Optimizing the Aliev-Panfilov Model of Cardiac Excitation on Heterogeneous Systems

Didem Unat *[1], Xing Cai[2], and Scott B. Baden †[1]

[1]*Computer Science and Engineering, University of California-San Diego, USA*
[2]*Department of Informatics, University of Oslo, and Simula Research Laboratory, Norway*

Abstract The Aliev-Panfilov model is a simple model for signal propagation in cardiac tissue, and accounts for complex behavior such as how spiral waves break up and form elaborate patterns. Spiral waves can lead to life-threatening situations such as ventricular fibrillation. We discuss an implementation and underlying optimizations for the nVIDIA Tesla C1060 GPU as well as an implementation on multiple GPUs running under MPI. We achieve nearly perfect scaling on 4 GPUs, in single precision, running 58 times faster than a CPU-only implementation and 26 times faster in double precision.

Keywords cardiac simulation, CUDA, GPU programming

1 Introduction

Numerical simulations play a vital role in health sciences and biomedical engineering and can be used in clinical diagnosis and treatment. For example, we can simulate the propagation of electrical signals in the heart using the Aliev-Panfilov model [1], which is a reaction-diffusion system [2]. The model comprises two parts: a system of two ordinary differential equations (ODEs) and a partial differential equation (PDE). The ODE part describes the kinetics of reactions occurring at every point in space, and the PDE part describes 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.

Our simulation has two state variables. The first corresponds to the transmembrane potential while the second represents the recovery of the tissue. Two numerical methods for solving the Aliev-Panfilov model were thoroughly discussed in Hanslien et al. [3]. For the GPU implementation in the present paper, the first-order explicit numerical scheme from Hanslien et al. is adopted.

Although the Aliev-Panfilov model is simple, it can demonstrate complex behavior of spiral waves that are known to cause life-threatening situations such as ventricular fibrillation. Since simulations run slowly at higher resolutions, we have an incentive to accelerate the simulation in order to improve turnaround time.

Graphic Processing Units (GPUs) are an affective means of accelerating cardiac simulations [5], including the Aliev-Panfilov model. Owing to their small size, it is possible to integrate them in the biomedical devices in a clinical setting GPUs provide a large number of streaming cores on a single chip and high on-chip memory bandwidth. Both the ODE and PDE solvers are highly data parallel and can take advantage acceleration. The ODE solver requires a lot of computational power and the PDE solver requires high memory bandwidth.

In this paper, we present results with, and an implementation of, the Aliev-Panfilov model running under the CUDA programming model [6]. We also describe a multi-GPU implementation that uses MPI to couple the GPUs, along with latency hiding techniques. We conclude with a performance evaluation to evaluate the effectiveness of our optimizations.

2 Implementation Details

The cardiac simulation uses the finite-difference technique for the PDE solver, which is similar to a Jacobi update for the 5-point Laplacian operator. The solver sweeps over a uniformly spaced mesh, updating the voltage according to weighted contributions from the four nearest neighboring positions in space, as illustrated in Figure 1. The PDE solver requires ghost cell communication with the neighboring threads in every iteration. On the other hand, the ODE solver is trivially parallelizable, because there are no data dependencies. We combine the two solvers into one CUDA kernel to minimize the global memory traffic.

Data Decomposition. On a GPU device, it is not possible to fit the entire grid into high level memory. Instead, the input grid must be divided into tiles. A typical approach would be 2D domain decomposition that generates 2D thread blocks. However, we observe benefits of generating 1D thread blocks for a 2D tile, which effectively reduces index calculations and enables reuse of data in registers or shared memory. As a result,

*Didem Unat was supported by a Center of Excellence grant from the Norwegian Research Council to the Center for Biomedical Computing at the Simula Research Laboratory.

†Scott Baden was supported by the Center for Biomedical Computing at the Simula Research Laboratory.

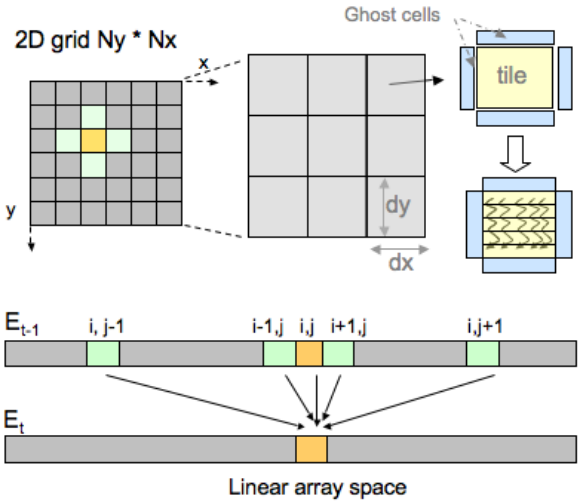

Figure 1: Stencil computation on a 2D grid and its memory access pattern. Data and thread block decomposition of N_x x N_y input mesh. The mesh is divided into d_x x d_y tiles. Each CUDA thread processes a column in a tile.

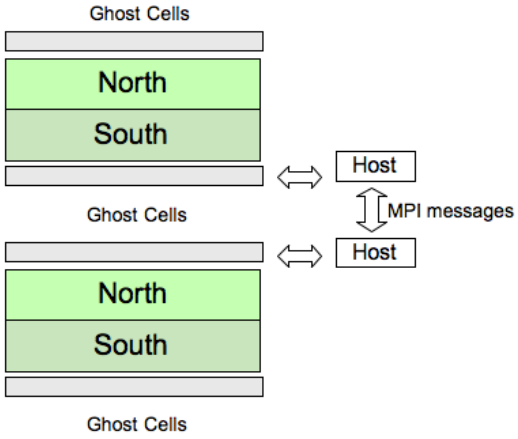

Figure 2: North-south computation allows asynchronous communication, thus hiding latency.

we divide the input grids into tiles of size $d_x \cdot d_y$ and create a one-dimensional thread block of size d_x to process a tile. A thread block processes one row of a tile at a time and a thread computes the entire column of a tile, as illustrated in Figure 1. Compared with a 2D thread blocking strategy, one-dimensional thread blocks provide a 12% improvement in double precision performance and 64% improvement in single precision.

Utilizing Registers and Shared Memory. Updating a row requires the rows above and below that row. We store the top and bottom rows in registers, but we keep the center row in shared memory since all threads share that row. To reduce the number of global memory accesses, we reuse the data already in shared memory and registers by letting a thread block compute more than one row. We implement this by keeping a queue of rows and rotating them. At all times we have 1 row in shared memory and 2 rows in registers. In this scheme, a row starts at the bottom, continues to the center and then to the top.

Ghost Cells. The ghost cells within a tile require special handling. To handle the east and west borders, we could create additional threads. However, this strategy would leave the border threads idle during computation. Instead, we choose to use threads in the first and last column of a thread block to read the east and west borders. The north and south ghost cells do not need special handling because all threads share the entire row.

Multi-GPU Implementation. At higher resolutions, simulations run very slowly. We project that a simulation would take around 17 hours on a single GPU for an input size of N=4K. Using 4 GPUs reduces the running time to 4 hours.

In a multiple-GPU implementation, we partition the input mesh along the slowest varying dimension among GPUs. One-dimensional partitioning allows us to transfer contiguous memory locations during ghost cell exchange without any data manipulation. We use MPI to manage host processes. Each GPU updates its region and then exchanges ghost cells with the other two neighbors before starting the next iteration. The only way for two GPUs to communicate is through their host processes. As a result, there are three communication links between two GPUs: GPU-host, host-host and host-GPU. We use *cudaMemcpy* between GPU-host and host-GPU communications and *MPI Isend-Irecv* for the host-host communications. Since CUDA kernel executions and memory copy calls are asynchronous, we can completely hide all the communication latencies as long as the communication time does not exceed the computation time.

For asynchronous communication, we divide the computation on a single-GPU into two regions: north and south computations as shown in Figure 2. While a GPU computes the north region, its host asynchronously copies the south ghost cells from the GPU, exchanges the ghost cells with the south neighbor and copies back the updated ghost cells to the GPU. Then, the GPU starts the computation on the south region while the host exchanges the north ghost cells. The disadvantage of doing north-south computation is that we need to launch two separate kernels with half the number of data blocks: one for north and one for south. In order for this approach to work, there should be enough thread blocks per kernel launch to occupy all the stream multiprocessors. As a result, we recommend using the latency hiding version for mesh sizes greater 4K.

N	T	Number of GPUs	Total Time (sec)	Communication Percentage	Number of Iterations	Speedup over 1 GPU
1K	1000	1	276.12	-	231291	1
2K	1000	1	4102.64	-	893118	1
2K	1000	2	2224.30	4.1%	893118	1.84
4K	10*	1	647.65	-	35413	1
4K	10*	2	333.75	3.1%	35413	1.94
4K	1000	4	17172.50	4.4%	3541290	3.77

Table 1: Simulation running times for different mesh size in double precision for T=1000. To save time and energy consumption we ran certain simuilations marked with an asterisk (*) until T=10.

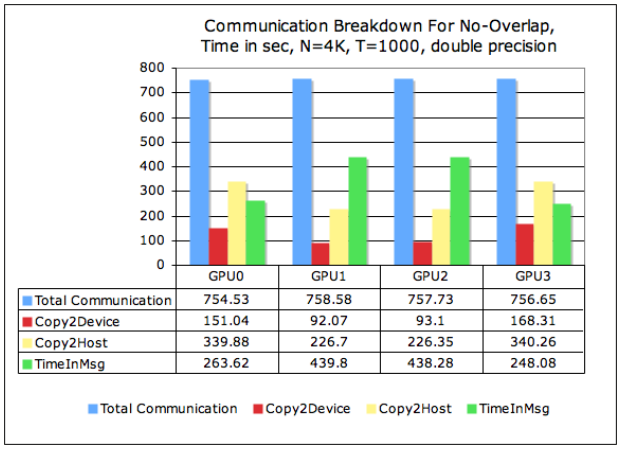

Figure 3: Communication latencies on 4 GPUs.

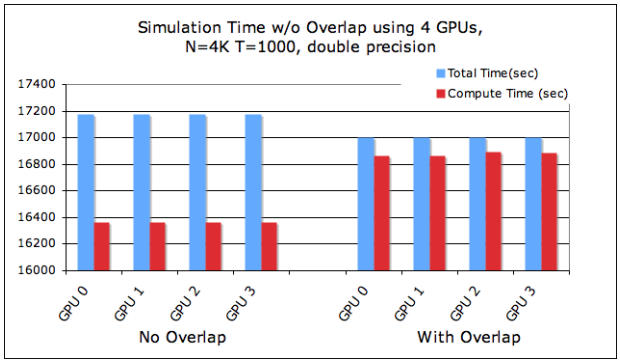

Figure 4: Comparison of simulation time w/o overlapping of communication and computation.

3 Results

We measure the performance of our implementation on a system with 4 nVIDIA Tesla C1060 devices with 4GB device memory in each GPU [4]. This device is 1.3 capable and ran under CUDA v2.3. Device code was compiled with `nvcc` using compiler options `-compiler-options -fno-strict-aliasing -O3 -arch Smylers_13`. The host CPU is a quad-core Intel Xeon processor based on the Nehalem architecture. Host code was compiled with `g++ 4.3.3` with using compiler options `-O3 -fno-strict-aliasing`.

We present the results for meshes of size 1K, 2K and 4K. Due to long execution times of some simulations, we project running times based on the $1/100^{th}$ of the actual simulation time. Storage of the final numerical solutions is of interest for visualization. We visualize data at a low enough frequency that did not observe any significant overhead in transferring data to the host, nor in the I/O time needed to display it. Hence, the results presented in this paper include both I/O and data transfer times.

Tab. 1 shows the simulation times for several mesh sizes. 1 GPU versions use a single-GPU implementation, others use the multiple-GPU implementation. In order to see where the time is spent we measure the simulation times with a synchronous version of the multiple-GPU implementation (no overlapping of communication and computation). As seen in the table, communication constitutes only a small portion of the execution time, around 3-4%.

Fig. 3 shows the breakdown of communication latencies for the synchronous multi-GPU implementation. The total communication includes data transfers between the host and the device and message passing between two hosts. On GPUs it is typically faster to transfer data from host to device than the other way around, and we observe similar behavior. The message passing time is higher for the GPUs 1 and 2 because they exchange ghost cells with two neighbors rather than one.

Fig. 4 compares the overlapping and non-overlapping timings for a mesh size 4K x 4K. The total execution time is reduced by less than 2%. Since communication is not the bottleneck and constitutes 3-4% of the overall simulation time, the benefit of latency hiding is not significant. However, this is an artifact of our using a small number of GPUs, and we expect that communication will become more costly as we increase the number of GPUs. We will report on results with more GPUs in the future.

4 Performance Analysis

We compare a CPU implementation a single-GPU performance. Tab. 2 compares GFlop rates for the Aliev-Panfilov method on the host and on the GPU for both single and double precision. We ran with a 4K x 4K mesh

N=4K	1 CPU-only	1 GPU	Speedup over CPU
Single	2.36	124.48	52.84
Double	2.01	25.69	12.81
N=16K	4 CPUs-only	4 GPUs	Speedup over CPU
Single	7.79	454.47	58.32
Double	3.88	102.48	26.41

Table 2: GFlop/s rates for CPU-only and GPU

Double Precision, N=4K	Using Div	Replace(div, add)
Peak BW (GB/s)	102	102
Peak DP GFlop/s	78	78
Sustained BW (GB/s)	73	73
Achieved Bandwidth (GB/s)	30.3	51.3
Achieved GFlop/s	25.7	43.6
% of Sustained Bandwidth	41.5 %	70.3 %
% of Peak GFlop/s	32.9 %	55.8 %

Table 3: Performance results if the division operation is replaced with an add operation in the kernel. BW: bandwidth, DP: double precision.

and a single MPI process (results are similar for an 8K x 8K mesh and a 16K x 16K device overflows memory). The single precision kernel offers a speedup of 53 over the host implementation. It nearly saturates the off-chip bandwidth, utilizing 98% of the sustainable bandwidth for Tesla (73 GB/sec), and achieving 124.48 GFlop/s, which is only 13.3% of the single precision peak performance of the Tesla C1060. This indicates the single precision kernel is memory-bandwidth limited. Double precision performance is not as high as single but it still outperforms the CPU. It utilizes only 41.5% of the sustained bandwidth and delivers 1/3 of the peak performance. This is true because each GPU vector core shares a single double precision arithmetic unit; hence the peak floating point rate in double precision is only 1/8 of that in single precision. Not surprisingly, this kernel is floating point bound (instruction throughput) , rather than bandwidth limited.

Moreover double precision performance is severely affected by the *division* operation in the ODE solver. In order to observe the cost of division operation, we replace the division operation with *addition*. Even though the solution is numerically incorrect, the resulting program behavior does not change the memory profile or floating point operation count in the kernel. Tab. 3 shows the GFlop/s rate after the replacement and compares it with the use of division. The modified kernel achieves 55.8 % of the double precision peak performance. We conclude that our Aliev-Panfilov double precision performance is acceptable, considering not all the operations are multiply-and-add instructions, that achieve the highest instruction throughput on the current nVIDIA GPUs. In addition, due to the limitation on the number of registers available, some of the operands are in shared memory rather than in registers. Even though the performance of our implementation is satisfactory, as a future work we would like to close the gap and improve our double precision results.

5 Conclusions

We have accelerated the Aliev-Panfilov cardiac model using the nVIDIA Tesla GPU. Our optimizations utilize shared memory and registers to reduce the number of global memory accesses to achieve higher performance. Our single precision implementation runs at near device memory bandwidth and double precision implementation achieves one-third of the double precision peak performance. Our multiple-GPU implementation reduces the simulation time further and enables solutions on meshes with high resolution. Finally, we compare our GPU implementation with CPU-only implementation. We achieve a speedup of about 50 in single precision and a somewhat smaller speedup in double precision, depending on the problem size.

Acknowledgments

Computations on the NVidia Tesla system located at UCSD were supported by NSF DMS/MRI Award 0821816.

References

[1] R. Aliev and A. V. Panfilov. A Simple two-variable model of cardiac excitation. *Chaos, Solions & Fractals*, 7(3):293-301, 1996.

[2] N. Britton. Reaction-diffusion equations and their applications to biology. *Harcourt Brace Janovich, Orlando, FL 32887-0405(USA), 1986, 288*, 1986.

[3] M. Hanslien, R. Artebrant, A. Tveito, G. Lines, and X. Cai. Stability of two time-integrators for the aliev-panfilov system. 2010. Submitted for publication.

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[5] F. Lionetti, A. McCulloch, and S. B. Baden. Gpu accelerated solvers for odes describing cardiac membrane equations. In *nVidia GPU Technology Conference*, October 2009.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.