Source-to-Source Optimization of CUDA C for GPU Accelerated Cardiac Cell Modeling

Fred V. Lionetti[1], Andrew D. McCulloch[2], and Scott B. Baden[1]

[1] Departments of Computer Science and Engineering
[2] Departments of Bioengineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0404 USA
http://www.cse.ucsd.edu/groups/hpcl/scg

Abstract. Large and complex systems of ordinary differential equations (ODEs) arise in diverse areas of science and engineering, and pose special challenges on a streaming processor owing to the large amount of state they manipulate. We describe a set of domain-specific source transformations on CUDA C that improved performance by $\times 6.7$ on a system of ODEs arising in cardiac electrophysiology running on the nVidia GTX-295, without requiring expert knowledge of the GPU. Our transformations should apply to a wide range of *reaction-diffusion systems.*

Keywords: Automatic code generation, source-to-source transformations, optimization, GPU, CUDA, cardiac cell modeling, ODEs.

1 Introduction

Large and complex systems of ordinary differential equations (ODEs) arise in diverse areas of science and engineering. For example, the systems of ODEs that are used in simulations of cardiac electrophysiology model the ionic fluxes across the cell membrane that cause changes in the transmembrane potential in cardiac ventricular cells. Numerical simulations of cellular membranes are useful for both basic science and increasingly for clinical diagnostic and therapeutic applications. While a traditional cluster based on CPUs could deliver the required performance, a GPU offers a more compact and appropriate solution in a clinical setting [1].

State-of-the-art cardiac electrophysiology simulators manage a large amount of state, and this poses special challenges on a streaming processor. The user is focused on the domain science and is constantly experimenting with new cell models. They have neither the expertise nor the background to implement the required CUDA optimizations. We have developed a system that enables the domain scientist to author high-level cellular models and invoke a software tool to automatically generate optimized GPU code incorporating the knowledge of an expert programmer.

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 38–49, 2010.

The workflow for our approach begins with a user program written in a scripting language (in our case *Python*) that specifies a mathematical representation of physical processes taking place in the cell. We translate the input into naïve CUDA C, which is subsequently optimized using source-to-source transformations. The resultant code is then compiled with *nvcc,* the CUDA C compiler.

While *nvcc* effectively manages registers in our simulations, it cannot manage global references as effectively, nor the on-chip "shared" memory. There is simply too much state, and many expressions that have complicated dependence patterns involving that state. Our optimizations transform device memory accesses to cheaper on-chip memory accesses, where the CUDA C compiler failed to do so. They eliminate redundant computations resulting from the higher-level model translation step and introduce thread parallelism to manage dual GPUs on the same graphics card. We explore various optimizations, and found that the best combination is to partition the generated kernels into more manageable pieces and to build an off-line (static) software cache residing in shared memory.

The ultimate measure of performance is the amount of time needed to simulate a single heartbeat. Our optimized implementation on an nVidia GTX-295 GPU reduced the computational bottleneck significantly, lowering the time to simulate a single heartbeat from 4.5 *hours* on a 48 core Opteron cluster (under MPI) to 12.7 *minutes* on a desktop workstation equipped with a $500 GPU accelerator. To deliver the equivalent performance, our Opteron cluster would require at least 1020 cores. We have effectively miniaturized the computational testbed, transforming the simulation technology so that it has become feasible in a clinical setting. All this can be accomplished without requiring knowledge of the GPU or the low-level programming details required to realize high performance.

Cardiac electrophysiology models belong to the important class of *reaction-diffusion* systems where one or more reactants, e.g. chemical species, in a dynamically changing system can diffuse in space. In our case the reactions are the ionic current ODEs and the "diffusion" of transmembrane potential represents current flowing from cell to cell in the tissue. Thus, our techniques can be applied to other important problems of computational science and engineering, including chemical reactors and climate systems together with biochemical reaction systems (pathways) occurring within spatially constrained environments in cells or tissues in the presence of spatial gradients of chemical species. Our approach does not accelerate the solution of the spatial coupling equations arising within the model, which requires the solution of large sparse linear systems of equations.

2 Contribution and Related Work

This paper makes two contributions. First, it describes a set of source-to-source optimizations that improve the use of fast, on-chip, GPU memory in large complex kernels arising in systems of ODEs. Second, it demonstrates a testbed for cardiac electrophysiology simulation that is appropriate in a clinical setting.

Electrophysiology modeling differs from other time dependent problems (such as the classic N-body problem) in that the cost of solving the Partial Differential

Equation (PDE) is relatively inexpensive compared with that of solving the resultant systems of ODEs. Sato et al. report on electrophysiology modeling on the GPU [2] using a model that uses 8 state variables. This is a simple model by modern standards. Liu et al. [3] implemented a source-to-source translator to automatically optimize a kernel against program inputs, and uses statistical learning techniques to search the optimization space. They demonstrated that the most effective optimizations are data dependent, which we also observed for our cell models.

Silberstein et al. [4] demonstrate a *user-managed cache* that, like ours, resides in shared memory. The impact of shared memory caching was more dramatic than ours, indicating that their kernels were likely far more memory bound. Their caching strategy is on-line. We use an optimal off-line strategy (Belady's MIN algorithm). They do not explore the possibility copying global data to local registers that we found to yield similar performance improvements to software caching. Eichenberger et al. [5] discuss source-to-source optimizations to improve on-chip local storage usage on the Cell Broadband Engine. They implemented an on-line software managed cache together with function partitioning [6]. Our partitioning strategy is coarser grained; we split large functions into pieces.

3 The nVidia GTX-295

Our GPU testbed is nVidia's GTX-295 processor, comprising 240 scalar cores organized as 30 *streaming multiprocessors (SM)*. We programmed with CUDA [7]. A CUDA program executes sequences of *kernels*, functions that run as a set of virtualized scalar threads, hierarchically organized into *thread blocks*. The hardware dynamically assigns each thread block to a single *streaming multiprocessor (SM)*, dividing each block into *warps* of 32 threads. All threads in a warp execute as a SIMD instruction; branching is costly and to be avoided.

Each SM contains two programmable on-chip memories vital to realizing high performance: 64KB of *registers*, split among all simultaneously running threads, and 16KB of *shared memory*, providing fast communication between threads in the same block, and split among all thread blocks simultaneously running on the SM. Arithmetic instructions run more slowly out of shared memory. For example, Volkov and Demmel noted a 33% slowdown for multiply-add (MAD) when the source was in shared memory [8]. The GPU has slow *device memory*, with roughly 150x the latency compared to registers or shared memory. We used two of the four logical partitions of this memory: *global memory* and *local memory*. Global memory is available to all threads and persists across kernel invocations. Local memory is a backing store for threads, when the compiler runs out of registers and is forced to *spill* them.

SMs hide device memory latency by maintaining multiple active warps (in practice, multiple thread blocks). The *occupancy* is the fraction of maximum possible active warps (supported by the hardware) that are in fact active. Since active warps tie up registers it is generally important to minimize register usage in order to increase occupancy, and similarly with shared memory.

4 Translation

In order that our simulator is usable by a domain scientist, it must deliver acceptable performance without requiring that the model developer be an expert GPU programmer. The model is a high level description and is written in Python, as shown in Fig. 1. Using this higher-level description, our translator automatically generates highly tuned CUDA kernels, and invokes a framework to integrate the kernels into a runnable simulator. This approach separates concerns between model development and performance tuning, and thus simplifies model and infrastructure development.

4.1 Code Generation

Our cell model simulations run under *Continuity 6*, a problem-solving environment that uses the finite element method for heart modeling. *Continuity* is distributed free for research by NBCR, an NIH funded resource, which enables biomedical research by developing computational technologies.[1]

```
 1. #st_start st_dur st_mag
 2. #ug vg
 3. vmax = 1.0, vrest = 0.0
 4. cm = 1.0, a = 0.130, b = 0.0130, c1 = 0.260, c2 = 0.10, d = 1.0
 5. st_end = st_start + st_dur
 6. heavi = HeavisideEq(t-st_start)*HeavisideEq(st_end-t)
 7. i_st = st_mag*heavi
 8. ug_n = (ug-vrest)/(vmax-vrest)
 9. dug_dt = (ug_n*(ug_n-a)*(1.0-ug_n)*c1-c2*vg*ug_n)*(vmax-vrest)+i_stim*/cm
10. dvg_dt = b*ug_n - b*d*vg
```

Fig. 1. A cell model. Line 1 declares the length of the simulation, line 2, the state variables. Each variable is evaluated at many co-location points, as described in §5.

The workflow begins with the scientist authoring a cell model in Python, specifying all state variables, parameters, and the equations comprising the model. Our translator takes over from here. The first step is to generate C code from the cell model (we use *sympy* [9], a Python library for symbolic mathematics). With C code in hand, the translator next builds an abstract syntax tree (AST) (we use *pycparser* [10]) and it consults the AST to construct dependence information for each state variable. Using this information and the C expressions, the translator next generates a CUDA kernel that composes the cell model code with an ODE solver provided by *Continuity*. For the most complex cell model we used–the Flaim model [11]– there are 87 state variables. For the Flaim model the translator generates a 4917 line CUDA kernel with 2278 assignments that reference an average of 2.8 variables per statement. In the next stage the optimizer applies various transformations (optimizations) specific to the GPU. Finally, the translator calls *nvcc* to produce a shared library accessible to *Continuity*.

[1] *Continuity* has been downloaded thousands of times and has over 700 registered users worldwide.

4.2 Optimizations

We explored a variety of optimizations that can be composed in various ways. As we detail these optimizations below, we will do so in the context of the Flaim model mentioned above. This model is typical of modern cell models that have a large amount of state. Our kernels perform few branches and access to global device memory is either read-mostly or write-only, and we can take advantage of such program behavior to optimize performance.

We begin with a "naïve" implementation, which is the result of composing our cell model with an ODE solver that carries out numerical integration. Composing the two modules introduces redundant calculations.

Optimization 1 automatically removes redundant calculations by searching and modifying the AST. With the help of *decuda*, a third party tool to disassemble the binary produced by nVidia's *nvcc* compiler[2], we determined that O1 reduces the number of assembly operations by a factor of 3.0, from 19,747 to 6,685. We use this variant as the baseline for performance. The next optimizations (2L and 2S) are mutually exclusive.

Optimization 2L copies the global variables (state variables) into local (automatic) variables. We observed that with simpler cell models with fewer variables, the compiler could allocate all the variables to fast registers. With larger cell models, such as the Flaim model, the compiler is forced to spill to *local memory* which, though physically located in device memory, suffers the same high latency penalty as global memory. Nevertheless, our tests indicate that the CUDA compiler manages variables in local memory more effectively than variables in global memory by "caching" frequently used references in registers and thus device memory accesses did not increase dramatically. We suspect that this optimization benefited from the fact that local variables are simpler to reference than global memory (e.g. they do not require a thread index), freeing up registers to reduce local memory references still further.

Optimization 2S implements a cache using shared memory. Because there are virtually no branches in our code–a characteristic of cellular models– we use an offline algorithm to implement replacement. Our replacement strategy is based on Belady's MIN page swapping algorithm [12], which is optimal. When the cache is full, we evict the variable whose next reference is the furthest in the future, because all other cached variables will be needed sooner.

Optimization 3: Kernel partitioning. This optimization is combined with one of 2L or 2S. As previously mentioned, our model is complex, and this challenges the CUDA compiler's ability to manage the registers. Our strategy is to split the large monolithic kernels into several smaller ones, which the CUDA compiler can then manage on its own, i.e. loop fission [13]. We partition kernels at the boundary between state variable updates, corresponding to the ODEs specified in the higher-level user input. The best kernel size appears to be right before the kernel has begun to spill to local memory. Each kernel saves, into global memory, the new value for each state variable that it is responsible for computing. We determine the dependencies for each state variable from the AST, so we can

[2] http://wiki.github.com/laanwj/decuda

treat each as its own independent unit without the need to save intermediate results between kernel calls. Saving these variables would require writing them to global memory, the only memory space that persists across kernel calls.

Optimization 4: Dual GPUs. We engage both available GPUs using separate threads as required by CUDA.

5 Application Testbed

We performed left ventricular electrophysiology simulations using the Flaim cell model, which is a reaction-diffusion system [14]. We solve a set of ODEs describing the kinetics of reactions that can occur at every point in space, and we also solve PDEs describing the spatial diffusion of reactants. In our models, the reactions are the cellular exchanges of various ions across the cell membrane during the cellular electrical impulse. The diffusion process is the flow of current through the tissue that allows the electrical impulses to propagate. There is just one PDE of the form $dVm/dt = \nabla * D\nabla * Vm - (I_{ion}/Cm)$ in which I_{ion} is the sum of the ionic currents which are each given by ordinary differential equations. These ODEs are advanced by the ODE solver at each time step. This approach is not unique to cellular modeling and is used in other domains, e.g. circuit simulations. Because the ODE solver dominates the conventional (CPU) implementation we consider only the ODEs in this work.

We used a 5,280 finite element model with 42,240 collocation points (i.e. 8 collocation points per element). These collocation points exist at the level of parallelism we wish to exploit: at each ODE time step, each point must solve its own independent ODE system, and it does so as a separate CUDA thread.

Because our simulations are capturing events occurring at different time scales, our systems of ODEs are *stiff*. To overcome stiffness we used a single iteration backwards Euler method derived from the standard backwards Euler method and a single Newton-Raphson iteration [15].

Because single precision is significantly faster than double precision on 200-series nVidia GPUs [16] we have an incentive to use it if we can tolerate the resultant loss of precision. We use RRMS error, a standard measure used in the literature to compare solvers [17], to determine the effects of reduced precision. We compare our results against a trusted gold standard reference solution, in our case using *radau5* [18], a 5th order Runge-Kutta backwards Euler (implicit) method with adaptive step sizes. *Continuity* has used radau5 for all ODE solving for several years, which has demonstrated sufficient reliability even when working with very stiff ODE systems. We conducted a 250ms simulation, roughly the duration of an action potential in the Flaim model. We compared double precision (DP) radau5 with DP backwards Euler (both on the CPU), and observed an error of 0.86%. We used a step size of 0.00157 for backwards Euler, while *Radau5* used a variable step size. To see if our errors were accumulating, we compared our single precision (SP) backwards Euler GPU solver against the DP radau5 CPU solver and observed an RRMS error of 1.34%. This gives us confidence that we are obtaining reasonable accuracy with the backwards Euler

solver. We then measured the error on the GPU. The difference between DP and SP on the CPU was small: 0.9%. The error between SP on the GPU and SP on CPU was even smaller: only 0.01%.

We deem this error acceptable in light of the fact that there are much larger uncertainties in measuring the empirical parameters used in the cell models, but that the errors in these parameters are too small to be distinguished by experimentation (see [11]). Thus, our experience indicates that single precision arithmetic was sufficiently accurate for our needs, although, this result is application dependent[3].

6 Results

Our hardware platform consisted of an Intel Nehalem desktop with an nVidia GTX-295, as described previously. The GTX-295 is a single card with two GPUs, each containing 240 single precision arithmetic cores units running at 1242 MHz with a theoretical peak performance of 894 GFLOPS. The device is 1.3 capable. Tab. 1 displays the results of applying the various combinations of our optimizations. In interpreting these results, it is important to distinguish *local memory* from *local variables*. Like global memory, local memory resides in GPU DRAM, and suffers an approximate 150x latency penalty over register or shared memory access. Local variables are simply variables that are local to a CUDA C function, i.e. local automatic variables. The compiler will attempt to place local variables into the fast registers whenever possible, although overuse of local variables will cause the compiler to spill to slower local memory.

We note that the counts in Tab. 1 correspond to the execution of a single thread. Since there is a single CUDA thread for each of the 42,240 collocation points in the finite element mesh used by the simulator, and there is no significant branching activity, we obtain the total counts by multiplying the reported counts by 42,240. We measured the number of references to local and global memory and the number of operations using *decuda*, and then added a correction to take into account dynamic behavior that cannot be measured with the static information reported by *decuda*. In particular, we must add a correction to take into account a loop that Opt2L introduces to copy global data to local variables.

The naïve GPU implementation took 52.7 sec. to simulate 20.0 msec. of heartbeat. A combination of optimizations 1, 3, 2S, and 4 lead reduced the running time to 7.9 seconds on 2 GPUs, an improvement of 6.7. We ran with 64 threads per block and a device occupancy of 4 concurrent thread blocks.

Decuda reveals that Opt1 reduces the number of assembly instructions by a factor of 3.0, global and local memory references by 1.2. The actual running time drops by a factor of 1.7, indicating that global and local memory references have a larger impact on performance than instructions executed. As a result, we always applied Opt1 regardless of what other optimizations were applied subsequently. We list multiple optimizations within curly braces in the order

[3] Xing Cai, Simula Laboratory, Private Communication, 2009.

Table 1. The impact of optimizations on the running time of the ODE solver. We report the running time to simulate 20ms of a heartbeat. Optimizations were applied cumulatively, in the order listed. *CPUx* and *GPUx* are the speedups over the baseline, the naïve GPU implementation. *Ops* are the number of *ptx* assembly operations for the kernel. *GR, GW, LR, and LW* give the Global Reads, Global Writes, Local Reads, and Local Writes and *Tot*, their sum. Counts are static, except for Opt2L, as described later in this section. Though not shown, Opt. 1 was always applied.

Method	Time(s)	GPUx	Ops	GR	GW	LR	LW	Tot
Naïve	52.7	1.0	19747	1149	266	34	34	1483
1	31.5	1.7	6685	743	266	136	78	1223
2L	16.4	3.2	5317	90	88	154	150	482
3+2L	41.5	1.3	9404	526	88	217	526	1357
2L+4	8.1	6.5	5317	90	88	154	150	482
2S	18.3	2.9	6039	190	88	176	88	542
3	26.1	2.0	8205	792	266	0	0	1058
3+2S	13.8	3.8	7557	243	88	0	0	331
3+2S+4	7.9	6.7	7557	243	88	0	0	331

applied, as in Opt{1+3+2S}. We may omit Opt1 from the sequence, as its use is implicit, e.g. we denote the previous optimization as Opt{3+2S}.

In addition to the speedup of 6.7 compared with the naïve implementation, we are also interested in those that improve re-use within fast on-chip memory. The relevant optimizations are Op2L, Opt2S, and Opt3. The best combination was Opt{3+2S}, which increased performance by 2.3x. As Tab. 1 shows, this result was obtained by eliminating all local memory accesses and 2/3 of the global memory accesses after Opt1 had been applied. This reduction more than compensates for a 10% increase in the instruction count, which as shown previously appears to be less a determinant of performance.

Prior to Opt3, Opt2L is the most effective. Although the *ptx* assembly reveals that data is simply copied from global memory to local variables, the compiler is able to allocate frequently accessed data to registers effectively. Indeed, the number of global memory references dropped from 1009 to 178, while the local memory references increased from 214 to 304, a net decrease in off-chip references. These results indicate that even when forced to spill, the compiler can do a good job of managing registers and local memory. We suspect that if the compiler could also spill to shared memory (as we have done with Opt2S), it would generate even more efficient code.

Opt2L also decreases the total instruction count. Many of these can be attributed to the net reduction in global and local references (which actually decreased)–741 of the 1368 instructions that were optimized away by Opt2L. Although the exact behavior of the compiler is proprietary, it appears that a good deal of the remaining savings comes from avoiding the need to compute indices based on the thread index, which consumes 3 instructions.

Unlike Opt2L, Opt2S caches global memory using statically determined replacement (i.e. at translation time) without adding any conditionals to the

generated code. The cache resides in SM shared memory, which is otherwise unused by our application, other than to transfer kernel arguments. Because shared memory is almost as fast as registers, it is ideally suited as a global memory cache. We had sufficient shared memory to cache 11 variables, achieving a miss rate of 15.2%, not far from the rate achieved by an infinite sized cache (13.8%, 24 words); cold start misses were costly. A larger cache reduced the occupancy, degrading performance. Due limitations on cache size, Opt2S resulted in 100 more global memory references than Opt2L. However Opt2L forces the compiler to spill some local variables to local memory, 40 in all, and these accesses are about as costly as global memory. Thus, Opt2S results in a net increase of 60 accesses to slow device memory. Together, the increases in instruction count (13.6%) and slow memory accesses lead to a performance difference of about 11%.

Up to this point, Opt 2L runs slightly faster than Opt2S. This is the result of a lower instruction count and fewer accesses to slow device memory. However, we can improve performance of Opt2S still further if we split the kernel (Opt3), which reduces register pressure. We can see this in Tab. 1, where the local memory references have dropped to zero. To probe further, we examined the CUDA source code. After splitting, the largest kernel referenced 67 different global variables and 128 different local variables, compared to 175 global variables and 267 local variables, respectively, in the monolithic kernel. With less than half the variables to manage, the compiler can allocate all *local variables* to *registers* without spilling any to *local memory*. However, because kernels run to completion, they must communicate via global memory or redundantly recalculate intermediate variables common to multiple kernels. We chose a compromise in which state variables are saved to global memory and temporary variables are recalculated, which added 53 global memory references. These added references, plus an increase in the number of instructions executed (from 6039 to 7757), was more than compensated for by eliminating all accesses to local memory. Opt3+2S runs 33% faster than Opt2S alone, we are now running 18% faster than Opt2L.

By comparison, under Opt2L, the CUDA C kernel (as opposed to *ptx*) references 175 global variables 178 times, yet references its 354 local variables 1585 times. Although the compiler clearly manages these 1585 references effectively, by using registers when possible, it is unable to eliminate 304 local memory references due to register pressure. Unfortunately, we were unable to apply kernel splitting to Opt2L, for reasons described in Lionetti's dissertation [15]. Further work remains to combine the two optimizations successfully.

As mentioned previously *decuda* provides only a static analysis. However, because we use a loop to copy global to local variables we must augment the counts of global and local references accordingly. Manual inspection of the generated assembly indicates that to account for this loop we needed to add 86 global memory references, 86 local references and 602 operations (at 7 instructions per copied reference) to the static count, which we have included in Tab. 1.

Our final optimization is to use both GPUs present in the GTX-295 video card. Working together, the GPUs are able to achieve an additional 1.7x speedup, consistent with that observed by others [8].

7 Discussion

All told, our optimizations improved performance by a factor of 6.7, with improvements in re-use (Opt2L and 2S) accounting for up to a factor of 2.3. Opt2L is far simpler than Opt2S–it merely copies from global to local memory, and relies on the compiler to manage the rest. Although the software cache outperforms the copying strategy, this is likely because it is amenable to kernel splitting, while copying is not. Were we be able to apply kernel splitting to Opt2L, then {Opt3+2L} may deliver the highest performance.

We looked at simpler cell models with fewer state variables– 2, 8, and 18 (see Lionetti's dissertation [15] for further details.) In each case there were sufficient registers to store all state variables and intermediate local variables without spilling to local memory. Since register access times are lower than shared memory access times [8], copying is preferable to a shared memory cache for smaller models. However, recent trends in cell model formulation suggest that more state variables are likely to be needed, rather than less, requiring creative optimizations such as those we have applied to the Flaim model.

A limitation of our shared memory cache is that because it employs static replacement, it cannot handle certain forms of conditional execution. However, it is a common practice to use lookup tables and other techniques to eliminate costly conditional execution whenever possible. For example, we used predication to elide conditional execution in our cell models (the *Heaviside* function in Fig. 1.) Therefore our offline caching technique may be applicable to certain common cases. However, when Opt2S is not applicable, we can use Opt2L, which we found to be competitive.

Our results have implications for Fermi, nVidia's next generation GPU architecture [19]. Although Fermi is not available at the time of writing, we are able to make some projections based on what is known about the processor publicly. From our perspective, the principal differences are that the number of cores has doubled, but the amount of fast on-chip storage in registers and shared memory essentially hasn't changed. In particular, the amount of register storage *per core* has dropped by 1/2. There is an L2 cache (768KB) shared by all streaming processors. But a chip-wide shared resource of this size is surely not as fast as registers, and the added storage is not sufficient to make up for the loss in registers per core. The Flaim model used all available registers for Opt2L (copying) and nearly all registers for Opt{3+2S} (kernel splitting and shared memory caching). As a result, we expect that occupancy will be reduced on Fermi. Perhaps the new on-chip L1 caching structure will help Opt2L, and we await the opportunity to evaluate the architectural impact on our optimizations.

8 Conclusions and Future Work

We have demonstrated an effective translation technique that encapsulates expert knowledge, allowing domain scientists to work at the level of the cell model without becoming entangled in low-level implementation details. Perhaps upcoming CUDA compilers will implement our copying strategy or the software managed cache. However, we believe that if a compiler were to succeed, it would need to exploit domain-specific knowledge that was unavailable to a general-purpose compiler.

Our approach is based on the observation that the GPU constrains the kinds of memory traffic that a well performing program will generate. Applications tend to update global memory infrequently and exhibit more predictable memory access patterns than on traditional CPUs. An interesting question is to what extent our approach will generalize. It should apply to an important class of *reaction-diffusion* systems arising in many applications in computational science and engineering, where threads do not share state, or to applications with phases bearing similar localized behavior. However, irregular problems are notoriously unpredictable, thus forcing the compiler to make conservative decisions to handle data dependent conditions.

Our translator is an enabling technology for running complex cell models on the GPU and has been put into production within the *Continuity 6* simulation infrastructure. In addition to the obvious benefits of faster cardiac simulations (e.g. faster research, larger models), our approach enables the accelerated heart simulator to be a transformational tool in the clinical setting; a key step to using cell models for patient-specific modeling and diagnosis. Simulations must be highly efficient in clinical applications because of the time constraints involved in diagnosis and treatment, and because the simulations typically must be run many times in parameter sweeps to be useful.

Having reduced the ODE bottleneck in electrophysiology simulations we are investigating solvers for sparse systems of linear equations that will reduce the PDE solver time, which is now the bottleneck. We are also interested in extending our results to GPU clusters and to exploring more exotic ODE solvers.

Acknowledgments

We thank Stuart Campbell (UCSD Bioengineering), Dr. Chris Anderson (UCLA Mathematics), and Dr. Xing Cai (Simula Research Laboratory) for the discussions on ODE Solvers and Dr. Jazmin Aguado-Sierra (UCSD Bioengineering) for supplying the sample Electrophysiology model. We also acknowledge the National Biomedical Computational Resource (NIH grant P41RR08605), the National Science Foundation (NSF grant BES-0506252), and the Simula Research Laboratory for supporting this research. Scott Baden dedicates his portion of this research to the memory of J. Ben Rosen (1922-2009).

References

1. Lionetti, F., McCulloch, A., Baden, S.B.: Gpu accelerated solvers for odes describing cardiac membrane equations. In: nVidia GPU Technology Conference (October 2009)
2. Sato, D., Xie, Y., Weiss, J.N., Qu, Z., Garfinkel, A., Sanderson, A.R.: Acceleration of cardiac tissue simulation with graphic processing units. Medical and Biological Engineering and Computing 47(9), 1011–1015 (2009)
3. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for gpu program optimizations. In: Int. Parallel and Distributed Processing Symp., pp. 1–10 (2009)
4. Silberstein, M., Schuster, A., Geiger, D., Patney, A., Owens, J.D.: Efficient computation of sum-products on gpus through software-managed cache. In: Proc. 22nd Ann. Intl. Conf. Supercomputing, pp. 309–318 (2008)
5. Eichenberger, A.E., OÕBrien, J., OÕBrien, K., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z.: Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. IBM Systems Journal 45(1), 59–84 (2006)
6. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. 49(4/5), 589–604 (2005)
7. NVidia: Cuda (2009), `http://www.nvidia.com/object/cuda_home.html`
8. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (2008)
9. Certik, O.: Sympy python library for symbolic mathematics (2008), `http://cadadr.org/fm/package/sympy.html`
10. Bendersky, E.: Pycparser (2009), `http://code.google.com/p/pycparser`
11. Flaim, S.N., Giles, W.R., McCulloch, A.D.: Contributions of sustained ina and ikv43 to transmural heterogeneity of early repolarization and arrhythmogenesis in canine left ventricular myocytes. American Journal of Physiology- Heart and Circulatory Physiology 291(6), H2617 (2006)
12. Belady, L.A.: A study of replacement algorithms for virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
13. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Pub., San Francisco (2001)
14. Britton, N.: Reaction-diffusion equations and their applications to biology. Harcourt Brace Janovich, Orlando, FL 32887-0405(USA), 1986, 288 (1986)
15. Lionetti, F.: Gpu accelerated cardiac electrophysiology. Master's thesis f (2010), `http://www-cse.ucsd.edu/groups/hpcl/scg/papers/2010/lionetti_ms_thesis.pd`
16. Bell, N., Garlandy, M.: Efficient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Corporation, Santa Clara, CA, USA (December 2008)
17. Dean, R.C., Spiteri, R.: On the performance on an implicit-explicit runge-kutta method in models of cardiac electrical activity. IEEE Transactions on Biomedical Engineering 55(5) (2008)
18. Hairer, E., Noersett, S.P., Wanner, G.: Solving ordinary differential equations. Springer, Heidelberg (1993)
19. nVidia: Nvidia's next generation cuda compute architecture: Fermi (2009), `http://www.nvidia.com`