GPU Code Generation for ODE-Based Applications with Phased Shared-Data Access Patterns

ANDREI HAGIESCU, National University of Singapore
BING LIU, Carnegie Mellon University
R. RAMANATHAN and SUCHEENDRA K. PALANIAPPAN, National University of Singapore
ZHENG CUI, Advanced Digital Science Centre, Singapore
BIPASA CHATTOPADHYAY, University of North Carolina
P. S. THIAGARAJAN and WENG-FAI WONG, National University of Singapore

We present a novel code generation scheme for GPUs. Its key feature is the platform-aware generation of a heterogeneous pool of threads. This exposes more data-sharing opportunities among the concurrent threads and reduces the memory requirements that would otherwise exceed the capacity of the on-chip memory. Instead of the conventional strategy of focusing on exposing as much parallelism as possible, our scheme leverages on the phased nature of memory access patterns found in many applications that exhibit massive parallelism. We demonstrate the effectiveness of our code generation strategy on a computational systems biology application. This application consists of computing a Dynamic Bayesian Network (DBN) approximation of the dynamics of signalling pathways described as a system of Ordinary Differential Equations (ODEs). The approximation algorithm involves (i) sampling many (of the order of a few million) times from the set of initial states, (ii) generating trajectories through numerical integration, and (iii) storing the statistical properties of this set of trajectories in Conditional Probability Tables (CPTs) of a DBN via a prespecified discretization of the time and value domains. The trajectories can be computed in parallel. However, the intermediate data needed for computing them, as well as the entries for the CPTs, are too large to be stored locally. Our experiments show that the proposed code generation scheme scales well, achieving significant performance improvements on three realistic signalling pathways models. These results suggest how our scheme could be extended to deal with other applications involving systems of ODEs.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Patterns (pipeline)*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: GPU, memory hierarchy, code generation

ACM Reference Format:
Hagiescu, A., Liu, B., Ramanathan, R., Palaniappan, S. K., Cui, Z., Chattopadhyay, B., Thiagarajan, P. S., and Wong, W.-F. 2013. GPU code generation for ODE-based applications with phased shared-data access patterns. ACM Trans. Architec. Code Optim. 10, 4, Article 55 (December 2013), 19 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555311

1. INTRODUCTION

General-purpose computing using Graphics Processing Units (GPUs) has been making rapid advances [Owens et al. 2007; Nickolls and Dally 2010]. High-level programming languages and APIs such as CUDA [NVIDIA 2012] and OpenCL [Khronos 2012] are

Authors' addresses: W.-F. Wong, National University of Singapore - Computer Science, 117417, Singapore; email: wongwf@nus.edu.sg.

DOI: http://dx.doi.org/10.1145/2555289.2555311

now available to ease the task of programming GPUs by shielding the programmer from the architectural details. The programming paradigm offered by these APIs consists of computational kernels that read in an input stream of data and produce an output data stream. These kernels are realized by launching a large pool of parallel threads, organized in scheduling units called *warps*, consisting of threads executing in lock-step. The input data streams are typically stored in the off-chip Global Memory (GM), which is accessible to all threads running on the GPU. The significant latency of the GM is masked through multiplexing and pipelining. Specifically, the GPU hardware multiplexes the warps on a number of pipelined Streaming Multiprocessors (SMs). Each SM consists of a large set of registers, a number of execution cores, and a scratch pad memory that is shared by all warps allocated to the SM.

Warps are grouped into thread blocks that are scheduled by the GPU hardware. Only warps within the same thread block are guaranteed to execute concurrently and may exchange data using a small on-chip *local memory*. Therefore, applications need to handle the opposing requirements that all threads exchanging data must map to a thread block (a few warps) and, at the same time, ensure that code encapsulated within each warp executes in lock-step to achieve high performance.

With this as motivation, this article describes a novel and efficient code generation scheme for GPU architectures. To concretely highlight the main features of our scheme, we shall describe its application to a systems biology application that exhibits the key mix of potential massive parallelism among threads but with phased data exchanges between the threads. To bring this out, we begin with a brief outline of this application. A standard model of the dynamics of a biochemical network is a system of Ordinary Differential Equations (ODEs). Our application consists of approximating such a system of ODEs as a Dynamic Bayesian Network (DBN). Typical models of biochemical networks involve a large number of ODEs (one corresponding to each molecular species involved in the network), and the DBN approximation considerably eases the task of analyzing the behaviour of the biochemical network. The approximation algorithm involves (i) sampling many (of the order of a few million) times from a set of initial states, (ii) generating *trajectories* through numerical integration, and (iii) storing the statistical properties of this set of trajectories in the Conditional Probability Tables (CPTs) of a DBN via a prespecified discretization of the time and value domains.

The large number of trajectories and the high dimensionality of the system makes the problem computationally intensive. A conventional approach would be to map the computation of each trajectory to a GPU thread. A large number of such threads can be executed in parallel in lock-step, as required for execution on GPU architectures. However, the cumulative size of the intermediate data used by these concurrent threads would be too large to be stored in the SMs' registers or SM memory. This is due to the coupling between the ODEs, where the computation of the next value of a variable along a trajectory has to make use of the current values of all variables appearing in the equation describing the dynamics of this variable. Thus, for high-dimensional systems, the GM has to be used to store the intermediate data. However, this leads to a vicious cycle in which more parallel threads have to be launched to hide the memory latency that in turn creates more accesses to the GM, leading to memory bandwidth saturation and eventually to performance degradation.

An important observation is that the computation of a single trajectory itself admits significant parallelism. For each variable x_i, the "next" value can be computed independently as a function of the current value of other variables. Our proposed implementation strategy exploits this fine-grained parallelism and determines the best way to distribute the equations among several parallel threads. We also introduce the required synchronization to propagate the new value of a variable to all equations in which it appears. As a result, there is a significant reduction in memory usage because the intermediate data of each trajectory is shared by all threads associated with each trajectory. Following this strategy, we achieved a higher GPU utilization (more parallel threads) while processing fewer trajectories at a time. In fact, our solution carefully constrains the number of parallel trajectories so that the size of the intermediate data matches the amount of available SM memory.

It is important to note that since the ODEs are not identical, the resulting threads will be heterogeneous. Hence, a significant feature of our code generation scheme is its ability to handle a heterogeneous pool of compute threads. We achieve this by dividing this pool into homogeneous groups and then scheduling the groups into warps that match the lock-step execution constraints of the GPU. In addition, a special class of threads is instantiated to handle the infrequent transfers to GM. Our implementation scheme also includes a performance model that accounts for the throughput of the underlying architecture (in our case, the NVIDIA Tesla 2.0 architecture [Glaskowsky 2009]).

In summary, this article makes the following contributions: (i) an execution strategy based on fine-grained parallelism and heterogeneous threads tuned to the Tesla GPU architecture, (ii) a code generation method which refactors applications to match the proposed heterogeneous execution scheme, and (iii) a solution that reduces the memory requirements of the GPU kernels.

We believe that this scheme is applicable to many other problems having a similar structure. A variety of such problems arise in chemical, economical, engineering, physical, environmental, and life sciences, involving the modeling, simulation, and analysis based on a system of ODEs. All of these settings would benefit from a scheme through which information is extracted from a trajectory as it is being generated while a large number of trajectories is generated in parallel. For instance, it has been demonstrated [Palaniappan et al. 2013] how the powerful analysis technique called Statistical Model Checking (SMC) [Younes and Simmons 2006] can be used to study the dynamics of a system of ODEs. This application again involves generating a large number of trajectories and performing a statistical analysis of the resulting trajectories. A key step in the analysis is deciding whether each trajectory has the dynamical property (usually reachability) specified by a temporal logic formula. Doing this *on the fly* using the so-called tableau-based model checking procedure will entail extracting and binning information from the trajectories. Hence, our code generation scheme is also a good candidate for obtaining scalable and efficient GPU implementations for this class of applications.

1.1. Related Work

A variety of previous schemes have been devised to improve the performance of GPU implementations. Of particular relevance to our work are the data prefetching and memory latency hiding techniques [Owens et al. 2008; Silberstein et al. 2008; Ye et al. 2010; Bodin and Bihan 2009; Wolfe 2010]. However, these techniques are not applicable in our context, as they rely on a large ratio between computation and the size of the dataset prefetched into the on-chip SM memory.

Another problem often affecting performance is the relationship between the kernel geometry and the layout of the data to be processed. In general, the selection of the number of parallel threads is correlated with data placement, and identifying a solution is not trivial [Ryoo et al. 2008]. In contrast, our framework goes beyond traditional data tiling [Aho et al. 2006; Bastoul 2004] and introduces an additional level of flexibility in thread scheduling that allows for changes in the kernel computation without affecting data placement. Our approach extracts fine-grained parallel code from the biopathway model and distributes it across a number of concurrent threads [Chen et al. 2011]. Other GPU code generation schemes utilize heterogeneous collaborative threads [Hormati et al. 2011; Hagiescu et al. 2011]. However, these schemes have only been directed to segregate slow GM accesses into separate threads, thereby freeing dedicated computation threads from such accesses. This work goes beyond these schemes and introduces multiple classes of dedicated compute threads.

As part of the code generation scheme, we need a customized performance model. Although other performance models exist [Hong and Kim 2009; Zhang and Owens 2011], their role is to evaluate the overall application performance, which is mainly driven by the memory access patterns and GPU occupancy. In contrary, the model in our article focuses on the number of cycles spent by threads in the GPU compute pipeline and is used for thread load balancing by providing a comparison between compute-only code segments.

In a related article, we have presented the broad outlines of our code generation strategy [Liu et al. 2012]. However, the focus of that article was on the computational systems biology issues, including a novel probabilistic verification scheme. In contrast, in this article, we discuss in detail the GPU architecture-specific code generation, memory allocation, and scheduling issues that must be addressed to handle massive parallelism accompanied by substantial data sharing.

1.2. Plan of the Article

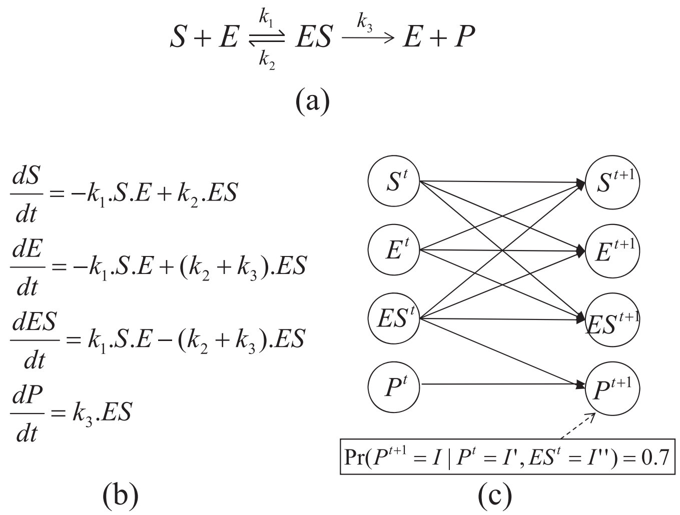
In Section 2, we describe the systems biology background of the application and describe the procedure for constructing the DBN approximation of a system of ODEs. In Section 3, we present our code generation scheme. In Section 4, we present the experimental results for a set of eight models designed to test various features of our code generation scheme. In particular, we present the results for three realistic models: the EGF-NGF pathway [Brown et al. 2004], the segmentation clock pathway [Goldbeter and Pourquie 2008], and the thrombin-dependent-MLC-phosphorylation pathway [Maedo et al. 2006]. In the final section, we summarize and discuss possible extensions of our work.

2. THE PROBLEM DOMAIN

Modelling and analysis of biopathways dynamics is a core activity in systems biology [Kitano 2002]. A *biopathway* is a network of biochemical reactions that is often modelled as a system of ODEs [Aldridge et al. 2006]. The equations describe the biochemical reactions with the variables representing the concentration levels of molecular species. The system often does not admit closed-form solutions. Instead, one has to numerically generate *trajectories* to study the dynamics. In addition, reaction rate constant parameters are often unknown, and the experimental data used for model training and testing are often population based and have limited precision. Consequently, Monte Carlo methods are employed to ensure that sufficiently many values from the distribution of model parameters are being sampled. As a result, basic tasks such as parameter estimation, model validation, and sensitivity analysis require a large number of numerical simulations.

The method developed in Liu et al. [2009] will be our focus here. It computes a large set of trajectories, induced by the ODE dynamics, starting from a distribution of the initial states. The key idea is to exploit the dependencies (as well as independencies) in the pathway structure to compactly encode these trajectories as a time-variant DBN [Murphy 2002]. This DBN is viewed as an approximation of the ODE dynamics, and analysis tasks can be performed on this simpler model using standard Bayesian inference techniques [Koller and Friedman 2009]. The applicability of this method has been demonstrated by previous work [Liu et al. 2011b].

Figure 1 shows a biopathway, its associated ODE model, and its DBN approximation. The ODEs are of the form $\frac{dx_i}{dt} = f_i(\mathbf{x}, \mathbf{p})$ for each molecular species x_i, with f_i describing the kinetics of the reactions that produce and consume molecular species $x_i \in \mathbf{x}$, where



$$S + E \underset{k_2}{\overset{k_1}{\rightleftharpoons}} ES \xrightarrow{k_3} E + P$$

(a)

$$\frac{dS}{dt} = -k_1.S.E + k_2.ES$$

$$\frac{dE}{dt} = -k_1.S.E + (k_2 + k_3).ES$$

$$\frac{dES}{dt} = k_1.S.E - (k_2 + k_3).ES$$

$$\frac{dP}{dt} = k_3.ES$$

(b)

(c)

Fig. 1. (a) The enzyme catalytic reaction network. (b) The ODE model. (c) The DBN approximation for two successive time slices.

x are all the molecular species taking part in these reactions and **p** are the parameters (rate constants) associated with these reactions [Liu et al. 2009].

The initial values of the variables x_i are assumed to be of certain distributions. The unknown parameters p_j are assumed to be uniformly distributed over their ranges. We then sample the initial states of the system many times and compute trajectories using numerical integration starting from each of the sampled initial states [Liu et al. 2009].

The DBN consists of an acyclic directed graph where the nodes are grouped into layers, with each layer representing a time point. The nodes in layer $t + 1$ will be connected to those nodes in the layer t on which they depend, as shown in Figure 1(c). The set of connections does not change as t ranges within a trajectory from 1 to T. Each node in the DBN is associated with one of the random variables and has a CPT associated with it. The CPT specifies the local probabilistic dynamics of the DBN and reflects the evolution of a variable x_i observed at time point t over multiple trajectories. In our setting, this value is x_i^t. To compute the CPT entries, we partition the range of values for each variable (unknown parameter) into a set of intervals $\mathbf{I}_i$. The CPT entries record the probability that the value of x_i^t falls in each interval $I_i^k \in \mathbf{I}_i$ at time t, in relationship with the intervals where the variables (unknown parameters) on which it depends were found at the previous time point. The probability is calculated through simple counting in what we shall call a *binning* step of the algorithm.

The GPU computation steps are shown in Figure 2. We assume that the system state used to construct the DBN is sampled each Δt, for a finite number of points, $\{0, 1, \ldots, T\}$. Since trajectories are generated through numerical integration, to ensure numerical accuracy, each interval $[0, \Delta t]$ is uniformly subdivided into r subintervals for a suitable choice of r. We compute an updated value of the variables every $\tau = \frac{\Delta t}{r}$. Each ODE updates one of the variables, and in doing so uses the previous value of other variables. Each variable may appear in multiple equations, leading to a large amount of read sharing. To ensure consistency, all variables are updated together in an atomic transaction. We use a fourth-order Runge-Kutta integration algorithm to compute the next value of a variable for each timestep. Overall, each trajectory is numerically simulated for $r \cdot T$ steps.

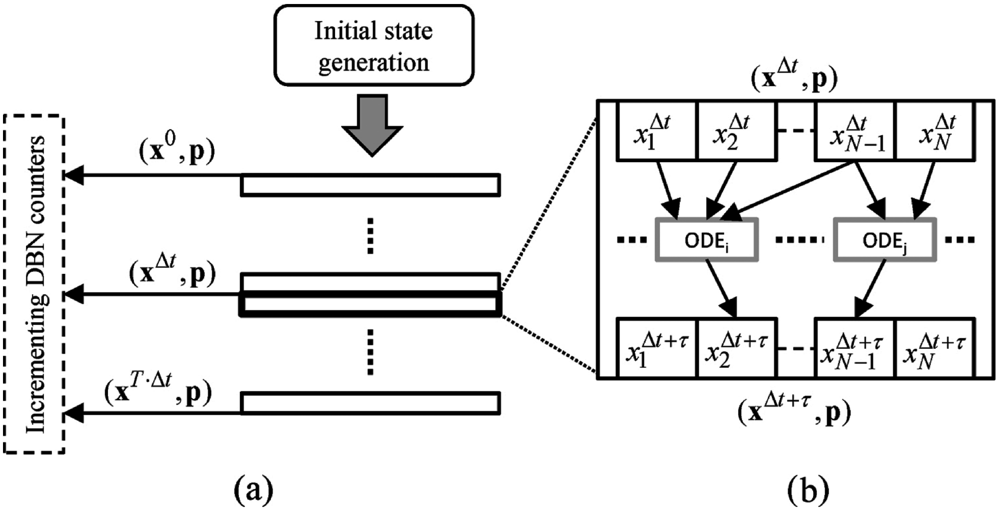

Initial state generation
Incrementing DBN counters
$(\mathbf{x}^0,\mathbf{p})$
$(\mathbf{x}^{\Delta t},\mathbf{p})$
$(\mathbf{x}^{T\cdot\Delta t},\mathbf{p})$
$x_1^{\Delta t}$
$x_2^{\Delta t}$
$x_{N-1}^{\Delta t}$
$x_N^{\Delta t}$
ODE$_i$
ODE$_j$
$x_1^{\Delta t+\tau}$
$x_2^{\Delta t+\tau}$
$x_{N-1}^{\Delta t+\tau}$
$x_N^{\Delta t+\tau}$
$(\mathbf{x}^{\Delta t+\tau},\mathbf{p})$
(a)
(b)

Fig. 2. Phases in computing a trajectory. (a) Computation steps. (b) Runge-Kutta integration step.

Finally, the current values of the variables sampled at each of the time points $\{0, \Delta t, \ldots, T \cdot \Delta t\}$ are used to count how many of the trajectories hit a particular interval of values for each variable at that time point. These counts are then used to derive the entries in the CPTs of the DBN. As described earlier, there will be one CPT for each variable and each time point of interest. Each CPT will have $|\mathbf{I}_i|^{m_i+1}$ entries where I_i is the set of intervals associated to the variable and m_i is the number of variables a node x_i depends on. It is important to note that m_i will almost always be much smaller than the number of variables in the system.

3. MAPPING TO A GPU ARCHITECTURE

In this section, we describe the *automatic code generation* scheme that we have developed to map our application onto the GPU architecture. We first explain the generic scheme and then give details of its application to our systems biology problem. We start with a review of the GPU architecture and its impact on performance. Essentially, in a GPU:

(1) A large number of threads must be instantiated to obtain the maximum performance.
(2) There is a *warp*-level affinity for lock-step execution (a more relaxed form of SIMD).
(3) The amount of fast SM memory is limited.

It is the programmer's responsibility to expose parallelism in the application through the programming model in order to satisfy the first requirement. However, this will often conflict with the other requirements. With a large number of threads instantiated, the SM memory quota for each thread is a small number of bytes, and often the user has to identify opportunities for data sharing across threads to achieve efficient execution.

Serialization occurs, with the accompanying penalty, when there is control flow divergence within a warp. Therefore, the programming model calls for as little divergence as possible. This leads, in general, to a particular type of data processing that we call *homogeneous* computing, in which loops are unrolled and distributed over the entire thread grid. The following example describes this approach:

```
for ($i = 0$; $i < N_i$; $i$++) {
   for ($j = 0$; $j < N_j$; $j$++)
       code$_0$($i$,$j$);
           ...
       code$_{C-1}$($i$,$j$);
  }
```

The conventional GPU pseudocode will be:

```
for ($i = 0$; $i < N_i/\Theta$; $i$++)
   for ($j = 0$; $j < N_j/\Pi$; $j$++)
       parallel for ($\theta \leq \Theta$, $\pi \leq \Pi$) {
           code$_0$($i \cdot \Theta + \theta$, $j \cdot \Pi + \pi$);
               ...
           code$_{C-1}$($i \cdot \Theta + \theta$, $j \cdot \Pi + \pi$);
       }
```

In the previous code, N_i, N_j, Π, and Θ allow for arbitrary geometric shapes of the loop structure. The loop body is formed of C code segments. We will discuss the significance of this in our context in Section 3.2. The execution does not diverge, as all threads execute the same *homogeneous* computation for different datasets. It is important to ensure that the product $\Pi \cdot \Theta$ is high enough so that enough GPU threads are utilized. However, we need to consider other details of the GPU architecture. In particular, it is desirable that all data accessed during the parallel execution is located in the SM memory. We use the notation $\mathbf{M}$(code) to denote the memory requirement of code segment code. The maximum SM memory requirement of all parallel threads is then:

$$\max_{\substack{i<N_i/\Theta \\ j<N_j/\Pi}} \left| \bigcup_{\substack{\pi<\Pi,\theta<\Theta \\ c<C}} \left(\mathbf{M}(\text{code}_c(i \cdot \Theta + \theta, j \cdot \Pi + \pi))\right) \right|$$

In contrast, our code generation scheme is built on the insight that there is no penalty when threads in *different* warps diverge—as long as those in the same warp do not. Therefore, the key concept behind our code generation scheme is to look for fine-grained parallelism, within the loop body, and identify independent code segments that can be executed in parallel. Assuming that code$_0$, code$_1$, ... are independent, we place these segments in threads that belong to different warps in a *heterogeneous* computing model. Obviously, some amount of loop-level parallelism is still necessary to fill each warp with similar threads. Therefore, we choose to partially unroll only the outer loop:

```
for ($i = 0$; $i < N_i/\Theta$; $i$++)
   for ($j = 0$; $j < N_j$; $j$++)
       parallel for ($c \leq C$, $\theta \leq \Theta$) {
           code$_c$($i \cdot \Theta + \theta$, $j$);
       }
```

In this implementation, the number of threads is determined by $C \cdot \Theta$. In addition, to ensure that threads with similar control flow can be grouped in each warp of size, W_{size}, $\exists w \in \mathbb{N}, \Theta = w \cdot W_{\text{size}}$. We will later discuss the penalty for not satisfying this constraint. More importantly, the memory requirement of this implementation is:

$$\max_{\substack{i<N_i/\Theta \\ j<N_j}} \left| \bigcup_{\substack{\theta<\Theta \\ c<C}} \big(\mathbf{M}(\text{code}_c(i \cdot \Theta + \theta, j)\big) \right|$$

When compared to the homogeneous approach, the main advantage is derived from the lower amount of unrolling, which for certain applications may significantly decrease the memory requirements. If

$$\forall c, i_1, i_2, j_1 \neq j_2, \mathbf{M}(\text{code}_c(i_1, j_1)) \bigcap \mathbf{M}(\text{code}_c(i_2, j_2)) = \emptyset,$$

the required memory can decrease Π times while ensuring sufficient GPU occupancy. It is important to observe that the proposed code transformations do not affect the inner loop. This allows us to optimize even for the case where the iterations of the inner loop are not independent.

The pseudocode associated with the previous heterogeneous computation may be easily written in CUDA C as follows:

```
...
#define Θ $w \cdot W_{\text{size}}$
dim3 blockDim(Θ, C, 1);
kernel<<<gridDim, blockDim>>>(...);
  ...
__global__ void kernel(...) {
   int idx = threadIdx.x;
   for ($i = 0$; $i < N_i/\Theta$; $i$++)
      for ($j = 0$; $j < N_j$; $j$++)
          switch (threadIdx.y) {
             case 0: $\text{code}_0(i \cdot \Theta + \text{idx}, j)$; break;
             ...
             case C-1: $\text{code}_{C-1}(i \cdot \Theta + \text{idx}, j)$; break;
          }
}
```

Although it may appear that the `switch` statement introduces divergence resulting in the serialization of the different code segments, we ensure that (based on the thread id) the branching decision is identical for all the threads in each warp. In this example, this is so because $\Theta = w \cdot W_{\text{size}}$. Each warp can execute independent control flow and will skip through the `switch` statement to its associated code segment with minimal penalty.

We have described a scheme where data resides only in SM memory. However, the input and output of the application must be transferred from / to GM. Due to the long latency of GM, any such transfer suffers a large delay, during which the requesting thread (and its associated warp) must stall. By default, the GPU architecture replaces the stalled warp with another available warp. This approach relies on a high enough computation to memory transfer ratio such that alternative warps are available. If the GM transfers are scattered across all warps, the memory access delay will impact all threads. Instead, our code generation schemes prefetches from the GM within a few warps, handling these transfers in parallel and without interfering with the execution of the other warps [Hagiescu et al. 2011].

3.1. Achieving Balance

Our scheme attains the optimal GPU performance only if the amount of computation in each code segment is balanced such that the GPU pipeline is always full. Otherwise, some of the warps will finish processing early, whereas the remaining warps are not capable of ensuring sufficient GPU occupancy to fill the GPU pipeline. Our code generation scheme distributes fine-grained computation blocks extracted from the loop body among code segments located in different warps and obtains feedback regarding the quality of the computational balance and pipeline occupancy by analyzing the PTX assembly generated by the CUDA compiler.

The loop body consists of a list of instructions corresponding to an integration step for each variable. We can cluster these instructions into groups that exhibit only inter-iteration dependencies, because each integration step is independent of the others. These clusters are $(eq_0, eq_1, \ldots eq_n)$. We initially compile the entire loop body as a single thread and analyze its PTX assembly, obtaining the number of PTX instructions in each cluster i as $\mathrm{PTX(eq_i)}$. We use this information to determine how to place these code clusters across threads in order to balance the pipeline occupancy.

Given the throughput stated in the documentation of the GPU for each arithmetic operation, we model the number of cycles required to issue each PTX instruction in the GPU pipeline. The pipeline has a latency of 22 cycles, and multiple warps are multiplexed by the GPU hardware to issue continuously instructions on the pipeline. The Tesla 2.0 architecture supports the simultaneous execution of two half-warps, each of them utilizing half the number of compute cores available. For single-precision floating point instructions, the pipeline occupancy analysis is equivalent to the assumption that a single full warp is processed at a time. By compiling the code for "fast math," we also ensure that the PTX instructions in the compiled code directly match the operations supported by the architecture.

Using the earlier assumptions, we model, for example, floating-point `add` instructions across one warp as being issued in a single cycle, whereas `div` instructions are issued within eight cycles. We use the notation $\mathrm{issue}(div) = 8$. We also model the timing of the `ld` and `st` instructions that access SM memory. With proper data alignment, all SM memory banks are utilized. Because of the inherent architectural two-way conflict on SM memory banks, a memory access is issued every two cycles. The pipeline occupancy represents the fraction of the execution cycles where a new operation is issued. For a code segment of size PTX(code) instructions, this is calculated as:

$$o(\mathrm{code}) = \frac{\sum\limits_{i \in \mathrm{PTX(code)}} \mathrm{issue}(i)}{22 \cdot |\mathrm{PTX(code)}|}$$

Our code generation scheme has two objectives: (i) to ensure that all C warps have a balanced *number* of instructions and (ii) to ensure that the pipeline occupancy achieved by summing the occupancy induced by each warp exceeds (but is close to) 1. Because the GPU has a fixed latency pipeline and we avoid GM accesses, when the occupancy is 1 or below, the number of instructions corresponds to the latency of their execution. Additional threads beyond an occupancy of 1 will queue for execution and lead only to additional register pressure and subsequent performance degradation.

We estimate how many threads C are required to occupy the pipeline by analyzing the average occupancy across all code segments: $C = \lceil \frac{1}{o(eq_0 \cup eq_1 \cup \ldots \cup eq_n)} \rceil$. This is a reasonable approximation because distributing the code over C threads increases the occupancy C times. We chose which clusters to allocate to each code segment code_i such that $|\mathrm{PTX(code_i)}| = \frac{|PTX(eq_0 \cup eq_1 \cup \ldots \cup eq_n)|}{C}$. We employ a greedy allocation, where instruction clusters are allocated in sequence to each code segment.

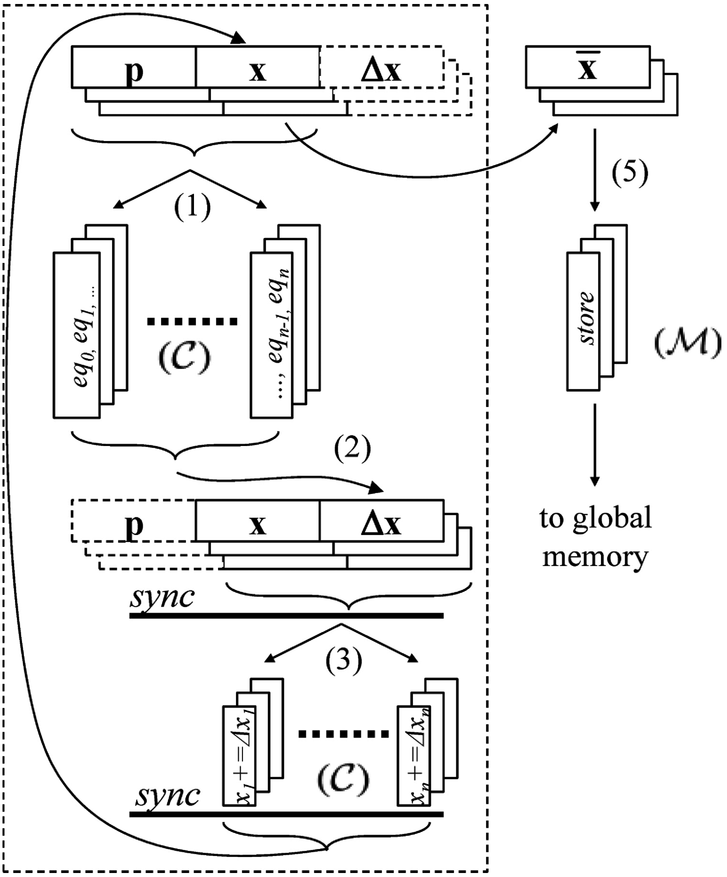


Fig. 3. Data movement during trajectory generation and binning.

3.2. Code Generation for the Systems Biology Application

We have implemented the automatic code generation scheme described previously for the application presented in Section 2. The user specifies the model as equations using a simple specification language. From these equations, the CUDA code that will perform the simulation is generated.

The memory requirements of the code generating each trajectory are large. Storing the data in SM memory will prevent GM bandwidth saturation. This is because the dependencies between the variables in the system of ODEs require the entire front of variables belonging to each trajectory to be computed together. Specifically, we recall that each variable x has an associated equation in the ODE system. For each time interval, the value of x at the end of the interval is determined by applying a Runge-Kutta numerical integration using the current value of x and the current values of other variables (and parameters) appearing in the ODE for x.

The computational pattern for each trajectory matches the loop body of the heterogeneous computation scheme introduced previously. We show the data movement during one computation step in Figure 3. The equations are valid instruction clusters and are distributed into compute threads $\mathcal{C}$ that will collaborate to generate a single trajectory. This entails sharing of the variables and parameters within a group. The number of threads in such a group is C. These threads read the current values of the variables

($\mathbf{x}$) and parameters ($\mathbf{p}$) from memory (step (1)). The $\Delta\mathbf{x}$ changes during a timestep are computed in parallel and are stored back to memory (step (2)). The vector of variables $\mathbf{x}$ is then updated (step (3)). This process is applied iteratively for each timestep (of duration $\tau = \frac{\Delta t}{r}$). The Θ trajectories are computed in parallel to satisfy the lock-step requirements of the GPU architecture. Each trajectory requires $N_j = T \cdot r$ integration steps. The trajectory computation process is repeated for all N trajectories, hence $N_i = \frac{N}{\Theta}$.

In the binning steps of the application, the number of times the threads hit the various intervals of values of the variables are counted. To do this, the vector $\mathbf{x}$ is replicated as $\bar{\mathbf{x}}$ (step (4)). The binning process executes in parallel, during the next Δt iteration, using the memory access threads ($\mathcal{M}$), which will store the results in a large table located in GM (step (5)). This will ensure that the numerical integration can continue during the binning process, which has long latency memory operations.

As a departure from the code generation scheme described previously, we require additional synchronization among the $\mathcal{C}$ threads after each integration step. These $\mathcal{C}$ threads belong to several warps; hence, they are scheduled independently, and their execution may not be synchronized. Synchronization is achieved by a partial synchronization primitive available since the Tesla 2.0 architecture. The `bar.sync` PTX instruction allows for an explicit number of threads to be waited for at the barrier. The number of threads may be smaller than the total number of threads executing on the GPU. Once all of the $\mathcal{C}$ threads arrive at the barrier, they proceed to the next integration step.

The vectors $\mathbf{x}$, $\mathbf{p}$, $\Delta\mathbf{x}$, and $\bar{\mathbf{x}}$ together form the *workset* of the trajectory. All threads of each SM have access to the dedicated SM memory, which is similar to a scratchpad [Li et al. 2009]. To ensure that enough parallel threads can be instantiated, the computation of each trajectory is unfolded onto the C threads. This enables a reduction of the number of trajectories being processed concurrently. In this way, the total memory footprint, consisting of the worksets of all Θ trajectories being computed in a SM, can be kept within the limit of the available SM memory.

We have also carefully selected the placement of the worksets in the SM memory in order to prevent bank conflicts. The number of banks is generally 2^{α}, but in our device is 2^4. If the size of the worksets is an even number, the stride between worksetsis increased by 1 by increasing the workset size to the next odd number. This is sufficient to ensure that the hardware will coalesce the accesses to the SM memory, because 2^{α} consecutive worksets (accessed by adjacent threads in the same warp) cannot hit the same bank [NVIDIA 2012]. Duplicating the data is not necessary, as the variables can be accessed by all threads involved in the generation of the same trajectory.

The GPU architecture requires all threads belonging to a warp to have matching control flow in order to achieve the highest performance. Otherwise, the threads' execution will be serialized. Accordingly, we organize the $\mathcal{C}$ threads belonging to each trajectory so that threads executed together in the same warp process the *same* subset of model equations from *different* trajectories. Given a warp size of 32 threads, this eliminates the control flow divergence in each warp if $\exists w, \Theta = w \cdot 32$.

However, Θ is constrained by the SM memory capacity to $\Theta = \frac{\mathrm{SM}_{\mathrm{size}}}{\mathrm{workset}_{\mathrm{size}}}$. Therefore, it is not always feasible to instantiate a sufficient number of parallel trajectories in order to completely fill each warp with $\mathcal{C}$ threads having similar control flow. In this case, we have chosen to fill the rest of the warp with threads that belong to the next equation group. This ensures the best utilization of the GPU register pool. However, to maintain warp boundaries, we decrease the number of trajectories to the immediately lower number that matches the equation $\exists \delta \geq -log_2(W_{\mathrm{size}}), \Theta = W_{\mathrm{size}} \cdot 2^{\delta}$. If a warp contains multiple sets of threads, their execution is serialized, and we can model the

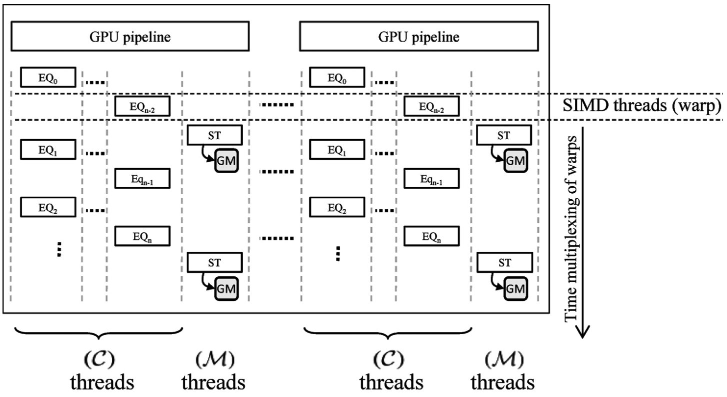


Fig. 4. Concurrent execution of trajectories inside an SM.

combined warp as if several warps were executed. Ideally, the total number of issue cycles for all $\mathcal{C}$ warps in the CUDA code has to match the pipeline length to ensure full GPU occupancy. In contrast, when $\Theta > W_{\text{size}}$, multiple warps may encapsulate the same code segment, and we determine C as follows:

$$C = \frac{22 \cdot |PTX(eq_0 \cup \ldots)|}{\sum_{i \in PTX(eq_0,\ldots)} issue(i) \cdot \lceil \Theta / W_{\text{size}} \rceil}$$

The overall orchestration of the application on each SM is shown in Figure 4. Instructions belonging to $\mathcal{C}$ warps are multiplexed onto the GPU pipelines. All of the GPU pipelines execute in lock-step. $\mathcal{M}$ threads are scheduled from time to time to transfer data to the GM. The specialized warps accessing GM are subject to delays of up to 400 cycles. $\mathcal{M}$ threads are grouped together into specialized memory access warps such that they will not interfere with the $\mathcal{C}$ threads' executions. The same orchestration is replicated on all SMs of the GPU. This can be easily implemented by computing a fraction of the total number of trajectories on each SM.

For each trajectory, we generate the initial states using a Mersenne twister algorithm based on the MT 19937 random number generator [Matsumoto and Nishimura 1998], running in each of the $\mathcal{C}$ threads. This algorithm utilizes a large table stored in the GM. Considering that this initialization step is done only once during the generation of a trajectory, the overhead due to storing this table in GM is minimal.

The repetitive Runge-Kutta numerical integration process is at the heart of the trajectory simulation algorithm. We used a fourth-order Runge-Kutta algorithm that requires each equation to be applied four times as part of the integration step. The code generation scheme produces the corresponding code for each equation and passes it to the CUDA compiler. The PTX assembly is analyzed using the previously described model to extract timing information for each equation. Our algorithm distributes the equations so that the corresponding timing is balanced among the $\mathcal{C}$ threads. Because we utilize a small number of threads, register pressure is low and there are no spills to local memory, hence avoiding any additional delay.

4. RESULTS

We have implemented the scheme described in Section 3 and have used it to generate CUDA code that was compiled for NVIDIA Tesla 2.0 ('Fermi') platforms using the CUDA 4.0 runtime. The target GPU is a S2050 at 1.15GHz with 2GB of memory. To evaluate the performance of our GPU-based implementation, we utilized eight different models.

Table I. Models Used

Model Abbrev.	Description
EGF-NGF	EGF-NGF signalling pathway [Brown et al. 2004]
m201	Segmentation clock network [Goldbeter and Pourquie 2008]
m88	Thrombin-dependent MLC phosphorylation [Maedo et al. 2006]
synthetic1	A variety of synthetic equations
synthetic2	A large number of synthetic equations
synthetic3	A set of short, balanced synthetic equations
synthetic4	A small number of unbalanced synthetic equations
synthetic5	A set of long, balanced synthetic equations

Table II. Characteristics of the Models

Model	$\|\mathbf{x}\|$	$\|\mathbf{p}\|$	Δt	T	r	N	Avg. ops	+/−	×	÷
EGF-NGF	32	20	60	100	100	10^6	7.4	87	106	44
m201	22	40	300	100	500	10^6	11.9	67	91	33
m88	105	164	2	100	2×10^4	3×10^4	13	419	942	2
synthetic1	200	86	300	100	500	10^4	6.7	650	680	0
synthetic2	205	163	2	10	2×10^4	3×10^3	15	939	2,132	2
synthetic3	100	66	300	100	500	10^4	8.4	385	450	0
synthetic4	40	35	60	100	1000	5×10^4	5.9	125	101	8
synthetic5	100	97	300	100	500	10^4	14.2	601	821	0

These models included three realistic pathway models [Liu et al. 2011a], as well as five synthetic models that tested various features of our scheme (Table II).

NVIDIA GPUs support both single-precision and double-precision floating point types. However, the computational throughput for double precision is known to be less than half of single precision, with significant overhead for divisions in the generation of GPUs that we used. For our application, single-precision computation suffices. We chose the number of trajectories such that the resulting DBN approximation was of sufficient good quality and that runtimes were sufficiently long.

Figure 6 shows the reaction network for the `EGF-NGF` model. The values for the parameters of this model are known. However, to mimic realistic biopathways models, we have set a subset of the parameters as “unknown” in each model and constructed the DBN approximation accordingly. This considerably increases the computational demands placed on the DBN construction algorithm. The same was done to two other pathway models, namely `m88` and `m201`. In addition, a set of synthetic benchmarks were created to verify the scalability of our approach, as well as to stress test the “corner cases.” `synthetic1` is a disparate set of 200 equations with varying size. `synthetic2` consists of 205 equations, with many long equations that factor into a limited set of terms. `synthetic3` is a set of 100 short equations of similar size. In contrast, `synthetic4` consists of 40 unbalanced equations. Finally, `synthetic5` contains 100 equations, 90% of which is a balanced subset of long equations.

For each model, we listed the number of variables ($|\mathbf{x}|$), the number of unknown parameters ($|\mathbf{p}|$), the simulation time between DBN nodes (Δt), the number of node levels (T), the number of integration steps (r), and the total number of trajectories (N). We also listed the average number of operators within each model equation, as well as the distribution of each operator’s type. For all models, the range of each variable and unknown parameter was discretized into five intervals of equal size. A smaller number of trajectories were computed for the larger models to keep the execution times within reasonable limits.

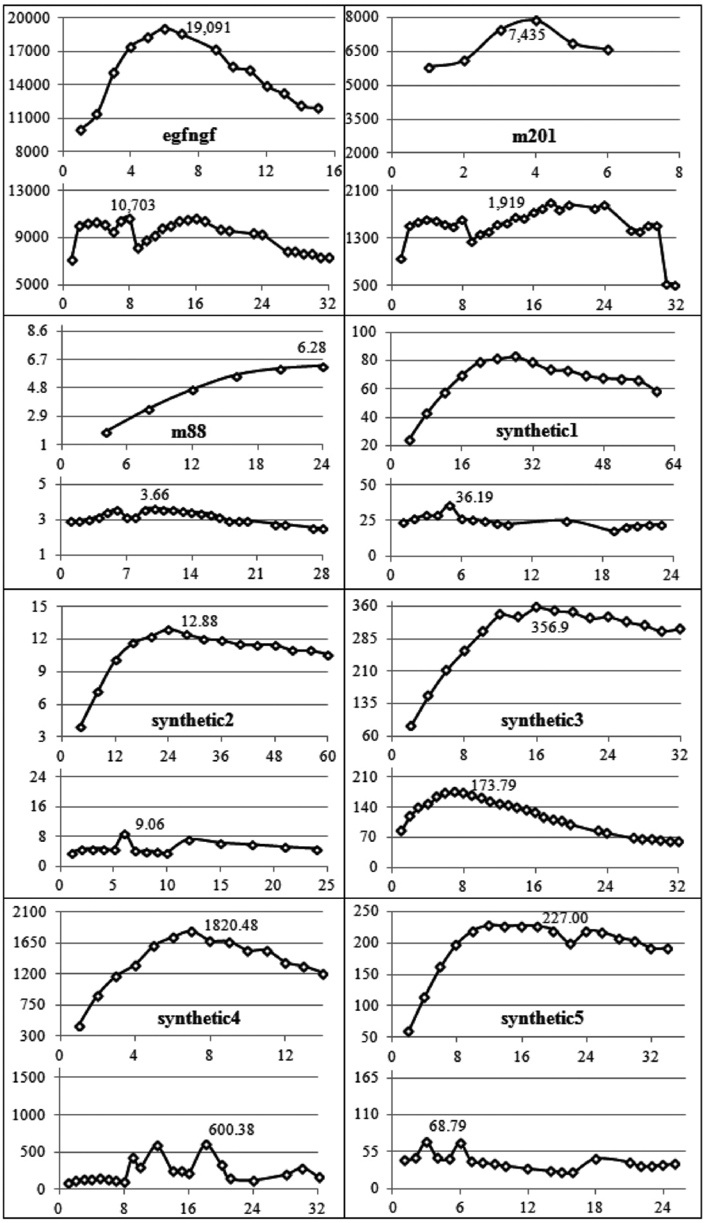


Fig. 5. Performance characterization of the proposed scheme (upper graph for each benchmark) versus the homogeneous approach (lower graph) on Tesla 2.0 S2050. The x-axis represents C for the proposed scheme and the number of warps for the homogeneous approach, whereas the y-axis represents how many trajectories are computed every second.

The following evaluation strategy was used. We implemented the target application using both a homogeneous computation approach (where the workset is stored in GM, as the datasets do not fit SM memory) and the proposed heterogeneous approach. To emphasize the efficiency of the proposed flow, we characterized a broad design space by varying the number of threads of both homogeneous and heterogeneous

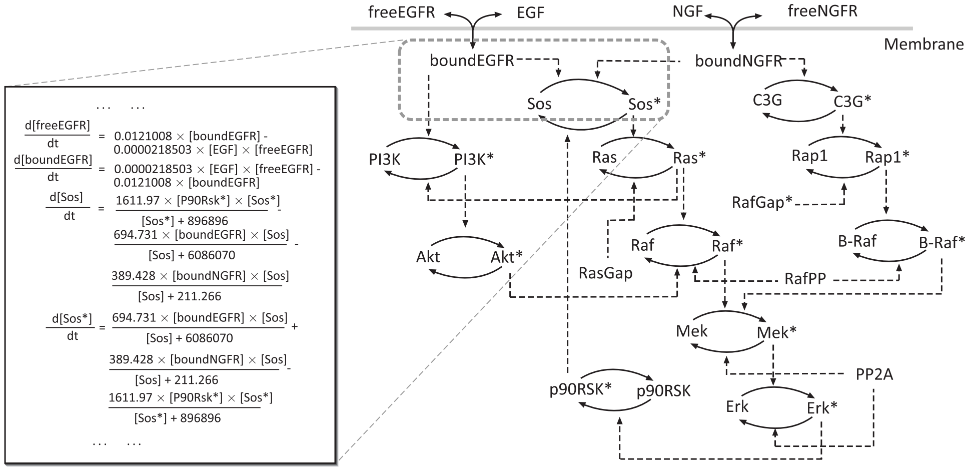

freeEGFR
EGF
NGF
freeNGFR
Membrane
boundEGFR
boundNGFR
Sos
Sos*
C3G
C3G*
PI3K
PI3K*
Ras
Ras*
Rap1
Rap1*
RafGap*
Akt
Akt*
Raf
Raf*
B-Raf
B-Raf*
RasGap
RafPP
Mek
Mek*
PP2A
p90RSK*
p90RSK
Erk
Erk*
... ...
d[freeEGFR]/dt = 0.0121008 × [boundEGFR] - 0.0000218503 × [EGF] × [freeEGFR]
d[boundEGFR]/dt = 0.0000218503 × [EGF] × [freeEGFR] - 0.0121008 × [boundEGFR]
d[Sos]/dt = 1611.97 × [P90Rsk*] × [Sos*] / [Sos*] + 896896 - 694.731 × [boundEGFR] × [Sos] / [Sos] + 6086070 - 389.428 × [boundNGFR] × [Sos] / [Sos] + 211.266
d[Sos*]/dt = 694.731 × [boundEGFR] × [Sos] / [Sos] + 6086070 + 389.428 × [boundNGFR] × [Sos] / [Sos] + 211.266 - 1611.97 × [P90Rsk*] × [Sos*] / [Sos*] + 896896
... ...

Fig. 6. The reaction network diagram of the EGF-NGF pathway [Brown et al. 2004]

approach schemes, producing a large spectrum of kernel geometries. For the homogeneous implementation, we varied the thread block size, whereas for the heterogeneous implementation, we varied C, the number of $\mathcal{C}$ threads collaborating to generate a trajectory.

In addition to an overall performance evaluation of our framework, we will show the contribution of each component of the framework: the proposed heterogeneous thread execution scheme, the usage of shared memory, the separation of the GM accesses, and the load balancing.

In Figure 5, we show a comparative design-space exploration for the eight models we considered. We overlap the performance of both the homogeneous and the heterogeneous implementations on the same graphs. For homogeneous graphs, the x-axis represents the total number of warps in a thread block, whereas for heterogeneous graphs, it represents the number of $\mathcal{C}$ threads. The performance is measured in trajectories per second. The performance of the homogeneous implementation ends up always being lower, as it is bound by the GPU memory bandwidth. In addition, this performance cannot be trivially estimated, as it depends on many factors such as the GM bandwidth, GPU occupancy, and register pressure. Large performance variations are observed when the number of threads (warps) is varied.

In contrast, our proposed heterogeneous scheme has a predictable as well as significantly higher performance. For all benchmarks, performance increases steadily as more parallel code segments are created.

A single code segment, containing all ODEs (the first point in each heterogeneous graph in Figure 5), is equivalent to a homogeneous implementation where the data have been moved from GM to SM memory. The performance is low, as having a single code segment prevents data reuse across threads, leading to a higher ratio of data/thread. Only Θ threads can be run concurrently due to the limited size of the SM memory. This indicates that simply changing the location of the workset without refactoring the computation pattern does not provide any performance boost.

Initially, splitting the code leads to a nearly linear performance increase with respect to the number of resulting code segments C. This shows that the resulting code segments can be well balanced and that the required synchronization has negligible overhead. Eventually, as more code segments are added, the performance reaches a plateau. This corresponds to reaching full pipeline occupancy. From this point onward,

Table III. Performance of the Proposed Approach Compared to a Homogeneous GPU Implementation

Model	Setup			Runtime(s)		
	$\|\mathbf{x}\|$	N	$T \cdot r$	Homogeneous	Our scheme	Speedup
EGF-NGF	32	3×10^6	10^4	280.29	157.14	1.8×
m201	16	3×10^6	5×10^4	1,563.6	403.5	3.9×
m88	105	3×10^4	2×10^6	8190	4596	1.8×
synthetic1	200	10^4	5×10^4	276.3	121.67	2.3×
synthetic2	205	3×10^3	2×10^5	331	233	1.4×
synthetic3	100	10^4	5×10^4	57.54	28.01	2.1×
synthetic4	40	5×10^4	10×10^4	83.28	27.47	3.0×
synthetic5	100	10^4	5×10^4	145.36	44.05	3.3×

Table IV. Execution Configuration, Register, and SM Usage of the Models

Model	Block Threads	Reg. Used	SM Used (KB)
EGF-NGF	64 × 6	32	36.25 (75.5%)
m201	128 × 4	49	46.50 (97.0%)
m88	16 × 23	63	37.69 (78.5%)
synthetic1	8 × 28	33	28.97 (60.4%)
synthetic2	8 × 24	63	28.22 (58.8%)
synthetic3	16 × 16	39	32.94 (68.6%)
synthetic4	64 × 7	35	47.75 (99.5%)
synthetic5	16 × 16	39	42.56 (88.7%)

Table V. Automatically Chosen Configurations

Model	Θ	C	C_{optimal}	Penalty
EGF-NGF	$64 = 32 \times 2$	6	7	2.8%
m201	$128 = 32 \times 2^2$	4	4	0%
m88	$16 = 32 \times 2^{-1}$	22	24	1.7%
synthetic1	$8 = 32 \times 2^{-2}$	28	28	0%
synthetic2	$8 = 32 \times 2^{-2}$	24	24	0%
synthetic3	$16 = 32 \times 2^{-1}$	16	16	0%
synthetic4	$64 = 32 \times 2^1$	7	7	0%
synthetic5	$16 = 32 \times 2^{-1}$	16	18	1.0%

there is no benefit from creating additional code segments. Instead, the performance experiences a small degradation due to the granularity of the load balancing and also due to the additional register pressure. For the smaller benchmarks, performance degrades significantly more when too many C threads are created. In this case, the load balancer handles fewer equations, and their granularity prevents adequate balancing.

As part of our code generation scheme, we have proposed an automated configuration selection algorithm. We have validated the automatically determined values against the design space in Figure 5. The value for C produced by this algorithm, related to the value determined through design space exploration, shows that we incur a penalty of less than 3%. Table V captures the actual penalty and the computation steps taken in automatically selecting a configuration. The other parameter determining aconfiguration, Θ (the number of trajectories computed in parallel on each SM), can be easily derived from the workset size and rounded according to the scheme's rules.

We have included the overall results in Table III. The speedup achieved by the heterogeneous scheme indicates the suitability of the proposed approach. Table IV includes additional details about the number of threads in each thread block of the kernel, the number of registers used, and SM memory occupancy.

Table VI. Benefit of Heterogeneous Groups and Specialized Memory Threads

	Heterogeneous Approach		Specialized Memory Access Threads	
Model	$\mathcal{C}$	Speedup	$\mathcal{C}+\mathcal{M}$	Additional speedup
EGF-NGF	7	1.65×	$6+1$	1.09×
m201	5	3.45×	$4+1$	1.13×
m88	24	1.78×	$22+2$	1.01×
synthetic1	32	2.07×	$28+4$	1.11×
synthetic2	28	1.27×	$24+4$	1.10×
synthetic3	18	2.04×	$16+2$	1.03×
synthetic4	8	2.70×	$7+1$	1.11×
synthetic5	18	3.03×	$16+2$	1.09×

Table VII. Contribution of Thread Balancing to the Overall Speedup

Model	Naïve Balancing Speedup	Additional Speedup
EGF-NGF	1.62×	1.11×
m201	3.80×	1.03×
m88	1.50×	1.20×
synthetic1	1.53×	1.50×
synthetic2	1.20×	1.20×
synthetic3	1.70×	1.25×
synthetic4	2.31×	1.30×
synthetic5	3.00×	1.11×
Average	2.08×	1.21×

In addition to SM memory usage and heterogeneous thread usage, we evaluate the contribution of the other individual components of our framework. We evaluate the impact of the memory thread specialization by comparing the speedup achieved by the models (i) when heterogeneous threads are used but computation and memory accesses are mixed within the same threads and (ii) when compute and memory threads are distinct. Table VI underlines the benefit of this separation. The additional speedup introduced by specialized memory access threads reaches up to 13%. The specialized threads provide better opportunity for data coalescing. In addition, because computation threads never stall, the $\mathcal{C}$ threads can more quickly reuse the small amount of SM memory.

We also provide experimental data that indicate the benefit of the proposed thread balancer. We compare to a naive load balancing, where the same *number* of equations is allocated to each compute thread. Unless the equations have the same complexity, some of the threads finish processing earlier, and the GPU is not fully utilized, leading to a significant performance degradation, as shown in Table VII. The proposed thread balancer can improve performance up to 1.5× for the set of benchmarks explored.

The results indicate that the heterogeneous scheme alone provides most of the performance improvement. Using heterogeneous threads not only exposes more parallel computation but also enables data reuse in SM memory; hence, the GM traffic is significantly reduced, whereas the level of parallelism increases.

5. CONCLUSION

In this article, we have proposed a novel GPU code generation strategy and have demonstrated its usefulness in the context of a computational systems biology application. This application consists of approximating the dynamics of a high-dimensional system of ODEs with many unknown rate constants as a DBN.

The main challenge was in reorganizing the computation in order to utilize the significant amount of fine-grained parallelism that exists due to the phased coupling of the variables at each integration step. In addition to the coarse-grained parallelism between the computations of different trajectories. A naïve GPU implementation that merely exploits the coarse-grained parallelism does not scale well for large models, because the intermediate data will be too large to be held in SM memory.

Another key idea driving our strategy is load balancing heterogeneous threads via a static timing model. Our experiments show that this is vital for good performance. Our overall method works well for high-dimensional systems of ODEs. Our results show that the performance of a conventional GPU implementation is both deficient and difficult to predict.

Our work shows that intricate knowledge of the GPU is required to obtain high performance. The good news is that this intricate knowledge can be built into an automated code generation scheme. Consequently, in our experiments, we were able to achieve up to 3.9× improvement over a conventional GPU implementation across a range of case studies despite using an automated code generation scheme.

We have used a specific systems biology application that has the generic mix of features that illustrate the advantages of our scheme. Our future goal is to further extend this scheme to other applications in which the system model is a set of ODEs and the analysis involves binning information during the process of generating trajectories. Such applications can be identified in a variety of fields, such as economics; engineering; and physical, biological, and environmental sciences. In particular, the application developed in Palaniappan et al. [2013], in which the powerful SMC technique is used to analyze a system of ODEs, is a good fit, and our current effort is on adapting our code generation scheme to this application in particular and to develop a GPU-based SMC procedure for distributed stochastic dynamical systems in general.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. 2006. *Compilers: Principles, Techniques and Tools,* 2 ed. Addison Wesley.

Aldridge, B. B., Burke, J. M., Lauffenburger, D. A., and Sorger, P. K. 2006. Physicochemical modelling of cell signalling pathways. *Nature Cell Biology 8*, 1195–1203.

Bastoul, C. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04).* IEEE Computer Society, Washington, DC, 7–16.

Bodin, F. and Bihan, S. 2009. Heterogeneous multicore parallel programming for graphics processing units. *Sci. Program. 17,* 4, 325–336.

Brown, K. S., Hill, C. C., Calero, G. A., Lee, K. H., Sethna, J. P., and Cerione, R. A. 2004. The statistical mechanics of complex signaling networks: nerve growth factor signaling. *Physical Biology 1*, 184–195.

Chen, L., Villa, O., and Gao, G. R. 2011. Exploring fine-grained task-based execution on multi-GPU systems. In *Proceedings of the 2011 CLUSTER Conference*. 386–394.

Glaskowsky, P. N. 2009. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Retrived December 2, 2013 from http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU.pdf.

Goldbeter, A. and Pourquie, O. 2008. Modeling the segmentation clock as a network of coupled oscillations in the Notch, Wnt and FGF signaling pathways. *Journal of Theoretical Biology 252*, 574–585.

Hagiescu, A., Huynh, H. P., Wong, W. F., and Goh, R. S. M. 2011. Automated architecture-aware mapping of streaming applications onto GPUs. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11).*

Hong, S. and Kim, H. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09).* ACM, New York, NY, 152–163.

HORMATI, A., SAMADI, M., WOH, M., MUDGE, T. N., AND MAHLKE, S. A. 2011. Sponge: Portable stream programming on graphics engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 381–392.

KHRONOS. 2012. Khronos OpenCL. http://www.khronos.org/opencl/.

KITANO, H. 2002. Computational Systems Biology. *Nature 420*, 206–210.

KOLLER, D. AND FRIEDMAN, N. 2009. *Probabilistic Graphical Models: Principles and Techniques (Adaptive Computation and Machine Learning)*. MIT Press.

LI, L., FENG, H., AND XUE, J. 2009. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim. 6,* 3, 1–17.

LIU, B., HAGIESCU, A., PALANIAPPAN, S. K., CHATTOPADHYAY, B., CUI, Z., WONG, W.-F., AND THIAGARAJAN, P. S. 2012. Approximate probabilistic analysis of biopathway dynamics. *Bioinformatics 28,* 11, 1508–1516.

LIU, B., HSU, D., AND THIAGARAJAN, P. S. 2011a. Probabilistic approximations of ODEs based bio-pathway dynamics. *Theor. Comput. Sci. 412,* 21, 2188–2206.

LIU, B., THIAGARAJAN, P. S., AND HSU, D. 2009. Probabilistic approximations of signaling pathway dynamics. In *CMSB*, P. Degano and R. Gorrieri, Eds., Lecture Notes in Computer Science Series, vol. 5688. Springer, 251–265.

LIU, B., ZHANG, J., TAN, P. Y., HSU, D., BLOM, A. M., LEONG, B., SETHI, S., HO, B., DING, J. L., AND THIAGARAJAN, P. S. 2011b. A computational and experimental study of the regulatory mechanisms of the complement system. *PLoS Computational Biology 7,* 1, e1001059.

MAEDO, A., OZAKI, Y., SIVAKUMARAN, S., AKIYAMA, T., URAKUBO, H., USAMI, A., SATO, M., KAIBUCHI, K., AND KURODA, S. 2006. Ca^{2+}-independent phospholipase A2-dependent sustained Rho-kinase activation exhibits all-or-none response. *Genes to Cells 11*, 1071–1083.

MATSUMOTO, M. AND NISHIMURA, T. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul. 8*, 3–30.

MURPHY, K. P. 2002. *Dynamic Bayesian Networks: Representation, Inference and Learning*. Ph.D. thesis, University of California, Berkeley.

NICKOLLS, J. AND DALLY, W. J. 2010. The GPU computing era. *IEEE Micro 30*, 56–69.

NVIDIA. 2012. NVIDIA CUDA.

OWENS, J. D., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J. E., AND PHILLIPS, J. C. 2008. GPU computing. *Proceedings of the IEEE 96,* 5, 879–899.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26,* 1, 80–113.

PALANIAPPAN, S. K., GYORI, B. M., LIU, B., HSU, D., AND THIAGARAJAN, P. S. 2013. Statistical model checking based calibration and analysis of bio-pathway models. In *Proceedings of the 11th International Conference on Computational Systems Biology (CMSB'13)*.

RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND HWU, W.-M. W. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using cuda. In *PPoPP'08*. 73–82.

SILBERSTEIN, M., SCHUSTER, A., GEIGER, D., PATNEY, A., AND OWENS, J. D. 2008. Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*. ACM, New York, NY, 309–318.

WOLFE, M. 2010. Implementing the PGI accelerator model. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'10)*. 43–50.

YE, X., FAN, D., LIN, W., YUAN, N., AND IENNE, P. 2010. High performance comparison-based sorting algorithm on many-core GPUs. In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposuim (IPDPS'10)*. 1–10.

YOUNES, H. L. AND SIMMONS, R. G. 2006. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation 204,* 9, 1368–1409.

ZHANG, Y. AND OWENS, J. D. 2011. A quantitative performance analysis model for GPU architectures. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society, Washington, DC, 382–393.

Received August 2012; revised November 2013; accepted November 2013