GPU-based Acceleration of Detailed Tissue-Scale Cardiac Simulations

Neringa Altanaite
Simula research laboratory
Lysaker, Norway
altanaite@gmail.com

Johannes Langguth
Simula research laboratory
Lysaker, Norway
langguth@simula.no

ABSTRACT

We present a GPU based implementation for tissue-scale 3D simulations of the human cardiac ventricle using a physiologically realistic cell model. Computational challenges in such simulations arise from two factors, the first of which is the sheer amount of computation when simulating a large number of cardiac cells in a detailed model containing 10^4 calcium release units, 10^6 stochastically changing ryanodine receptors and 1.5×10^5 L-type calcium channels per cell.

Additional challenges arise from the fact that the computational tasks have various levels of arithmetic intensity and control complexity, which require careful adaptation of the simulation code to the target device. By exploiting the strengths of the GPU, we obtain a performance that is far superior to that of the CPU, and also significantly higher than that of other state of the art manycore devices, thus paving the way for detailed whole-heart simulations in future generations of leadership class supercomputers.

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; *Massively parallel algorithms*; *Multi-scale systems*; Parallel programming languages; • **Hardware → Emerging architectures**;

ACM Reference Format:
Neringa Altanaite and Johannes Langguth. 2018. GPU-based Acceleration of Detailed Tissue-Scale Cardiac Simulations. In *GPGPU-11: General Purpose GPUs, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3180270.3180274

1 INTRODUCTION

A realistic simulation of the human heart tissue requires the precise replication of all the intracellular and intercellular processes including the electrophysiology of the heart. Modeling a cardiac cell is a challenging task because of its complex interior structure. However, the understanding of the cardiac cell processes and their properties has increased significantly in the last decades. Models of cardiac cells of different species at different levels of detail have been developed in order to understand the cause of various heart diseases.

Due to the computational demands of realistic cell models, it is challenging to perform tissue scale cardiac simulations at this level of detail. A human heart has around 2×10^9 cells [1]. Each cell has about 10^6 ryanodine receptors (RyR) and 10^5 L-type calcium channels that are distributed among 10^4 calcium release units. New realistic simulations with detailed cell models of calcium handling at the tissue or organ level are now becoming computationally feasible due to the increased power of modern supercomputers.

Due to their massive parallelism, tissue-level simulations can take advantage of hardware accelerators. Motivated by the computational power of the GPU, we enable massively parallel simulations of calcium circulation at the dyad-level and implement the 3D Human Tissue-Scale model on heterogeneous clusters.

2 PHYSIOLOGICAL BACKGROUND

At the tissue-scale, electrical activity is modeled as a reaction-diffusion equation called the monodomain model. It is a simplification of a more accurate binomial modeling under the assumption that conductivity in extracellular space is proportional to conductivity in the intracellular space. It is governed by the monodomain equation (1):

$$\frac{\partial V_m}{\partial t} = \frac{-I_{ion}}{C_m} + D_x \frac{\partial^2 V_m}{\partial x^2} + D_y \frac{\partial^2 V_m}{\partial y^2} + D_z \frac{\partial^2 V_m}{\partial z^2}. \quad (1)$$

Here, V_m is the membrane potential and I_{ion} is the algebraic sum of all the currents provided by the underlying multiscale cell model of calcium handling. $C_m = 1\mu F cm^{-2}$ is the membrane capacitance of the cell. $D_x = D_y = D_z = 0.2mm^2/ms$ are the voltage diffusion coefficients in x, y and z-dimensions respectively. The solution domain is modeled as a 3D uniform grid of cardiac cells.

To solve Equation (1) we use an operator-splitting approach [17]. The equation is split into two parts: the diffusion part, which is solved using the finite difference method and the I_{ion} part which is computed by solving the cell model.

We use the stochastic multiscale calcium cycling model [5] that replicates calcium release processes at the dyadic and the whole-cell level. Moreover, we combine it with the O'Hara-Rudy (ORd) model [16] to reproduce the cardiac ventricular action potential of a healthy human heart. The multiscale cell model is shown in Figure 1.

In the model a cell consists of 10000 calcium release units or dyads arranged as a $100 \times 10 \times 10$ grid. Calcium release units are coupled by diffusion. Each dyad contains five calcium distribution compartments: myoplasm, sub-membrane space, dyadic space, network sarcoplasmic reticulum and junctional sarcoplasmic reticulum

GPGPU-11, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5647-3/18/02…$15.00
https://doi.org/10.1145/3180270.3180274

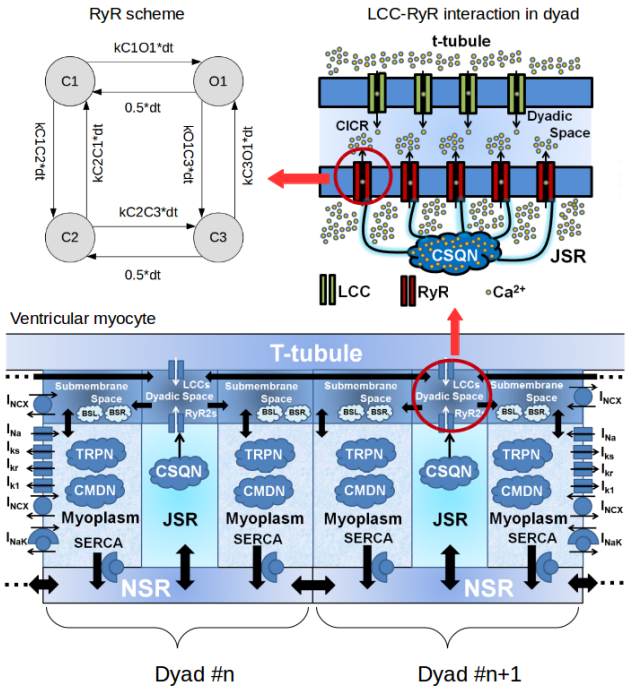

Figure 1: The multiscale myocyte model from [5]. Calcium release occurs in the dyadic space, which contains RyRs and L-type voltage gated calcium channels. Each RyR can be in one of four states at any given time.

(JSR). Calcium release occurs in the dyadic space, which consists of 15 L-type calcium channels (LCCs) and 100 RyRs.

Each RyR can be in one of four states labeled C1, C2, C3, and O1. The channel is open and calcium is released from it during the state O1. The probability of the transition from one state to another is related to the local calcium concentration. Thus, stochastic methods are needed to represent the state of calcium channels numerically.

Since L-type calcium channels and RyRs are handled similarly, we focus on the simulation of RyR state transitions. In [6] it was shown that an efficient way of obtaining the number of RyRs that changed state in the current time step is to take two samples from binomial distributions for each of the four states. We thus use the binomial distribution sampling method presented in [8].

The binomial cumulative distribution function is defined as follows:

$$F(k,n,p) = Pr(X \leq k) = \sum_{i=1}^{k} \binom{n}{k} p^k (1-p)^{n-k}, \quad \text{where } \binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{2}$$

In Equation (2), k is the number of successes in n trials with the individual probability of success p. Let x_i be the number of RyRs in the state i and k_{ij} the number of RyRs transitioning from the state i to the state j. We take one sample from binomial distribution $B(n,p)$ with $n = x_i$ and $p = p_{ij}$ to compute the number of RyRs transitioning from state i to state j. Then, we take another sample to compute the number of RyRs transitioning from state i to state l with $n = x_i - k_{ij}$ and $p = p_{il}/(1-p_{ij})$. The final number of RyRs in the next time step is obtained by adding the RyRs that transitioned from the neighbour states to the state i as shown in Equation 3. If no transition happens, the RyR remains in its original state.

$$x_i^{t+1} = x_i^t - k_{ij}^t - k_{il}^t + k_{ji}^t + k_{li}^n \tag{3}$$

Moreover, in [8] the optimized implementation of the binomial distribution sampling function was described. The distribution function is computed iteratively by subtracting from random number r. The computation stops when the smallest k satisfying the condition $r \leq F(k,n,p)$ is found.

3 IMPLEMENTATION OF THE SIMULATOR

The general structure of the program is outlined in Listing 1. On each GPU the computation is parallelized at the dyad-level. Thus, in each time step we let one CUDA thread perform the computation of one dyad. The cell voltage diffusion, which does not consume a significant amount of time, is performed on the CPU. Communication between CPU and GPU is accomplished by CUDA memory copying operations. After each time step the voltage values of the cells are copied from device to host and the updated values are copied back to the device for the next time step. The CPUs use MPI to exchange subdomain boundary voltage values among the nodes. While it would be possible to use GPUDirect [12] to exchange these values directly between the GPUs, this operation is not performance critical. We thus communicate via the CPUs for compatibility reasons.

3.1 Code Structure

As shown in the pseudo code, the CUDA initialization and memory transfers from host to device are performed before the time loop. The data needed for the computation remains on the device for the lifetime of the program. The cell computation is performed at each time step. The current voltage value is copied from host to device, then the dyad computation is executed on the device, and the computed voltage is copied back to the host. After each time step diffusion of intercellular voltage concentrations between the cells according to Equation (1) is performed.

Structure of the program

(1) Initialization
(2) Memory allocation on device
(3) Memory copy from host to device
(4) **For T time steps do:**
  (a) Cell computation
    (i) Copy voltage from host to device
    (ii) Dyad computation on the device
    (iii) Copy voltage from device to host
  (b) Cell diffusion

Listing 1: Program structure of the simulator. All cells are initialized from physiological constants stored in the code. Due to the large number of cells/dyads, only the dyad computation is performance critical. Typically T is set to 10,000 which amounts to a single heartbeat.

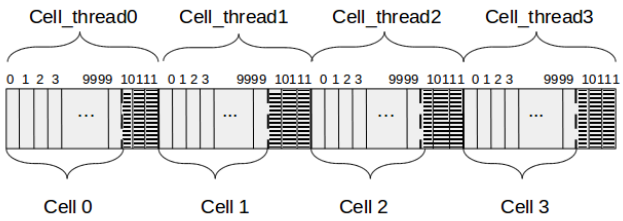

Figure 2: Data assignment to the threads. The rectangles represent threads defined via grid composition. *Cell_thread ID* **represents the threads corresponding to a block, while** *Cell ID* **represents all dyads in a cell. The thick vertical line defines the boundary from threads corresponding to two different blocks. The dashed line defines the end of cell data. Threads in the striped areas are idle.**

3.2 Thread configuration

The correct assignment of the data to the threads is important for maximizing utilization of the GPU. We arrange threads as a one dimensional (1D) grid. Consequently, data is arranged as 1D arrays. By empirical search, we found that 128 threads is the optimal block size to perform computation on cells with 10000 calcium release units on all tested architectures. This requires 79 thread blocks for a single cell since one thread performs computation of one dyad.

Most of the dyad-level computations are independent of each other, so any calculation performed on calcium release units of one cell cannot influence results of another cell. This means that computations performed by different threads do not overlap. Thus, threads that perform dyadic computations on the different cells can be placed in the same thread block.

However, execution of the functions that involve more complex computations, such as diffusion, or usage of the shared memory, such as reduction, cannot have threads corresponding to different cells in the same block, because it would cause data overlapping between cells. Thus, we choose to associate each thread block with a single cell and increase the grid accordingly. Thus, we use more threads than dyads in a cell, and the last block in each cell contains some threads that are idle during the computation as shown in Figure 2.

In our model one calcium release unit requires 29 double values. Including cell data and temporary values, one cell requires about 3 MB of data in total. Device memory generally limits the number of cells we can place on a single GPU to between 1500 and 5000, depending on the memory of the GPU. While it is possible to directly access CPU memory from the GPU, doing so severely reduces performance.

3.3 Tissue-Level parallelization

The cardiac ventricle tissue is represented as a 3D Cartesian grid of cardiac cells, which is distributed among the nodes and GPUs. Each GPU is assigned a cuboid subdomain of uniform size. We thus place one MPI process per GPU on the nodes. To perform the diffusion using a finite difference approach, each cell needs to have access to the voltage values corresponding to its neighbors in all three spatial dimensions. During each time step the computed whole-cell voltage values are transferred from device to host and the diffusion computation is performed on the CPU. However, since the global domain is decomposed into the smaller sub-domains that are assigned to the different MPI-processes, the diffusion computation is also distributed between different processes. This requires communication between the MPI-processes to provide access to the data from the neighboring cells. The communication is performed as a standard halo exchange. We introduce ghost cells at the boundary points that hold the voltage values that correspond to the neighbor cells located in the sub-domain assigned to the neigbouring process. At each time step after the diffusion computation, we exchange voltage values along the faces of the sub-domains. This is done via MPI messages [13] between the host CPUs.

4 DYAD LOOP IMPLEMENTATION

Dyad loop takes up more than 99% of the simulation time, therefore our code optimizations focus exclusively on this loop. The dyad-level computation is divided into seven parts, as detailed in Listing 2. Each part is executed on the device by launching one or more kernel functions synchronously. The different kernels are described below in detail.

For each dyad on the device do:

(1) Random number generation
(2) L-type calcium channel simulation
(3) RyR probability calculation
(4) RyR opening computation
(5) Ca concentration computation
(6) Dyad diffusion
(7) Reduction

Listing 2: Structure of the dyad computation loop. In the typical case of 10000 dyads, each of the seven kernels is launched synchronously with 79 blocks of 128 threads per cell. The last 112 threads perform no computation.

4.1 Random number generation

The sampling from binomial distributions requires drawing a randomly generated number from a uniform distribution in the range between 0 and 1. There are eight potential transitions in the RyR channels and two possible transitions in the L-type calcium channels. Therefore, ten random numbers are generated per dyad. Since in our model each cell contains 10,000 calcium release units, 100,000 random numbers per cell are generated in each time step. We precompute the necessary random numbers at each time step using the NVIDIA CUDA Random Number Generation library (cuRAND) [2], which provides a high performance GPU-accelerated random number generation. No further optimization was performed on this code.

4.2 L-type calcium channel simulation

In the naive implementation, the L-type calcium channel simulation consists of two kernels. The first calculates the L-type calcium channel state transition probabilities. This requires a small amount

of regular computation. The second computes the openings of 15 L-type calcium channels in each dyad by sampling from the binomial distribution using the calcium state transition probabilities. This operation is irregular since the number of compute steps for the sampling always depends on the random numbers generated.

In order to avoid redundant memory accesses, L-type channel transition probability calculation and opening computation were fused into the same kernel function. The computed probabilities and random numbers can thus be stored in the shared memory, which helps to lower the number of registers. This resulted in a significant performance benefit.

4.3 RyR probability calculation

The RyR probability calculation kernel computes the state transition probabilities for RyR channels and Ca, Na, and K currents. The main computation is decomposed into four smaller device functions, which are called and executed on the device.

To compute the entire cell value of the currents, we use a reduction operation. The basic approach would be to have separate kernels for the main computation and for the reduction. Because the reduction operation requires two kernel launches, in total three kernel launches would have to be performed. Since kernel launces are expensive, we merge the partial-reduction on the dyad-level with the RyR probability calculation in our implementation. Moreover, considering that the RyR opening computation is independent of the entire cell currents, we delay the final block reduction until the Ca concentration computation is executed, where another reduction takes place, and perform both reductions simultaneously. As a result, only one kernel function is launched to perform the RyR probability calculation.

The main computation requires a large number of double precision floating point division operations. The Kepler GPU uses numerically implemented division based on basic instructions, which is computationally expensive. As an alternative, CUDA provides intrinsic functions, which use fewer instructions but are less precise than the standard functions. In our implementation, we interchange division operation with multiplication by the intrinsic reciprocal rounded to the nearest value, which speeds up the computation. We found that the loss of accuracy from using these functions was negligible. Similar to the probability calculation of the L-type chanels, this kernel consists of heavy regular computations.

4.4 RyR opening computation

The RyR opening kernel function computes eight possible transitions between the four RyR states. For each state, a thread checks if there are any RyRs in that states and if the state is not empty, the thread reads two random numbers from the global memory and samples from the binomial distribution in a similar way as in the L-type channel simulation kernel function.

The binomial sampling method requires a power operation, which is computationally expensive. In our implementation, the value of the exponent always ranges from 0 to 100, which makes it cumbersome to interchange the power operation with a simple direct multiplication. It also makes this computation more irregular than the L-type opening, where the exponent is fixed at 15. To improve computational performance, it is beneficial to adopt bit-wise operations. The exponent is bit-wise compared with the smallest possible exponent, which in this case is one. If the bit exists in both operands, the power of one becomes a part of the combination of power functions. The binary left shift is performed on the smallest exponent in order to compare the next bit of the exponent. Thus, to compute the power of 100, it is enough to perform bit-wise computation seven times, because $a^{100} = a^4 \times a^{32} \times a^{64}$.

4.5 Ca concentration computation

The calcium concentration kernel computes the Ca release flux in the dyads, Na-Ca exchanger currents, Ca currents, Ca Pump currents and Ca concentration. The implementation consists of two parts: main kernel and reduction kernel. The former is decomposed into several smaller device functions in order to limit the use of registers and thus increase occupance. Doing so had a large impact on the performance of this kernel. Moreover, in order to reduce kernel launch overhead, the partial-reduction is merged with the main computation.

Elements that are common for a cell and variables that are accessed more than once by a thread are loaded into the shared memory. The power of two and division operations are substituted by multiplications. For divisions that cannot be interchanged with multiplications, we use the CUDA intrinsic functions that compute the reciprocal of a variable in round-towards-nearest mode. The main kernel contains by far the largest regular computation in the simulation. Thus, its performance is closely tied to the GPUs floating point capability.

The reduction kernel operates on partially reduced data across the blocks of threads and sends the final result as the entire cell value to the corresponding cell. The kernel function is called only once. To obviate unnecessary kernel launch overheads, the reduction over partially reduced items of the RyR probability calculation function is combined with the reduction of the Ca concentration computation. Thus, the reduction kernel operates on the Ca, Na, K, Na-Ca, Ca Pump currents and on the SR Ca release flux.

4.6 Dyad diffusion

This kernel computes the Ca diffusion between dyads of a cell. It follows the basic structure of a three-dimensional stencil computation. Two-dimensional diffusion method was adopted from [10] and [18]. The main idea is to organize threads into 2D blocks and use a loop to obtain the coordinate in the third dimension. Each thread operates on several grid points which are placed $Nx \times Ny$ distance apart, assuming that z is the slowest varying dimension. Moreover, the points (x, y, z-1) and (x, y, z) are cached into the registers and only the point (x, y, z+1) is read from the global memory at every iteration. In addition, we utilize the read-only cache since the buffer containing input grid is used only for the read operations.

To avoid checks for the x-y boundary conditions inside the loop, we introduce four distinct distances in directions $x - 1$, $x + 1$, $y - 1$ and $y + 1$. This allows us to check the boundary conditions and adjust the distances before the for-loop.

Since we use registers to cache values over z-direction, we can initialize values for the boundary $z = 0$ before the loop. For the other boundary, $z = Nz - 1$, we construct an if test inside the loop and check the boundary condition at every iteration.

Accelerator	K20	K20X	K40	P100
Architecture	Kepler	Kepler	Kepler	Pascal
# of SMs	13	14	15	56
Memory size	5 GB	6 GB	12 GB	16 GB
Peak DP, GFLOPs	1170	1310	1430	4700
Peak BW [GB/s]	208	250	288	720
STREAM [GB/s]	151	181	218	557

Table 1: An architectural overview of the K20, K20X, K40 and P100 GPU accelerators. STREAM refers to the sustained memory bandwith benchmark in [11]. All other values are specified by the vendor.

4.7 Reduction kernel

The entire cell calcium concentration is computed using a reduction operation, which passes over $O(N)$ input elements and generates $O(1)$ results. The majority of the computations corresponding to the estimation of the calcium concentration are executed on the dyad-level. To represent calcium level in a cell, we average local calcium concentrations across the dyads and assign them to the corresponding cell.

The CUDA UnBound library (CUB) [4] provides an implementation with a multitude of algorithmic strategies of the state-of-the-art algorithms in parallel for arbitrary data types and widths of parallelism. In our implementation we use the *BlockReduce* class that supports several functions which perform parallel reduction of items partitioned across the CUDA threads.

We have implemented the reduction operation as two kernel solution. For the first kernel launch, each thread contributes with two elements and use only half of the grid size for the computation. For the second kernel launch, first addition is performed during the global load, threads contribute with one element and perform reduction on already partially reduced data.

5 EXPERIMENTAL EVALUATION

5.1 Setup and test hardware

The performance was tested on three different NVIDIA Tesla GPU accelerators of the Kepler architecture, i.e. the K20, K20X, and K40, and on P100 GPUs of the new Pascal architecture. Their hardware specifications are shown in Table 1. The Kepler memory hierarchy [14] consists of shared memory, L1 and L2 caches, read-only cache, and the device memory. Each *streaming multiprocessor* (SM) has 64 KB of on-chip memory that is divided between shared memory and L1 cache. For example, the on-chip memory can be partitioned as 48 KB of the shared memory and 16 KB of the L1 cache or vice versa. The size of the L2 cache is 1536 KB (1280 KB for the K20). It is shared across the device. Each thread can access up to 255 32-bit registers, but each SM is limited to 65536 such registers. Thus, a high register count per thread limits the number of threads that can be active concurrently. For the data that is only being read, Kepler introduces a hardware-managed 48 KB read-only data cache (known as texture cache in previous GPU generations), which can be directly accessed by the SM.

In contrast to Kepler, Pascal [3] has 64 KB of dedicated shared memory on each SM. It has a combined L1 and read-only cache of 64 KB. The L2 cache has been increased to 4096 KB. The number of registers per SM remains the same as for Kepler, but due to the increased number of SMs, the total number of registers is far larger.

Shared memory is on-chip memory that can be accessed by all the threads in a thread block. It is private to each block, and it has a life-time of the block execution. Shared memory has a far higher bandwidth and a lower latency than the device memory. It allows faster access to data that is reused in the thread block and can serve as a communication channel between the threads in a block.

Data that is used as a read-only for the duration of the kernel function can be stored in the read-only cache. Like the shared memory, it is considerably faster than the device memory, but it does not require explicit management by the programmer. The compiler can automatically access data that is flaged as read-only via the read-only cache and minimize redundant access to the global memory which can benefit the performance of bandwidth-limited kernels.

For all experiments, we use a fixed time step of 0.05 ms. To discretize the diffusion terms in Equation (1), we use a fixed spatial mesh resolution of 0.5 mm. To evaluate the performance of the separate kernel functions on the device, we run the tests for a single time step. Our first set of experiments aims at benchmarking the kernels individually on the K20 GPU. To this end, each kernel was executed five times and the fastest time was selected. We measure kernel execution time only, excluding the cost of data transfer.

5.2 Code optimization impact by kernel

We first investigate the impact of the code optimizations discussed in Section 4. Our baseline is a GPU code derived from the CPU code presented in [6] via a straightforward manual code translation. The optimized code was obtained by developing and testing the optimizations described above. In Figure 3, we show the impact of these optimizations by kernel. The largest improvements are achieved on Dyad diffusion and Ca concentration computation functions. The total reduction in running time after all optimizations of these functions were 57% and 45% respectively. The running time of L-type was reduced by 35%, RyR probability calculation by 19%, RyR opening computation by 15% and Reduction by 33%. Note that the baseline code does not contain any obvious performance impediments such as unnecessary data transfers, which would result in far larger improvements.

Most of these improvements come from judicious use of shared memory and read only cache. Smaller contributions come from the use of CUDA's intrinsic reciprocals, and from the replacement of the power function that uses bit-wise operations. Numerous smaller optimizations also added up to an improvement of similar magnitude. An overview is shown in Figure 4.

5.3 Single GPU cell computation speed

A test on a single K20X GPU was performed to find out how the number of cells affects the speed of the cell computation. To determine the speed of the cell computation, we use a metric called number of cell computations per second (CC/sec), which is simply the *number of cells* × *number of time steps* divided by execution time for the entire application. Figure 5 shows the obtained results. The speed of computation increases with the number of cells, since

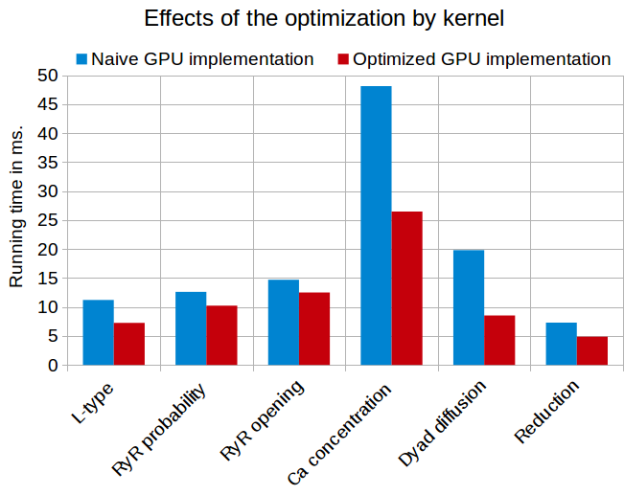

Figure 3: Comparison between naive and optimized implementations on Kepler K20.

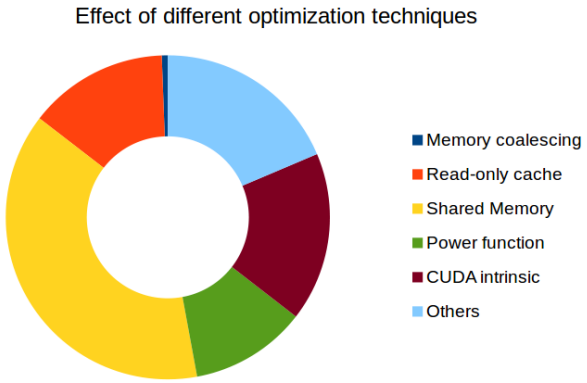

Figure 4: Impact of different optimization techniques. Note that most memory accesses are coalesced by default due to the choice of data structures even in the naive implementation.

a small number will not utilize the GPU fully. In addition, the kernel launch and other overheads are amortized over the number of cells. The same is true for the computation of the per cell calcium concentrations, hundreds of which can be run in parallel at no additional running time cost. Thus, computation for a single cell is extremely slow. The speed rises rapidly when increasing the number of cells from one to ten. Further expanding the number of cells, we increase the speed. It becomes almost stable when the number of cells reaches 1000.

5.4 Performance evaluation on different GPUs

As most kernels are memory bound, we use the attained memory bandwidth to evaluate the implementation. The results are shown in Table 2 and Figure 6. The STREAM measured bandwidth can be considered an upper bound on performance. Clearly, most kernels

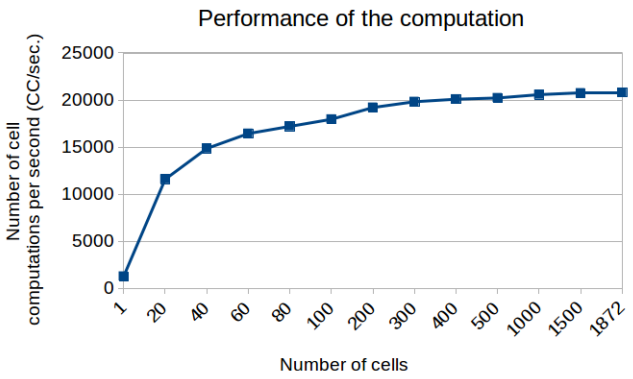

Figure 5: Speed of the cell computation on one K20X GPU. The graph shows the dependency between the performance of the computation and the number of cells on one GPU. The performance is given in CC/sec. 1872 cells is the maximum number of cells that can be contained in the memory of the K20X.

come relatively close to this performance, which implies that our implementation is close to the maximum attainable performance. Attained memory bandwidth and execution time of the individual kernels on all four GPUs are shown in Table 2 and Figure 6. Even though the achievable memory bandwidth is 20% higher on K20X than on K20, most of the kernel functions achieve no more than 15% higher bandwidth. The same behavior can be seen on K40, which has 44% higher peak memory bandwidth than K20, but for most of the individual functions, the attained bandwidth has not increased significantly. However, for the P100 performance is over 3 times higher compared to the Kepler cards, more than the increased peak bandwidth would suggest.

The Ca concentration computation benefits most from the increased computational power of the faster GPUs. Because of the large memory bandwidth and even larger computational speed on P100, the kernel execution time decreases by a factor of 4 there.

The reduction kernel function achieves 100% STREAM measured memory bandwidth on K20 and K20X, but for K40 and P100 it attains only 87%. Most of the kernels executed on K20 and K20X achieve the performance that is close to the attainable maximum. The difference between attained and STREAM measured bandwidth increases with the speed of the hardware.

As we can see, the performance on K20, K20X, and K40 do not differ substantially. The attained performance is close to 20000 CC/sec. However, simulations on P100 have shown significant improvements. The attained performance increased 3.5 times overall.

5.5 Multi-GPU scaling experiments

We verify the strong and weak scalability of our simulator on GPU-equipped clusters. We use the K20X equipped nodes of Abel, a supercomputer operated by the University of Oslo [15], and Epic, a small cluster featuring P100 cards at NTNU Trondheim. In both cases, a maximum of 8 nodes with 2 NVIDIA GPUs each is available. As usual, we run the experiments for 10,000 time steps using 10,000

Kernel function	K20	K20X	K40	P100
Random number gen.	11.27	11.14	10.05	3.95
L-type channel sim.	7.25	6.16	5.77	1.95
RyR probability calc.	10.22	8.98	8.69	2.83
RyR opening comp.	12.5	10.58	9.9	3.3
Ca concentration comp.	26.46	23.49	21.48	6.37
Dyad diffusion	8.53	7.25	6.93	2.56
Reduction	4.88	4.00	3.88	1.23
Total time	81.11	71.6	66.7	22.19
Attained performance	19950	20740	22000	69300

Table 2: Individual kernel time measurements in ms on different GPUs using 1500 cells. Performance is given in CC/sec.

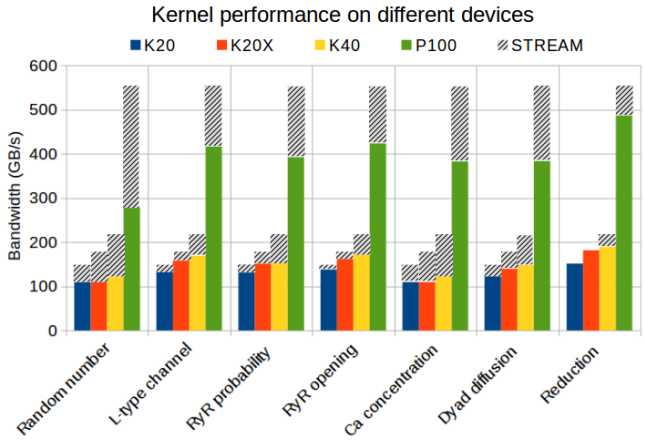

Figure 6: Individual kernel bandwidth measurements on different GPUs using 1500 cells. STREAM indicates the STREAM measured performance for different devices.

dyads per cell. Both machines are equipped with FDR InfiniBand interconnects, and we use CUDA 8.0 on both. Experiments where run using Intel icc 17.2 and Intel MPI 5.0.2 on Abel, and GCC 5.4 and OpenMPI 1.10 on Epic. The number of nodes used ranges from 1 to 8, and thus from 2 to 16 GPUs. Results for a single GPU were already discussed in Section 5.4.

The performance of weak and strong scaling tests is shown in Figure 7. For the P100 GPUs a grid of size $32 \times 16 \times 16$ cells was used on each node, which amounts to 4096 cells per GPU. The tissue size for 8 nodes thus becomes $64 \times 32 \times 32$ cells. However, as shown in Figure 5, once the number of cells per GPU is high enough, the actual number has very little impact on the performance. The weak scaling attains almost 100% efficiency in every test case due to the low amount of communication between the nodes and the expensive computation. For the strong scaling test on P100 GPUs the tissue size was fixed at $16 \times 16 \times 16$ cells, which is equivalent to that of the weak scaling experiment for a single GPU. The communication overhead becomes more visible for the larger number of compute nodes in the strong scaling experiment because each node performs

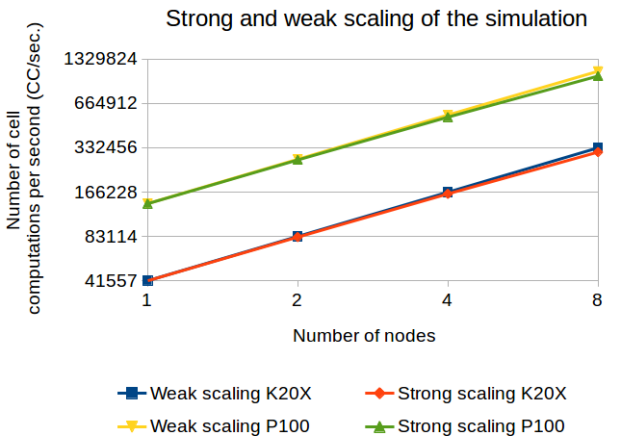

Figure 7: Performance under weak and strong scaling for P100 and K20X GPUs. Performance is given in CC/sec.

a smaller amount of the computation. In addition, the number of cells per GPU becomes so low that it affects the computation speed of the GPU, losing about 8% of its maximum performance at 8 nodes.

For K20X GPUs the experiments were performed using smaller grids of cells due to the limited amount of memory. We used 1872 cells in a $18 \times 13 \times 8$ grid per GPU for the weak scaling experiments, which constitutes the maximum feasible number. In order to keep the number of cells from getting too small, we use twice that number as a basis for the strong scaling experiment, which results in 234 cells per GPU when using all 16 GPUs. The scaling test results show the same behavior for K20X as for P100 GPUs. We see that almost the same performance is achieved using both weak and strong scaling. The linearly increasing curves indicate a good scaling for both P100 and K20X GPUs.

6 CARDIAC SIMULATIONS

In addition to testing performance, we run a series of experiments to assert the quality of the physiological simulation. In these experiments, we simulate a slab of tissue of size $12\ mm \times 12\ mm \times 12\ mm$ for different numbers of dyads in a cell. As before, the time step is set to $dt = 0.05\ ms$, the cycle length is 500 ms, and the spatial mesh resolution is $dx = dy = dz = 0.5\ mm$, which results in a total of 13,824 cells.

6.1 Paced action potential

In order to verify the model, we perform simulations of a slab of cardiac tissue under normal conditions. The y-z plane is stimulated at $t = 50\ ms$, which causes a paced action potential in a cell. The action potential is shown in Figure 8. During normal paced heart beat when the membrane voltage reaches positive values, L-type channels became active and extracellular calcium flows into the dyad. The calcium activates RyRs, which release a greater calcium flow from the calcium stores inside the sarcoplasmic reticulum.

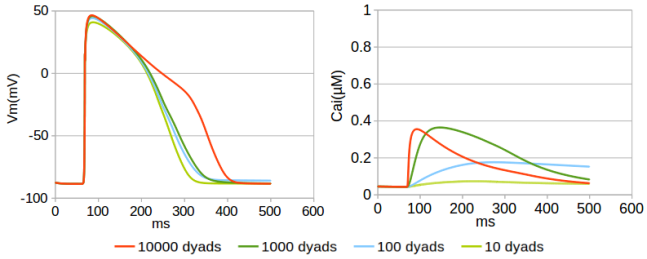

Figure 8: Paced beat of a cell under healthy conditions. The figure shows paced action potential (left) and intracellular calcium concentration (right). The tissue was stimulated at $t = 50$ *ms*, basic cycle length is 500 *ms*.

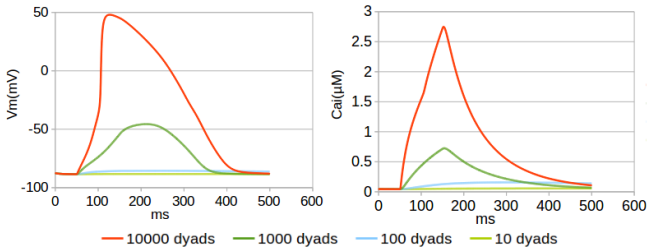

Figure 9: Spontaneous beat of a cell. The figure shows spontaneous action potential (left) and intracellular calcium concentration (right). At $t = 50$ *ms* all RyRs were open for 100 *ms*.

6.2 Spontaneous action potential

We perform tissue simulations under abnormal conditions by opening all RyRs at $t = 50$ *ms* for 100 *ms*. Open RyRs release a high calcium flow, which causes a spontaneous action potential in a cell. The spontaneous action potential is shown in Figure 9. Initially, intracellular calcium has a steep rise when there is a steep depletion of JSR. However, under abnormal conditions, we have a slower calcium release as JSR is refilled from NSR, and we have a linear increase of calcium concentration until the RyRs are closed. The calcium release activates Na-Ca exchange current, which slowly increases the voltage at the beginning. When the voltage reaches a certain threshold, L-type Ca current and fast Na currents are activated and this forms a spontaneous action potential, which can lead to arrhythmias.

6.3 Effect of the number of dyads

Figure 8 shows that under normal conditions the amount of released calcium is proportional to the number of dyads in a cell. Consequently, if a cell contains a small number of dyads, the calcium release will be small and the repolarization phase will be short.

As depicted in Figure 9, the spontaneous action potential fails when the number of dyads employed in a cell is small. Under the abnormal conditions, a great number of dyads in a cell leads to a great number of open RyRs. Thus, we have a large calcium release, shown in the plots of intracellular calcium concentration. On the other hand, a small number of dyads leads to a small calcium release, which lowers the ability to trigger an action potential in a cell. This leads to a conclusion that the accuracy of the simulation depends on the number of dyads and a cell model of 10000 dyads is required to achieve the results that comply with theoretical assumptions.

7 CONCLUSION

The general aim of using the computational models of the human heart is to develop a precise perception of the role played by disturbances of calcium handling in cardiac arrhythmias. The multiscale myocyte model [5] was adopted to develop a 3D Tissue-Scale simulator which can be used to investigate various dysfunctions of subcellular calcium release processes and action potential propagation. We performed physiologically realistic simulations in order to evaluate the selected model and show the scientific purpose of the cardiac simulator. The same model was implemented and tested on CPUs and Xeon Phis in previous research [6–9]. While these implementations showed good scaling over 400 nodes, our new results show that with careful optimizations, the Pascal generation of NVIDIA GPUs provides even better performance. The Intel Xeon Phi 7250 reached about 44,000 cell computations per second, which is about 35% less than the P100. In addition, based on the announced performance characteristics, we expect that the upcoming Volta generation will be at least another 25% faster than Pascal. Thus, the upcoming *Summit* supercomputer with more than 4,600 nodes containing 6 V100 Volta GPUs each would be capable of performing more than 2 billion cell computations per second. If data can be efficiently swapped between NVRAM, CPU, and GPU memory, detailed organ scale cardiac simulations would finally be within reach.

REFERENCES

[1] CP Adler and U Costabel. 1974. Cell number in human heart in atrophy, hypertrophy, and under the influence of cytostatics. *Recent advances in studies on cardiac structure and metabolism* 6 (1974), 343–355.
[2] Nvidia Corporation. 2016. CURAND LIBRARY Programming Guide. (2016). http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf
[3] Nvidia corporation. 2016. The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the Worlds Fastest GPU. (2016). https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[4] Nvidia corporation. 2017. CUB. (2017). https://nvlabs.github.io/cub/index.html
[5] Namit Gaur and Yoram Rudy. 2011. Multiscale modeling of calcium cycling in cardiac ventricular myocyte: macroscopic consequences of microscopic dyadic function. *Biophysical journal* 100, 12 (2011), 2904–2912.
[6] Qiang Lan, Namit Gaur, Johannes Langguth, and Xing Cai. 2015. Towards Detailed Tissue-Scale 3D Simulations of Electrical Activity and Calcium Handling in the Human Cardiac Ventricle. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 79–92.
[7] Johannes Langguth, Chad Jarvis, and Xing Cai. 2017. Porting Tissue-scale Cardiac Simulations to the Knights Landing Plattform. In *IXPUG Workshop "Experiences on Intel Knights Landing at the One Year Mark", ISC*. IEEE.
[8] Johannes Langguth, Qiang Lan, Namit Gaur, and Xing Cai. 2016. Accelerating Detailed Tissue-Scale 3D Cardiac Simulations Using Heterogeneous CPU-Xeon Phi Computing. *International Journal of Parallel Programming* (2016), 1–23.
[9] Johannes Langguth, Qiang Lan, Namit Gaur, Xing Cai, Mei Wen, and Chun-Yuan Zhang. 2016. Enabling Tissue-Scale Cardiac Simulations Using Heterogeneous Computing on Tianhe-2. In *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*. IEEE, 843–852.
[10] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*. 89–95.
[11] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.
[12] Mellanox Technologies. 2010. *NVIDIA GPUDirect Technology - Accelerating GPU-based Systems*. Technical Report.
[13] MPICH. 2017. High-Performance portable MPI. (2017). https://www.mpich.org

[14] C Nvidia. 2012. NVIDIAs next generation CUDA compute architecture: Kepler GK110. *Whitepaper* (2012).
[15] University of Oslo. 2012. University of Oslo: Abel. (2012). http://www.uio.no/english/services/it/research/hpc/abel/
[16] Thomas O'Hara, László Virág, András Varró, and Yoram Rudy. 2011. Simulation of the undiseased human cardiac ventricular action potential: model formulation and experimental validation. *PLoS Comput Biol* 7, 5 (2011), e1002061.
[17] Zhilin Qu and Alan Garfinkel. 1999. An advanced algorithm for solving partial differential equation in cardiac conduction. *IEEE Transactions on Biomedical Engineering* 46, 9 (1999), 1166–1168.
[18] Anamaria Vizitiu, Lucian Itu, Cosmin Niță, and Constantin Suciu. 2014. Optimized three-dimensional stencil computation on Fermi and Kepler GPUs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 1–6.