garland Science, 2007.

[4] F. Holstege, E. Jennings, J. Wyrick, T. Lee, C. Hengartner, M. Green, T. Golub, E. Lander, and R. Young, "Dissecting the Regulatory Circuitry of a Eukaryotic Genome," *Cell,* vol. 95, no. 5, pp. 717-728, Nov. 1998.

[5] J. Hasty, D. McMillen, F. Isaacs, and J. Collins, "Computational Studies of Gene Regulatory Networks: In Numero Molecular Biology," *Nature Rev. Genetics,* vol. 2, no. 4, pp. 268-279, Apr. 2001.

[6] T. Tian and K. Burrage, "Stochastic Models for Regulatory Networks of the Genetic Toggle Switch," *Proc. Nat'l Academy of Sciences of USA,* vol. 103, no. 22, pp. 8372-8377, 2006.

[7] G. Ewing, D. McNickle, and K. Pawlikowski, "Multiple Replications in Parallel: Distributed Generation of Data for Speeding up Quantitative Stochastic Simulation," *Proc. Int'l Assoc. for Mathematics and Computers in Simulation (IMACS '97),* pp. 397-402, 1997.

[8] L. Dematte and T. Mazza, "On Parallel Stochastic Simulation of Diffusive Systems," *Proc. Sixth Int'l Conf. Computational Methods in Systems Biology (CMSB '08),* pp. 191-210, 2008.

[9] T. Tian and K. Burrage, "Parallel Implementation of Stochastic Simulation for Large-Scale Cellular Processes," *Proc. Eighth Int'l Conf. High Performance Computing and Grid in Asia-Pacific Region,* vol. 0, pp. 621-626, 2005.

[10] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K.S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-Processor Performance on the Tera MTA," *Proc. IEEE/ACM Conf. Supercomputing (SC '98),* pp. 4-4, 1998.

[11] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, D. Yeung, G. D'souza, and M. Parkin, "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *IEEE Micro,* vol. 13, no. 3, pp. 48-61, June 1993.

[12] F. Irigoin and R. Triolet, "Supernode Partitioning," *POPL '88: Proc. 15th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 319-329, 1988.

[13] D. Gillespie, "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions," *J. Computational Physics,* vol. 22, pp. 403-434, 1976.

[14] D. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions," *J. Physical Chemistry,* vol. 81, no. 25, pp. 2340-2361, 1977.

[15] M. Gibson and J. Bruck, "Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels," *J. Physical Chemistry A,* vol. 104, pp. 1876-1889, 2000.

[16] H. Li and L. Petzold, "Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems," technical report, Dept. of Computer Science, Univ. of California, http://www.cs.ucsb.edu/cse/publications.php, 2006.

[17] Y. Cao, H. Li, and L. Petzold, "Efficient Formulation of the Stochastic Simulation Algorithm for Chemically Reacting Systems," *J. Chemical Physics,* vol. 121, no. 9, pp. 4059-4067, 2004.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia Tesla: A Unified Graphics and Computing Architecture," *IEEE CS Hot Chips,* no. 19, pp. 39-45, Mar./Apr. 2008.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue,* vol. 6, no. 2, pp. 40-53, Mar./Apr. 2008.

[20] P. Maciol and K. Banas, "Testing Tesla Architecture for Scientific Computing: The Performance of Matrix-Vector Product," *Proc. Int'l Multiconf. Computer Science and Information Technology,* vol. 3, pp. 285-291, 2008.

[21] *Nvidia Compute PTX: Parallel Thread Execution, ISA Version 1.4,* NVIDIA Corporation, 2701 San Tomas Expressway, vol. 3, 2009.

[22] G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS '67 : Proc. Apr. 18-20, 1967, Spring Joint Computer Conf.,* pp. 483-485, 1967.

[23] D. Gillespie, "Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems," *J. Chemical Physics,* vol. 115, no. 4, pp. 1716-1733, 2001.

[24] J. Murray, *Mathematical Biology 1: An Introduction,* third ed. Springer Verlag, 2002.

[25] J. Vilar, H. Kueh, N. Barkai, and S. Leibler, "Mechanisms of Noise-Resistance in Genetic Oscillators," *Proc. Nat'l Academy of Sciences of USA,* vol. 99, no. 9, pp. 5988-5992, 2002.

[26] D. Wilkinson, *Stochastic Modelling for Systems Biology.* Chapman & Hall/CRC, 2006.

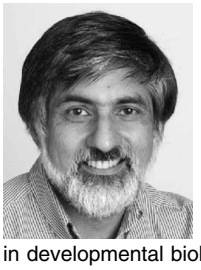
[27] *NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3,* NVIDIA Corporation, 2701 San Tomas Expressway, July 2008.

[28] *NVIDIAs Next Generation CUDA Compute Architecture: Fermi,* NVIDIA Corporation, 2701 San Tomas Expressway, v 1.1, 2009.

Guido Klingbeil received the diploma degree in computer engineering from the Technical University of Hamburg-Harburg. Currently, he is working as a member of the Systems Biology Doctoral Training Centre at the University of Oxford. His research interests include mathematical biology, stochastic simulation algorithms, gene regulatory networks, and the use of GPUs for scientific computing applications.

Radek Erban received the PhD degree from the University of Minnesota. Currently, he is working as a Royal Society University Research Fellow in the Mathematical Institute, University of Oxford. His research interests include mathematical biology, multiscale modeling, partial differential equations, stochastic simulation algorithms, gene regulatory networks, mathematical fluid dynamics, and applications of mathematics in medicine.

Mike Giles received the BA degree in mathematics from the University of Cambridge, and the SM and PhD degrees in aeronautical engineering from MIT. Currently, he is working as a professor of scientific computing in the Mathematical Institute at the University of Oxford, and is also a member of the Oxford-Man Institute of Quantitative Finance and the Oxford e-Research Centre. His research interests include the development and numerical analysis of Monte Carlo methods in computational finance, and the use of GPUs for a wide variety of scientific computing applications.



Philip K. Maini received the DPhil degree in mathematics from Oxford University. Currently, he is working as a professor of mathematical biology and the director of the Centre for Mathematical Biology, Mathematical Institute, Oxford. He is also a part of the Oxford Centre for Integrative Systems Biology, Department of Biochemistry. His research interests include deterministic models of cell and tissue-level interactions to signaling cues with applications in developmental biology, cancer growth, and wound healing.