

Fat vs. 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 minimise data access time by relying on shared memory and registers potentially sacrificing parallelism. The thin thread approach maximises 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 (J. Phys. Chem, Vol. 81, p. 2340-2361, 1977). 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. One is the memory hierarchy consisting of fast on-chip memory and device memory which needs to be managed explicitly by the programmer, another 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 modelled using ordinary differential equations (ODEs). But in many biological systems small molecular abundances make deterministic ODE based simulations inaccurate [1]. As an example, the average copy numbers of some mRNA molecules relevant to the budding yeast cell cycle per cell may be less than one. Proteins, the actual work horses of the cell, are translated from mRNA [2]. In this scenario all protein of a certain kind may be translated from a single mRNA molecule [3]. This means depending on even a single event occurring, the transcription of a mRNA molecule, the system may take different paths. The behaviour of the reaction system in a single cell, like switching between two states, may depend on stochastic fluctuations [4], [5].

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 than fat threading for larger biochemical reaction networks.

This paper is organised 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.

2 STOCHASTIC SIMULATION ALGORITHM

The Gillespie stochastic simulation (SSA) of chemical reaction systems answers two questions [6], [7]. First,

- *Guido Klingbeil is with the Centre for Mathematical Biology, Mathematical Institute, University of Oxford, Oxford OX1 3LB. E-mail: klingbeil@maths.ox.ac.uk*
- *Radek Erban is with the Oxford Centre for Collaborative Applied Mathematics, Mathematical Institute, University of Oxford, Oxford OX1 3LB. E-mail: erban@maths.ox.ac.uk*
- *Mike 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*
- *Philip 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*

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 realisations 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, below. The SSA considers a spatially homogeneous (well mixed) molecular system of M reactions R_i , $i = 1, \dots, M$, in thermal equilibrium of the N molecular species S_j , $j = 1, \dots, N$. The current state vector

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

where $x_i(t)$ is the number of molecules of chemical species $S_i(t)$, $i = 1, 2, \dots, N$.

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

$$\mathbf{a}(\mathbf{X}(t)) = (a_1(\mathbf{X}(t)), \dots, a_M(\mathbf{X}(t)))$$

and the state change vectors ν_i . The ν_i compose the $N \times M$ stoichiometric reaction matrix ν . The N components

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

of the change vector ν_i describe the change in the number of molecules of species S_i due to a R_j reaction event. The term $a_j(\mathbf{X}(t))dt$ is the probability, given the current state $\mathbf{X}(t)$, that a 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 [6], [7]

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

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 $a_0(\mathbf{X}(t))$.

```

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

Listing 1. Main loop pseudocode of the SSA algorithm.

Then $a_j(\mathbf{X}(t))/a_0(\mathbf{X}(t))$ is the probability that the j -th reaction occurs.

Several more efficient reformulations of the exact SSA have been proposed like the next reaction method (NRM) by Gibson and Bruck [8], the logarithmic direct method (LDM) by Li and Petzold [9], or the optimised direct method (ODM) by Cao *et al.* [10]. 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 (CUDA) 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 [11], [12]. 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 can not be well parallelised, several instances can be executed in parallel. The stochastic simulation of chemical reaction systems is not well parallelisable 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 8 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 cache¹. Each SM is a threaded single-issue processor with SIMD. Each SM can execute up to eight thread blocks with a total of 1024 threads, this is 128 threads per SP, concurrently. So each of the SM's eight SP has a 2K 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 threads of all SMs but it has a high access latency of 400 to 600 clock cycles [13].

Each SP can perform two floating point operations per cycle by its multiply-add unit. Each SFU can perform 4 instructions per cycle. This gives a total per SM of 16 operations per cycle for the eight SPs and eight for the two SFUs [14].

CUDA uses a variation of single instruction multiple data (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,

1. Based on NVIDIA GTX 200 generation GPUs [13].

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 stay idle. This allows threads of the warp to branch and take other execution paths. The execution of threads taking different branches is serialised 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 serialised. Obviously, full efficiency is gained when all threads of a warp take the same execution path [11].

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 2 cycles). If register space is exceeded, local variables spill into global memory.

4 FAT VS. THIN THREADS

The idea to distinguish a fat and thin threading approach is based on the characteristics of modern GPUs. In contrast to a central processing unit (CPU), where many details, like 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 threads². Regarding both threading approaches, the stoichiometric reaction matrix ν and the reaction rate constants c_i , $i = 1, \dots, M$, are stored in cached constant memory. The current time t , final time T , index j of the reaction which occurs in the next time step (see Listing 1), the total propensity function $a_0(\mathbf{X}(t))$, and the uniform random number 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 [13]:

- 1) Load data from device memory to shared memory.
- 2) Synchronise with all the other threads of the block so that each thread can safely execute the chosen algorithm in shared memory.

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.

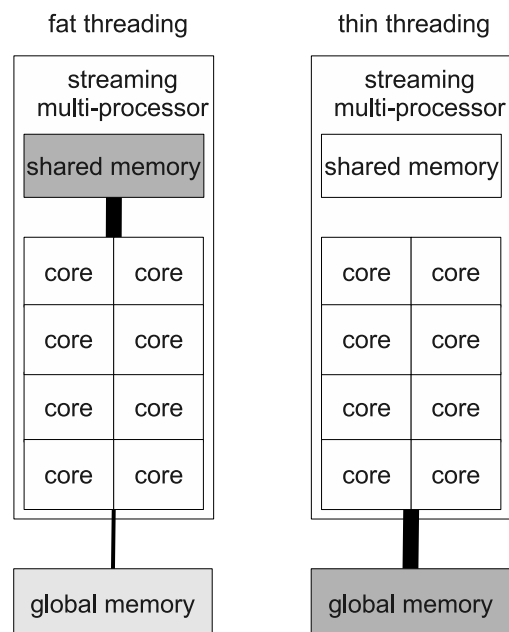


Fig. 1. Outline of the fat threading approach on the left and the thin threading approach on the right. Each streaming multiprocessor contains 8 processing cores and 16 kB of shared memory. The thickness of the lines between the computational cores and shared/global memory as well as the shading of the 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.

- 3) Synchronise 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 again may hamper the overall performance.

In order to minimise 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 it is written to global memory. 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 pseudo random number generator (PRNG) state vector is stored in registers.

For each thread, the state vector $\mathbf{X}(t)$ of length N and the propensity function vector $\mathbf{a}(\mathbf{X}(t))$ 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:

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

Re-arranging 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 [13], 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 elaborate memory hierarchy, but in contrast to the CPU which relies heavily on caching, the programmer has to manage different memory types to maximise performance. A second characteristic of a GPU is the ability to spawn huge 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 $\mathbf{aX}(t)$ 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 minimised by coalesced transfers. The thin thread does not use the above protocol, but works in global memory similar as one would do using a CPU. The PRNG state vector is moved from registers to shared memory. This is not contradictory to the aim of this paper since registers are the fastest storage available [15]. A 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 [15].

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 [13].

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.

The stochastic simulation software only uses 32-bit, or 4

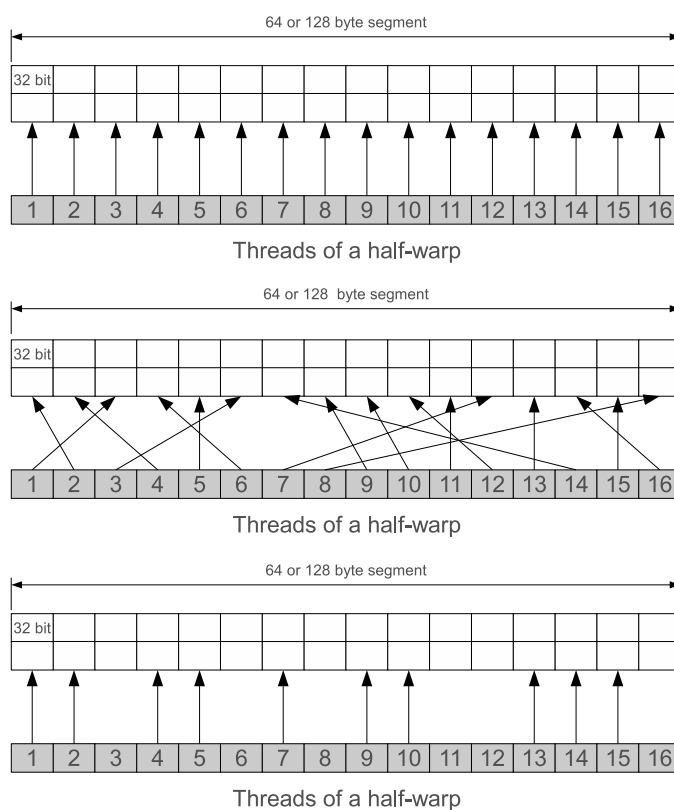


Fig. 2. Examples of coalesced global memory access patterns using a GPU with compute capability 1.2 and beyond grouping all access of 16 threads into one transaction. The top panel shows a consecutive access. The middle is a random access onto a 64 or 128 byte segment. The bottom panel shows that not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

byte word, values. Using a GPU with compute capability greater 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 Figure 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 [13]:

- 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:
 - If the transaction is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes.
 - If the transaction is 64 bytes and only the

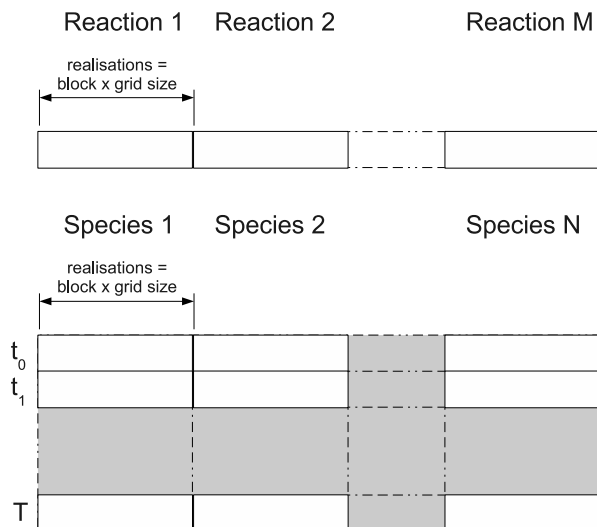


Fig. 3. The coalesced global memory layout used. Storage of the propensity functions of the M reactions of the reaction system is shown at the top, of the time evolution data at the bottom. The time evolution data are stored row wise. Each row corresponds to one sampled time point. Each row consists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

lower or upper half is used, reduce the transaction size to 32 bytes.

- 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 Minimising the number of memory transactions

The key to minimising 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 Figure 3. Each thread is computing a single realisation. The storage addresses of the molecular populations and propensity functions are grouped by species.

This scheme works perfectly for the propensity functions and 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 minimising 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

TABLE 1

Number of reactions and molecular species. The right most column gives the maximum feasible block sizes using the fat thread approach.

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)
<i>lac</i> -operon	16	11	128 (4)
Fully connected network	30	6	96 (3)

steps. First, we assess the optimum block size using a small example system. Second, the fat and thin thread approach to parallel stochastic simulation using GPUs is compared to the sequential implementation on a CPU³. The results are given as a speedup (ratio of sequential run-time on a CPU to parallel run-time on the GPU) as well as in run-time in milliseconds.

5.1 Example systems

We use five example reaction systems of varying size. They are described in detail in the Supplemental Online Material. The first example system is a decay-dimerisation model used by Gillespie [16], the second is the Oregonator, a simulation of the oscillating Belousov-Zhabotinskii reaction discovered in 1951 [17]. 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 [18]. The fourth is a simplified simulation of the *lac*-operon presented by [19]. 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 minimising 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

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

TABLE 2

Determining the optimal block size in latency hiding. Run-time per realisation 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.

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

the time evolution of the Oregonator system. The run-time per realisation across the increasing block sizes is given in Table 2 below. 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 sizes 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 sizes [20].

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.

Since the actual run-time 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 speed-up for the example systems used is given in Figure 4 below. While the fat thread approach outperforms the thin thread one at the dimerisation-decay and Oregonator system, at the *Lac*-operon, the thin thread approach performs better at the circadian cycle and the diffusion system. The fat thread

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 [13].

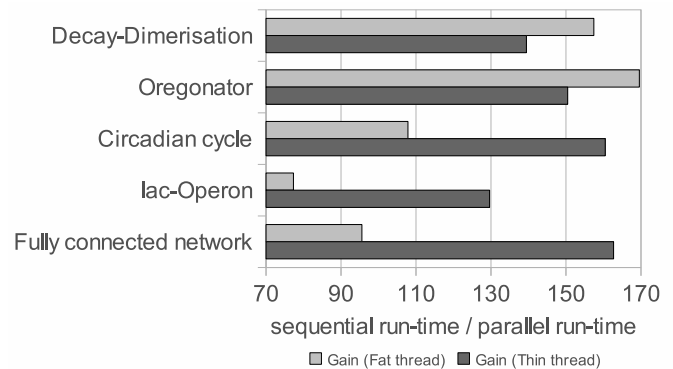


Fig. 4. Comparing the gain, ratio of the sequential run-time of the CPU implementation and the parallel GPU implementation, of the fat and thin thread approach.

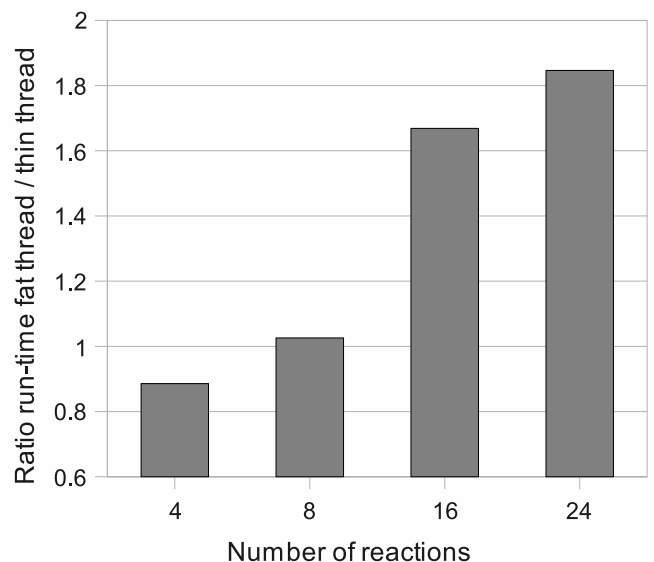


Fig. 5. Run-time ratio of the fat thread and thin thread approach. A ratio smaller than 1 means the fat thread approach is faster, greater than 1 the thin thread is faster. To create larger reaction systems with similar characteristics, multiple (1, 2, 4, 6) dimerisation-decay reaction systems are concatenated to create reaction systems with 4, 8, 16, and 24 reactions, respectively⁷. Even for 8 reactions, the thin thread approach is slightly faster than the fat thread. For 16 and 24 reactions, the thin thread approach is clearly beneficial.

approach seems to be favourable 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-dimerisation systems at once. The details of these models are given in the Supplemental Online Material. The run-time ratio of the fat thread and thin thread approach is shown in Figure 5. A ratio of smaller than 1 means the fat thread approach is faster.

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 five biologically meaningful and one artificially constructed example systems. The fat threading follows the recommended approach using the fast on-chip shared memory to minimise 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.

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.

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 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* [21]. 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 USED

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 at 2.16 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.

7. Details of the example reaction systems used are given in the supplemental material.

ACKNOWLEDGMENTS

GK was supported by the Systems biology Doctoral Training Centre (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 UK Engineering and Physical Sciences Research Council under research grant EP/G00210X/.

PM was partially supported by a Royal Wolfson Merit Award.

REFERENCES

- [1] 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 Journal on Applied Mathematics*, vol. 70, no. 3, pp. 984–1016, 2009.
- [2] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter, *Molecular Biology of the Cell*. Garland Science, New York, 2007.
- [3] F. Holstege, E. Jennings, J. Wyrick, T. Lee, C. Hengartner, M. Green, T. Golub, E. Landry, and R. Young, "Dissecting the regulatory circuitry of a eukaryotic genome." *Cell*, vol. 95, no. 5, pp. 717–728, Nov 1998.
- [4] J. Hasty, D. McMillen, F. Isaacs, and J. Collins, "Computational studies of gene regulatory networks: In numero molecular biology," *Nature Review Genetics*, vol. 2, no. 4, pp. 268–279, April 2001.
- [5] T. Tian and K. Burrage, "Stochastic models for regulatory networks of the genetic toggle switch," *Proceedings of the National Academy of Sciences*, vol. 103, no. 22, pp. 8372–8377, 2006.
- [6] D. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *Journal of Computational physics*, vol. 22, pp. 403–434, 1976.
- [7] —, "Exact stochastic simulation of coupled chemical reactions," *Journal of Physical Chemistry*, vol. 81, no. 25, pp. 2340–2361, 1977.
- [8] M. Gibson and J. Bruck, "Efficient exact stochastic simulation of chemical systems with many species and many channels," *Journal of Physical Chemistry A*, vol. 104, pp. 1876–1889, 2000.
- [9] H. Li and L. Petzold, "Logarithmic direct method for discrete stochastic simulation of chemically reacting systems," Department of Computer Science, University of California Santa Barbara, Tech. Rep., 2006. [Online]. Available: <http://www.cs.ucsb.edu/cse/publications.php>
- [10] Y. Cao, H. Li, and L. Petzold, "Efficient formulation of the stochastic simulation algorithm for chemically reacting systems," *Journal of Chemical Physics*, vol. 121, no. 9, pp. 4059–4067, 2004.
- [11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Computer Society Hot Chips*, no. 19, pp. 39–45, March/April 2008.
- [12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, no. 2, pp. 40–53, March/April 2008.
- [13] *Nvidia CUDA Programming Guide, Version 2.1*, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, California, 12 2008.
- [14] P. Maciol and K. Banas, "Testing tesla architecture for scientific computing: the performance of matrix-vector product," *Proceedings of the International Muticonference on Computer Science and Information Technology*, vol. 3, pp. 285–291, 2008.
- [15] *Nvidia Compute PTX: Parallel Thread Execution, ISA Version 1.4*, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, California, 3 2009.

- [16] D. Gillespie, "Approximate accelerated stochastic simulation of chemically reacting systems," *Journal of Chemical Physics*, vol. 115, no. 4, pp. 1716–1733, 2001.
- [17] J. Murray, *Mathematical Biology 1: An Introduction*, 3rd ed., ser. Interdisciplinary applied sciences. Springer Verlag, 2002, vol. 1.
- [18] J. Vilar, H. Kueh, N. Barkai, and S. Leibler, "Mechanisms of noise-resistance in genetic oscillators," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 9, pp. 5988–5992, 2002.
- [19] D. Wilkinson, *Stochastic Modelling for Systems Biology*. Chapman & Hall/CRC, 2006.
- [20] *NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3*, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, California, July 2009.
- [21] *NVIDIAs Next Generation CUDA Compute Architecture: Fermi*, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, California, 2009, v 1.1.



Philip K. Maini received his DPhil in mathematics from Oxford University. He is currently Professor of Mathematical Biology and Director of the Centre for Mathematical Biology, Mathematical Institute, Oxford. He is also part of the Oxford Centre for Integrative Systems Biology, Department of Biochemistry. His interests are mainly in deterministic models of cell- and tissue-level interactions to signalling cues with applications in developmental biology, cancer growth and wound healing.



Guido Klingbeil received a diplom in computer engineering from the Technical University of Hamburg-Harburg. He is currently 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. He is currently a Research Fellow in the Oxford Centre for Collaborative Applied Mathematics, Mathematical Institute, University of Oxford. His research interests include mathematical biology, multiscale modelling, partial differential equations, stochastic simulation algorithms, gene regulatory networks, mathematical fluid dynamics and applications of mathematics in medicine.



Mike Giles received a BA in mathematics from the University of Cambridge, and a SM and PhD in aeronautical engineering from MIT. He is 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.