Towards Detailed Real-Time Simulations of Cardiac Arrhythmia

Johannes Langguth[1], Hermenegild Arevalo[1], Kristian Gregorius Hustad[1], Xing Cai[1,2]

[1] Simula Research Laboratory, Lysaker, Norway
[2] University of Oslo, Oslo, Norway

Abstract

Recent advances in personalized arrhythmia risk prediction show that computational models can provide not only safer but also more accurate results than invasive procedures. However, biophysically accurate simulations require solving linear systems over fine meshes and time resolutions, which can take hours or even days. This limits the use of such simulations in the clinic where diagnosis and treatment planning can be time sensitive, even if it is just for the reason of operation schedules. Furthermore, the non-interactive, non-intuitive way of accessing simulations and their results makes it hard to study these collaboratively. Overcoming these limitations requires speeding up computations from hours to seconds, which requires a massive increase in computational capabilities.

Fortunately, the cost of computing has fallen dramatically in the past decade. A prominent reason for this is the recent introduction of manycore processors such as GPUs, which by now power the majority of the world's leading supercomputers. These devices owe their success to the fact that they are optimized for massively parallel workloads, such as applying similar ODE kernel computations to millions of mesh elements in scientific computing applications. Unlike CPUs, which are typically optimized for sequential performance, this allows GPU architectures to dedicate more transistors to performing computations, thereby increasing parallel speed and energy efficiency.

1. Introduction

The Virtual Arrhythmia Risk Predictor (VARP) [1] has been shown to provide better predictions on patient outcomes w.r.t. ICD implantation than invasive clinical procedures. However, it is computationally expensive. Currently, it relies on tools provided by the Cardiac Arrhythmia Research Package (CARP) [2], a flexible software suite for cardiac electrophysiology simulations.

In this paper, we investigate whether designing a specialized code built on state of the art best practices of high-performance computing is capable of speeding up the computation enough to affect clinical usage in the real world. Currently, the CARP based simulations take about 24 hours per patient. Arguably, speeding up the computation by a factor of 10 is not enough to fundamentally change how clinicians can interact with the simulation. A factor of 100 however would mean that results can be easily be delivered during a patient visit, while a factor of 1000 would imply that the simulations are approaching real time.

Similar work has been done in the past for supercomputers [3]. However, such devices are not realistically usable in a clinical setting. Thus, we focus on maximizing performance on devices that could be operated by a clinic onsite. Doing so is a crucial requirement in many countries due to patient data privacy. We attained a speedup of about 125x compared to running CARP on a single CPU-based supercomputer node[1]. This is roughly equivalent to simulating two heartbeats per minute using 11.7 million computational cells. In comparison, Cardioid [3] was estimated to be capable of simulating nine beats per minute on 370 million computational cells using the entire *Sequoia* supercomputer. A cluster of about 150 DGX-2 nodes would be needed for `LYNX` to match that value. Such a cluster would also have roughly the same peak performance of about 20 PFLOP/s. However, `LYNX` is designed for unstructured mesh, which means that it needs fewer computational cells than Cardioid for a comparable accuracy. Furthermore, VARP [1] has shown the clinical suitability of simulations of much smaller size.

During the last decade, it has become increasingly clear that manycore accelerators such as GPUs, which offer higher throughput at the expense of latency, are the preferred architecture for many applications in scientific computing, especially fully explicit schemes. Typically, the same computation is applied to a large number of computational cells in a different state, thus creating a natural opportunity for wide *single instruction multiple data* (SIMD) processing. In previous work, we studied the effect of combining CPUs and GPUs on the same computation [4]. While the experiment was successful, it was based on the assumption that the CPUs would provide 20-40% of

[1]This was the scaling limit in our experiments, but it should not be considered a benchmark for CARP.

the total computational power in a node. However, in the last five years, so called *fat nodes* with at least four GPUs[2] have become common. In such nodes, the CPUs typically provide less than 10% of the total computational power, both by memory bandwidth and by the number of FLOP/s. In addition, memory bandwidth contention can slow down MPI communication when the CPU is also used for computation [5]. Thus, fully heterogeneous computation no longer pays off due to the synchronization cost and additional code complexity involved in such a scheme. Consequently, our code is designed as a pure GPU code where the CPU only handles communication and control tasks.

2. Computational model

Diffusion is a very common phenomenon in nature, whose simplest mathematical description is the following partial differential equation:

$$\frac{\partial u(x,t)}{\partial t} = \text{div}(\vec{K}(x)\text{grad } u) \tag{1}$$

where $u(x,t)$ is typically some concentration modeled as a function of space and time, and $\vec{K}(x)$ denotes a spatially varying tensor field that, together with the concentration gradient, determines the speed and direction of how high concentration spreads towards low concentration.

We use a cell-centered finite volume approach for numerically solving (1) in 3D on an unstructured tetrahedral mesh. The same method was used and is described in more detail in Langguth et at. [4]. Each tetrahedron constitutes a computational cell. The method aims to find u_i^ℓ as the approximation of $u(x_i, \ell\Delta t)$, where x_i denotes the geometrical center of tetrahedron i, ℓ denotes the discrete time level, and Δt the length of the time step.

The finite volume method starts with integrating (1) over each tetrahedron. Using a forward Euler temporal discretization, as well as the famous divergence theorem, this will produce for tetrahedron i the following result:

$$\int_{V_i} \frac{u^\ell - u^{\ell-1}}{\Delta t} dx = \int_{\partial V_i} \vec{n} \cdot (\vec{K}(x)\text{grad } u^{\ell-1}) \tag{2}$$

where V_i denotes the volume of tetrahedron i, ∂V_i denotes its surface (i.e., four triangular faces), and $\vec{n}$ denotes the outward normal vector on ∂V_i. Then, the left-hand side of (2) is approximated by $\frac{u_i^\ell - u_i^{\ell-1}}{\Delta t} V_i$, whereas the surface integral on the right-hand side is approximated individually over the four triangular faces.

Without going into the mathematical details, it suffices to say that a second-order spatial discretization can give rise to an explicit formula for calculating u_i^ℓ as a weighted sum of $u_i^{\ell-1}$ and values from its four first-tier neighboring tetrahedra, as well as values from the twelve second-tier neighbors.

Because the second-order spatial discretization allows an explicit formula for calculating u_i^ℓ, the entire computation for time step ℓ can be written as a sparse matrix-vector multiplication (SpMV):

$$\mathbf{u}^\ell = \mathbf{Z}\mathbf{u}^{\ell-1}, \tag{3}$$

where $\mathbf{Z}$ is a sparse matrix that has, in addition to a nonzero main diagonal, up to 16 nonzero values per row. This suggests the use of the ELLpack format, which is known to be very efficient for GPUs [6]. We modify it by storing the main diagonal in a separate vector D. Thus, the off-diagonal entries are stored in a padded, dense $N \times 16$ matrix A where N denotes the total number of tetrahedra in the mesh. Consequently, the column indices are stored in a matching $N \times 16$ matrix of integer index values called I. In both A and I, in case of nonexistent or duplicate neighbors, rows are padded to exactly 16 entries.

For the reaction part, we use the ten Tusscher model (2006 version) [7] which tracks 19 state variables per cell. The model is provided using CellML [8], which is then translated to CUDA using the gotran code generator [3].

3. Implementation

We tested multiple numerical schemes including forward Euler (FE), Rush-Larsen (RL), first order accurate generalized Rush-Larsen (GRL1) and a method applying Rush-Larsen for stiff m gates and FE for all other state variables. This method is referred to as lightweight Rush-Larsen (LRL). The methods differ in accuracy and performance. FE is fast, but it only offers acceptable accuracy at a time step of $2\mu s$. The other methods offer an acceptable accuracy of about 97% already at a time step of $20\mu s$. Among the remaining methods, LRL outperformed RL slightly and GRL1 by about a factor of 2. Thus, we chose LRL for our implementation.

For the cell model, we performed a number of manual optimizations. They mostly include coalescing of memory accesses precomputing some cell type dependent values and reducing warp divergence when cells in a warp are of different type. Other optimizations include the extraction of integer powers and the elimination of common subexpressions in order to reduce the number of exponential function calls. Unlike Cardioid [3], `LYNX` does not use polynomials to approximate exponential functions.

Since the diffusion is a sparse matrix-vector multiplication (SPMV), we use established optimizations for the problem. Since most tetrahedra in the mesh have four

[2]Examples include the DOE Summit and Sierra supercomputers, and the Japanese AI Bridging Cloud Infrastructure or TSUBAME3.0 (http://www.top500.org).

[3]https://bitbucket.org/simula-camo/gotran/src/master/

neighbors, the stiffness matrix has usually 16 or 17 nonzeroes per row. We thus store the matrix in an ELLpack format with *integer* indices and *double* values. This is know to be efficient for SPMV on the GPU [6]. In addition, we store the diagonal separately, since it is always nonzero and requires no index. As discussed in previous work [9] the block structure of a sparse matrix is crucial for SPMV performance. We thus use partitioning software such as Metis [10] to reorder the matrix rows into blocks. A plain implementation of (3) is given in the code segment below:

```
for (i=0; i<N; i++) {
  u_new[i] = D[i]*u_old[i];
  for (j=0; j<16; j++)
    u_new[i] += A[i,j]*u_old[I[i,j]];
}
```

To calculate each value in vector $\mathbf{u}^{\ell}$, i.e. $\mathtt{u_new}$, 33 floating-point operations (17 multiplications and 16 additions) are needed. Since the length of each row is only 16 while the warp size of the GPU is 32, we switch from row-major to column major multiplication. In order to do so, we transpose the storage order of blocks of 32 rows such that coalesced access to a column of length 32 becomes possible. We then use fused multiply-adds to perform the SPMV efficiently.

The code is written mostly in C, and uses an inconsequential number of C++ features. The parallel code combines MPI for inter-node communication, OpenMP for intra-node parallelization, and CUDA to access the GPUs. We use hierarchical partitioning to minimize the inter-node communication. Details can be found in previous work [4]. The tetrahedra on the inter-node boundary are aligned in such a way that very little packing is necessary. It would be possible to use CUDA-aware MPI and spawn one MPI process per GPU, but this tends to perform slightly worse than threaded implementations. Since our test system contains only one DGX-2 node, MPI communication is not active in these experiments. We considered using NVIDIA NCCL,[4] but the system currently lacks efficient support for the required communication patterns. Thus, parallelism is achieved by running one OpenMP thread per GPU to control the computation, while data between the GPUs is being transferred exclusively using GPUdirect [5] peer-to-peer transfers via *cudaMemcpy*. Transfers are overlapped with computation through high-priority streams.

4. Experiments

We test the `LYNX` code on a single NVIDIA DGX-2 [6] machine using CUDA 10.1. It is equipped with 16 V100 Volta GPUs connected by a 300 GB/s NVLINK-2 crossbar switch offering full bisection bandwidth, which is much faster than typical supercomputer interconnects such as Infiniband at 25 GB/s or PCI-E3 connections at 32 GB/s.

As a result, it is relatively easy to perform intra-node communication fast enough so that it doesn't impact overall time. We also performed additional experiments on a server equipped with 8 NVIDIA P100 cards. These GPUs are connected with the more conventional PCI-E3 bus. This means that communication takes significantly longer, but it can still be overlapped with computation entirely.

We tested the simulator with 30 meshes from VARP [1]. In 24 cases, stability was never problematic. For the remaining 6 meshes, it was necessary to reduce the time step. For our experiments, we used one mesh with 11.7 million tetrahedra. The meshes differ in performance only via different time steps length. There are no significant differences in the time per time step per tetrahedron.

4.1. Single device performance

As the interconnect of the test system is very fast, the main target for optimization was the single device performance. Table 1 shows the overall results. The last two rows indicate that we are very close to the limits of the GPUs. Further optimizations might be possible, but unless they reduce the amount of computation that is actually performed, their impact is likely to be very small. Interestingly, the V100 improves upon the P100 by 80%, rather than the roughly 50% that could be expected from its specifications.

Table 1. Performance achieved on a single device compared to memory and computational performance bounds. Running time is given as wall time per time step.

Time	P100	V100
Memory bound	5.64 ms	3.55 ms
Unoptimized compute bound	7.45 ms	4.34 ms
Optimized compute bound	5.22 ms	3.03 ms
Achieved	7.60 ms	4.20 ms
Memory bound ratio	0.742	0.845
Compute bound ratio	0.686	0.721

4.2. Multi device performance

Table 2 shows the results using up to 16 GPUs. We observe good but not perfect scaling. In order to find the reason for the scaling limitation, we performed additional experiments. First, we measured the time required by the communication. This indicated that intra-node communication does not slow down overall execution on the DGX-2

[4] https://developer.nvidia.com/nccl
[5] https://developer.nvidia.com/gpudirect
[6] https://www.nvidia.com/en-us/data-center/dgx-2

or the P100 based server, although for the latter the margin is much smaller. In a second experiment, we tested the single-device performance on smaller instances, which drops by about 7% when cutting instance size by a factor of 16, i.e. from 11.7 million to 730,000 due to the lack of parallelism. The remaining loss in parallel efficiency can be attributed to load balance. Since the problem is irregular and performance varies depending on cell type and state, removing the remaining load imbalance is not feasible.

Table 2. Scaling performance achieved on up to 16 GPUs. Time is the wall time required for one second of simulation. Clearly, the application scales reasonably well. Mem. ratio is the ratio of the achieved performance compared to the theoretical limit set by the device's memory bandwidth.

GPUs	Time (s)	Speedup	Efficiency	Mem. ratio
1	400.1	1.000	1.000	1.156
2	208.5	1.919	0.959	1.205
4	105.6	3.790	0.947	1.220
8	53.8	7.437	0.930	1.244
16	28.2	14.208	0.888	1.302

5. Conclusions and future work

We have shown that electrophysiological simulations can be accelerated dramatically from taking a day to a few minutes by selecting the correct combination of numerical schemes, implementation, and hardware. Unlike previous work which made use of an entire supercomputer, the required hardware is now within the limits of what is feasible to install in a hospital. Even at 16 GPUs, our implementation is within 30% of the theoretical limit set by the memory bandwidth, indicating that we have essentially exhausted the potential of our numerical method.

Naturally, the ultimate goal is to accelerate the computation to real time simulations. However, while weak scaling is relatively easy to achieve with the current code, the strong scaling required to attain real time is not. Since GPUs hide latencies by running a large number of concurrent threads, they depend on having a sufficient amount of parallelism. If that requirement is not met, efficiency drops rapidly. In addition, both the communication latencies and the lengths of the time steps are in the low microsecond range. At real time, this can become a serious limitation.

Thus, it appears that our current design and system size occupies a sweet spot w.r.t. achieved performance compared to the resources required. Going towards real time is likely to require a new combination of numerical scheme, low-latency computational hardware, low-latency interconnects, and optimized implementations.

Acknowledgments

This research was funded by the Research Council of Norway (grant no. 251186/F20). It makes use of the *eX3 - Experimental Infrastructure for Exploration of Exascale Computing* Norwegian national infrastructure.

References

[1] Arevalo HJ, Vadakkumpadan F, Guallar E, Jebb A, Malamas P, Wu KC, Trayanova NA. Arrhythmia risk stratification of patients after myocardial infarction using personalized heart models. Nature Communications 05 2016; 7:11437 EP –.

[2] Plank G, Clayton R, Boyd D, Vigmond E. Integrative biology: modelling heart attacks with supercomputers. Capability Computing The newsletter of the HPCx community 2006;7:4–9.

[3] Richards DF, Glosli JN, Draeger EW, Mirin AA, Chan B, luc Fattebert J, Krauss WD, Oppelstrup T, Butler CJ, Gunnels JA, Gurev V, Kim C, Magerlein J, Reumann M, Wen HF, Rice JJ. Towards real-time simulation of cardiac electrophysiology in a human heart at high resolution. Computer Methods in Biomechanics and Biomedical Engineering 2013;16(7):802–805.

[4] Langguth J, Sourouri M, Lines GT, Baden SB, Cai X. Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes. IEEE Micro 2015;35(4):6–15.

[5] Langguth J, Cai X, Sourouri M. Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures. 2018 IEEE 24th International Conference on Parallel and Distributed Systems ICPADS 2018;497–506.

[6] Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[7] ten Tusscher KHWJ, Panfilov AV. Alternans and spiral breakup in a human ventricular tissue model. American Journal of Physiology Heart and Circulatory Physiology 2006;291(3):H1088–H1100.

[8] Cuellar AA, Lloyd CM, Nielsen PF, Bullivant DP, Nickerson DP, Hunter PJ. An overview of cellml 1.1, a biological model description language. Simulation 2003;79(12):740–747.

[9] Langguth J, Wu N, Chai J, Cai X. Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes. Journal of Parallel and Distributed Computing 2015; 76:120–131. ISSN 0743-7315.

[10] Karypis G, Kumar V. Multilevel k-way hypergraph partitioning. VLSI Des 2000;11:285–300.

Address for correspondence:

Johannes Langguth
Martin Linges vei 25, 1364 Fornebu, Norway
langguth@simula.no