A Comprehensive Comparison of GPU Implementations of Cardiac Electrophysiology Models

Abouzar Kaboudian[1(✉)], Hector Augusto Velasco-Perez[1], Shahriar Iravanian[2], Yohannes Shiferaw[3], Elizabeth M. Cherry[1,4], and Flavio H. Fenton[1]

[1] Georgia Institute of Technology, Atlanta, GA 30332, USA
abouzar.kaboudian@physics.gatech.edu
[2] Emory University, Atlanta, GA 30322, USA
[3] California State University, Northridge, CA 91330, USA
[4] Rochester Institute of Technology, Rochester, NY 14623, USA

Abstract. Cardiac disease is the leading cause of death in developed countries, and arrhythmias, which are disorders in the regular generation and propagation of electrical waves that trigger contraction, form a major class of heart diseases. Computational techniques have proved to be useful in the study and understanding of cardiac arrhythmias. However, the computational cost associated with solving cardiac models makes them especially challenging to solve. Traditionally, hardware available on personal computers has been insufficient for such models; instead, supercomputers have been employed to overcome the computational costs of cardiac simulations. However, in recent years substantial advances in the computational power of graphics processing units (GPUs), combined with their modest prices and widespread availability, have made them an attractive alternative to high-performance computing using supercomputers. With greater use of GPUs, however, new challenges have emerged. GPUs must be programmed using their own languages or extensions of other languages, and, at present, there are a number of languages that support general-purpose GPU codes with substantial differences in programming ease and available levels of optimization. In this work, we present the implementation of cardiac models in several major GPU languages without language-specific optimization and compare their performance for different levels of model complexity and domain sizes.

Keywords: Cardiac electrophysiology · GPU · MATLAB · OpenACC · Python · Numba · TensorFlow · WebGL · Abubu.js · NVIDIA CUDA

1 Introduction

Cardiac disease remains the leading cause of death worldwide [1]. Ventricular fibrillation (VF), a life-threatening arrhythmia, is associated with disruption of

E. Bartocci et al. (Eds.): From Reactive Systems to Cyber-Physical Systems, LNCS 11500, pp. 9–34, 2019.
https://doi.org/10.1007/978-3-030-31514-6_2

the ventricular electrophysiological signalling that controls the contraction of the heart muscle. Such disruption manifests as spatiotemporally disorganized electrical waves [2–5] that require immediate intervention. Another form of arrhythmia that can last for years and impair the quality of life of patients is atrial fibrillation (AF). It is estimated that over 2.7–6.1 million Americans suffer from AF [1]. If untreated for prolonged periods, AF can lead to more problematic arrhythmias or even stroke.

It is possible that VF and AF treatment could be improved by designing patient-specific prevention, control and/or therapy using new computational tools that are fast, accessible and easy to use [6]. In fact, numerical simulations of cardiac dynamics are becoming increasingly important in addressing patient-specific interventions [7] and evaluating drug effects [8]. It is noteworthy that the Food and Drug Administration recently sponsored a new Cardiac Safety Research Consortium initiative (CiPA) [8,9] that specifies the use of mathematical models of cardiac cells to aid pro-arrhythmic drug risk assessment. However, as the mathematical models incorporate more detailed and sophisticated biophysical mechanisms, they are becoming extremely complex mathematically, with some of them requiring the solution of 50–100 nonlinear ordinary differential equations (ODEs) per computational cell [10]. Such ODEs typically are stiff and thus require a small temporal discretization, which is further complicated by the spatial discretization size imposed by the size of the cardiac cells. These complicating factors make cardiac dynamic simulations too large for traditional serial CPU-based computing. While some efforts have been made to create programs to aid with cardiac cell simulations in PCs [11,12], in general, scientists have used supercomputer-based high-performance implementations of cardiac models to study cardiac electrophysiology, especially for large two- and three-dimensional tissues. However, supercomputers are expensive to acquire and hard to maintain, and even when such resources are managed by individuals other than the end users, users typically are required to submit their programs for execution as batch jobs, which can be inconvenient.

Substantial advances in the computational power of graphic processing units (GPUs) have made them an attractive alternative to traditional high-performance computing. Currently available GPUs are equipped with thousands of powerful computational cores, and they can be acquired at affordable prices sometimes as low as a few hundred US dollars. As such, they can provide high-performance computing on personal computers at merely a fraction of the cost of traditional CPU-based supercomputers. However, GPUs require machine code that is prepared for the specific target GPU hardware. Thus, computer codes either need to be implemented in a special language that is intended for GPU programming or should be modified such that they become suitable for execution on GPUs. At present, there are several languages and programming solutions that enable implementation of GPU applications. As might be expected, each solution and programming language has certain benefits and may perform differently for different applications. Therefore, a comprehensive study focused on comparing the ease of programming and performance of such programming

languages and solutions when applied to cardiac models can be beneficial to help researchers in the cardiac community choose the appropriate approach.

In this study, we investigate some of the major languages and solutions in cardiac GPU computing. Specifically, we consider (1) GPU computing solutions available in MATLAB, (2) the pragma-based approach of OpenACC, (3) Python-Numba, (4) TensorFlow, (5) WebGL 2.0, and (6) NVIDIA CUDA together with the Abubu.js library. Our comparisons will be based on implementations without any substantial program-specific optimization. Of course, we expect that applying language-specific optimizations could improve performance. However, it is fair to assume that most cardiac researchers are not necessarily experts in GPU programming, so that in many cases the solution that would provide the best performance with minimal effort would be ideal. Nevertheless, our comparisons will help users with a broad range of programming expertise make informed choices about GPU implementations for cardiac models.

2 Methods

2.1 Models

We will compare performance using three different models with different complexity. The FitzHugh-Nagumo (FHN) model [13,14] is a two-variable model used as a generic excitable media model and in some cases as a cardiac model. Tuning the model's parameters can change features like the trajectory of the spiral wave tip [15].

The Minimal Model (MM) [16] is a four-variable model developed to reproduce many important properties of cardiac cells while also prioritizing computational tractability. The model includes a variable representing voltage as well as three gating variables that govern the dynamics of summary sodium and calcium currents; a time-independent potassium current also is included. Different parameterizations of the MM have been shown to reproduce the dynamics of other models with good fidelity [6,16–18].

The Beeler-Reuter (BR) model [19] is an eight-variable model that includes sodium, calcium, and potassium currents. It was the first model developed to simulate ventricular tissue and the first to include an intracellular calcium concentration. We made modifications to the BR model by speeding up the τ_f and τ_d in the model to 50% of their original value to prevent the model from breakup [20]. If the original model was used with the default parameter set, it would gives rise to spiral wave breakup in two dimensions [20,21]. This is also the first model for which it was shown that reaction-diffusion equations for cardiac cells can produce spiral waves in 2D [21].

2.2 Numerical Methods

The cardiomyocytes' membrane potential (V) propagation through gap junctions (and in neurons through synapses) can be modeled by a cable equation [22], which is given by

$$\partial_t V(\boldsymbol{x}, t) = \nabla \cdot (\tilde{D} \nabla V) - \frac{I_{total}}{C_m}. \tag{1}$$

Here, the membrane potential diffuses with a diffusion coefficient $\tilde{D}$ (which represents the fiber orientation of the heart [23,24] and, in general, is anisotropic and heterogeneous), while the ionic concentrations are local in cardiac as well as neuronal tissues. The transmembrane currents for all ions as well as the ion pumps and exchangers are included in $I_{\text{total}} = \sum I_i(V, y_i)$. The most general form of a transmembrane current I_i permeable to ion i is simply $I_i = g_i(V - E_i)$, where g_i is a conductance term, V is the membrane potential or voltage, and E_i is the Nernst potential for ion species i. Often, the conductance is calculated using gates following the Hodgkin-Huxley [22] formalism, in which the conductance term is decomposed into the product of a maximal conductance term and one or more separate normalized variables that represent the probability of finding the channel open, which typically depends on the membrane potential or an ion concentration. These variables follow first-order differential equations of the form

$$\frac{dy_i(t)}{dt} = \alpha_{y_i}(V)(1 - y_i) - \beta_{y_i}(V) y_i \tag{2}$$

where α_{y_i} is the probability that the channel gate y_i will transition from closed to open and β_{y_i} is the probability it will transition from open to closed; both probabilities are a function of voltage. An alternative representation used in some models is achieved through Markov chains, where each state s_p follows a differential equation of the form

$$\frac{ds_p(t)}{dt} = \sum_{q=1, q \neq p}^{n} (k_{qp} s_q - k_{pq} s_p), \tag{3}$$

where k_{qp} is the transition rate from state s_q to s_p. With either formulation, the ordinary differential equations become partial differential equations once a spatially extended system, rather than a single cell, is considered. More details on how to numerically integrate these equations including convergence and boundary conditions can be found in Ref. [25].

In all cases, we used a domain size of $20 \times 20\,\text{cm}$. The diffusion coefficient $\tilde{D}$ was assumed to be isotropic and homogeneously defined over the domain. Finite differences were used for numerical simulations. To discretize the spatial term in Eq. (1), a second-order central difference scheme was used both in the x and y directions. All ODEs were solved using the forward Euler time-stepping scheme for most variables. As an exception, the time-integration of the Hudgkin-Huxley-type gates in Eq. (2) used the Rush-Larsen time-stepping scheme [26]. In all cases, a uniform Cartesian grid was employed. The grid sizes used were 256×256, 512×512, 1024×1024, and 2048×2048. This implies that for smaller grid sizes, the solution was not fully numerically resolved. However, we emphasize that our objective was to compare the same solution obtained under different conditions. This would guarantee that for the same model, we deal with the same

loading conditions on the GPU cores. The time step was chosen as $\Delta t = 0.05$ ms up to a grid size of 2048×2048, where $\Delta t = 0.01$ ms was employed instead to satisfy the CFL condition. Our initial conditions were set to the resting state of the cells everywhere in the domain, except for nodes with $x < 1$ cm to create a traveling wave toward the right-hand side of the domain. Later, at $t = 600$ for the FHN model and at $t = 370$ ms for the MM and BR models, a depolarizing wave is applied at the bottom half of the domain where $y < 10$ cm by changing the transmembrane potential to a higher depolarizing potential. This voltage was set to 1.0 for the FHN and MM model and 30 mV for the BR model. For more information on the implementation details, see the computer codes that can be downloaded from http://abouzar.net/SmolkaFest2019/codes.zip (Fig. 1).

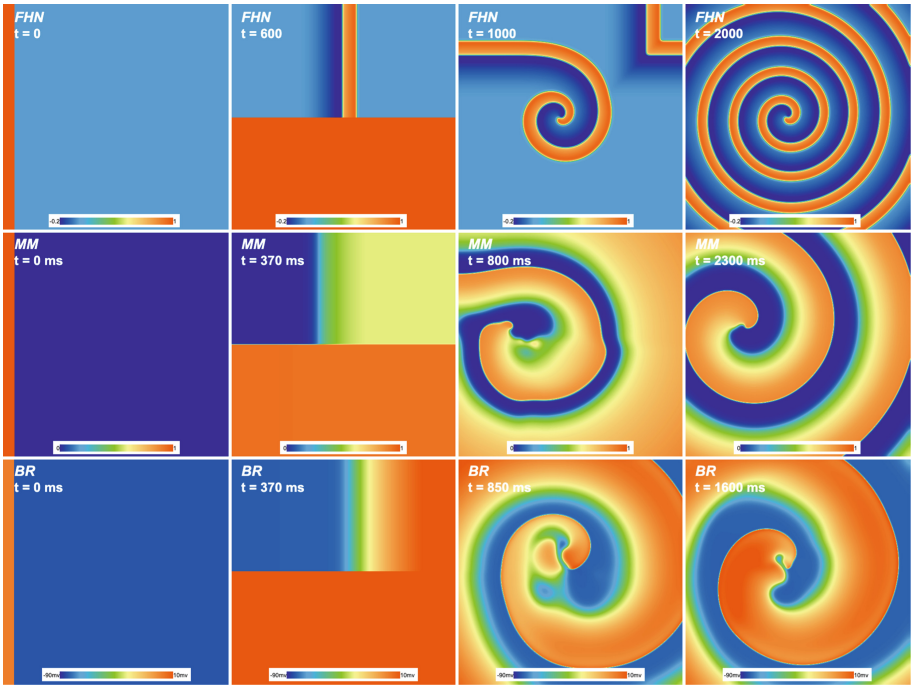

Fig. 1. Membrane potential for the FHN (first row), MM (second row), and BR (third row) models at the initial time (first column), application of the depolarizing voltage from the bottom half of the domain (second column), transient spiral wave dynamics (third column), and after the spiral wave stabilizes (fourth column).

Because the details of GPU programming are closely connected with the different implementations studied, this information is provided below in the next section.

3 Comparison of GPU Implementations

Below, we describe six different GPU implementations of the three models (FHN, MM, and BR). In some cases, we also compare additional options available for a particular configuration. Along with measurements of speedup as a function of the number of grid points, we also comment on ease of programming.

First, we implemented a serial version of all three models in the C programming language. The PGI-C compiler was used to generate the machine code. This serial version was used in all speedup calculations. The speedup was defined as follows:

$$\text{Speedup} = \frac{\text{wall-time of single-core serial CPU C-program}}{\text{wall-time of GPU implementation}}, \tag{4}$$

where wall-time is the measured time of execution of the program that an ordinary wall-clock would measure, albeit here, we used the computer's clock for measurements.

All measurements were carried out on a Linux Manjaro operating system with Kernel version 4.19.34. The system had an AMD® Ryzen threadripper 2990wx 32-core processor that was used for CPU time measurements (although only one core was used in the CPU case). The graphics card that was used for GPU measurements was a NVIDIA TITAN V/PCIe/SSE2.

In this study, all measurements were carried out in double precision, except for the WebGL 2.0 and TensorFlow cases. For the CUDA and OpenACC implementations, we tried single-precision calculations and the speedups did not change more than 10% on this GPU.

3.1 MATLAB

MATLAB, originally meaning matrix laboratory, is a proprietary programming language developed by MathWorks. MATLAB allows for easy matrix operations and is equipped with several linear solvers, as well as built-in plotting and visualization features, that together make it very popular for general programming in academic settings. MATLAB's easy-to-learn programming syntax makes it attractive to novice programmers, and its feature-rich environment makes it attractive to seasoned programmers, for both prototyping algorithms and research. Additionally, MATLAB has an interactive user interface that combined with MATLAB's interpreter removes the hurdles of compiling, running, and visualization of the data. As such, MATLAB has been adopted as the companion language or the language of choice in several books [27–35]. MATLAB is also widely used in several research fields, including but not limited to, fluid mechanics [36,37], geophysical studies [38–40], volcanology [41–44], astrophysics [45–48], chemical engineering [49–52], image analysis [53–57], neural networks [58–60], cell modeling [61–64], and cardiac studies [65–71]. MATLAB also provides GPU parallelism through fully automated GPU acceleration, the arrayfun command which applies a function to each element of arrays, and CUDA kernel calls.

Here, we implemented the arrayfun and CUDA kernel call options for each of the three models. The arrayfun function applies a MATLAB function to all elements of an array. After sending the arrays to the GPU using the gpuArray() function, calling the arrayfun function for each time step allows the function to be run on the GPU. MATLAB's interpreter recognizes that it can run the function independently for each element of the array on the GPU and it will do so. Since this approach still relies on automatic detection of the parallelizable section and acceleration of the code, it is expected to be less than "ideal". The second approach in MATLAB is to manually write the GPU code as a CUDA kernel and run the CUDA kernel. This approach is supposed to result in the best observed performance since there is no "guess-work" necessary by the MATLAB interpreter and the GPU code is already parsed. The upside is that all MATLAB visualizaton and data analysis tools still can be used, and the CUDA kernel will only be in charge of running the accelerated code in an optimum way. However, writing CUDA kernels requires familiarity with the NVIDIA CUDA C language in addition to familiarity with MATLAB. Hence, it is expected that a smaller number of MATLAB users will be comfortable programming CUDA kernels. Speedup is assessed for problems sizes of 2^{16}, 2^{18}, 2^{20}, and 2^{22} grid points. By default, all implementations in MATLAB use double-precision variables.

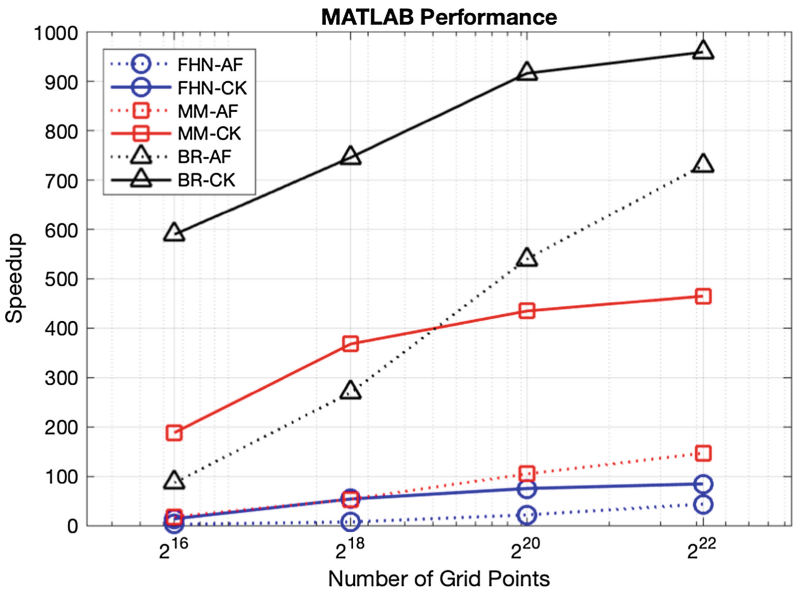

Fig. 2. Speedup of models vs. grid size for the FHN, MM, and BR models using MATLAB with array functions (dashed lines) and with CUDA kernels (solid lines).

Figure 2 shows that in all cases, as expected, the use of CUDA kernels provided more substantial speedup than the corresponding arrayfun implementations by as much as a factor of six. The largest speedup was found for the BR

model, which is not surprising given that it has the most equations and thus the most potential for concurrency within a given time iteration. Correspondingly, the FHN model attained the smallest speedups, but it still achieved a speedup of nearly two orders of magnitude for the largest grid size using CUDA kernels. The MM achieved speedups more than twice that of the FHN model, most likely because although it has twice the number of ODEs, it has a significant number of additional algebraic equations evaluated during each time step, thus allowing greater potential for performance increase through greater parallelization over each time step.

3.2 OpenACC

OpenACC is a programming standard that developed as a joint effort between Cray, CAPS, NVIDIA and PGI as an alternative to low-level CUDA programming. OpenACC, similar to OpenMP, uses a pragma-based approach to identify the computer code regions that can be parallelized on the GPU. The pragma directives, together with environment variables and library calls, facilitate accelerating regions of the serial C/C++ or FORTRAN CPU codes that can benefit from parallelization, typically loops, and in this case support use of GPGPU computing. As a result, OpenACC provides an approach that can accelerate mature CPU C/C++ or FORTRAN codes with minimal effort. This feature has made OpenACC an attractive choice to a large group of researchers in various fields including but not limited to fluid mechanics [72–76], earthquake modeling [77], deep neural networks [78,79], astrophysics and data mining [80], cardiovascular [81] and cardiac electrophysiology [82,83].

First, we implemented a serial version of all three models in the C programming language. This serial version was used in all speedup calculations. OpenACC pragmas were added to the serial code to achieve parallelism. After initializing the solution we used the OpenACC's "data in" pragmas to copy the data to the GPU. Then in the parallel loops these arrays were marked as present on the GPU to avoid unnecessary copy of the arrays in and out of the GPU. The data was copied out to CPU memory only on the time-steps that we intended to write data to disk. This was achieved through the "update self" pragma. The data was written to disk only for debugging and during performance measurements no data was written to disk. Both single-precision and double-precision implementations were tested and the variations in performance were limited to less than 10% on this particular GPU. The results presented here were generated using double-precision variables.

Figure 3 shows the resulting speedups, which are quite similar to the speedups obtained using MATLAB with CUDA kernels. Again, the BR model benefited the most from acceleration, with speedups of up to three orders of magnitude due to the fact that, for the BR model, the computational cost of the reaction operations is much more than for the diffusion term. This suggests that the more complicated models of cardiac dynamics can benefit even further from the use of OpenACC implementations.

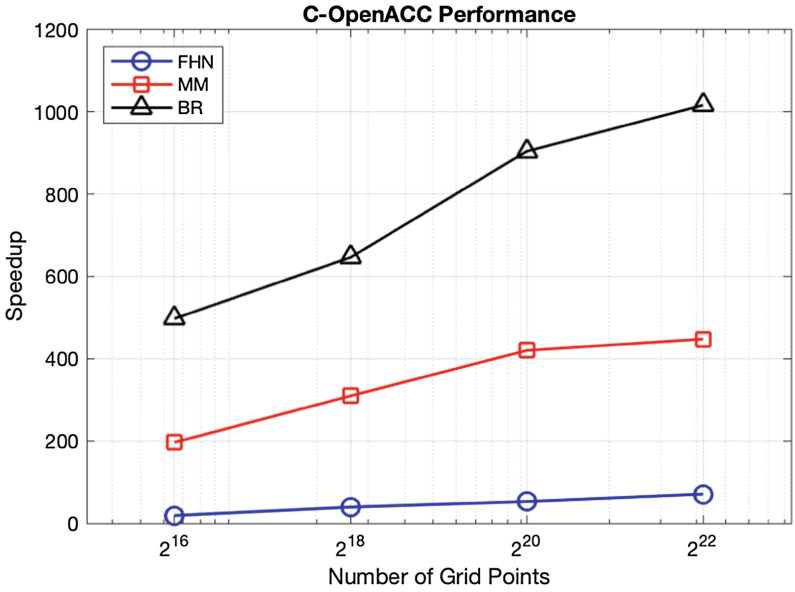

Fig. 3. Speedup of models vs. grid size for the FHN, MM, and BR models using OpenACC directives in the C programming language.

3.3 Python Numba Implementation

Python is an interpreted general-purpose language created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands as a successor of ABC [84]. Python supports multiple programming paradigms, including functional, object-oriented, and procedural programming. Due to its feature-rich environment and its approachable learning curve, Python is widely popular as a language for teaching [85,86] and research in various field such as astrophysics [87–89], machine learning [90,91], neural networks [92], and business [93]. Similarly, Python is also used in cardiovascular [94–101], cardiac electrophysiology [102–107], and arrhythmia detection [108] studies. Python's popularity is evident in the large number of conferences that are held each year dedicated to Python programming including DjangoCon Europe, EuroPython, EuroSciPy, Kiwi PyCon, O'Reilly Open Source Convention, Plone Conference, PyCon conferences held in different regions of the world, PyData, PyGotham, SiPy and many more [109]. Project Jupyter has also contributed significantly to the popularity of the Python language by providing a web application that allows users to create and share documents that contain live interactive Python codes, equations, visualization and narrative text [110].

The widespread popularity of Python has resulted in a broad range of Python libraries that can be used for various different applications. One very popular library is NumPy, which provides support for definition of multi-dimensional array objects and array operations [111], which can be very useful in

scientific computing. Numba is an open-source Just-In-Time (JIT) compiler that translates a subset of Python and NumPy code into accelerated machine code [112]. The Numba compiler can provide acceleration through multicore CPUs or GPGPU. Numba has a very simple approach to accelerating Python code. In fact, the Numba website provides a tutorial that teaches Python programmers to start accelerating their Python code in as little as five minutes [113]. This small learning curve makes Numba an attractive choice for cardiac modeling.

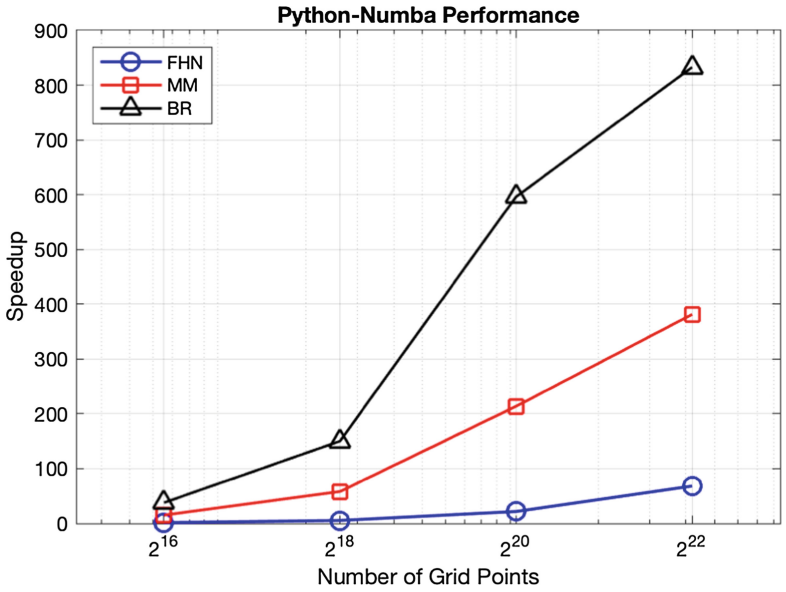

Fig. 4. Speedup of models vs. grid size for the FHN, MM, and BR models using Python Numba.

Figure 4 shows speedup results using Python-Numba. All measurements were carried out using double-precision variables. As expected, the BR model achieves the greatest speedup and the FHN model the least. Most notable is that large performance gains do not appear until grid sizes of 2^{20}, which may be due to the fact that for each time step there is a certain overhead time imposed for launching a parallel code on the GPU. However, that effect becomes less important when we keep the GPU busy for longer periods before advancing to the next time step.

3.4 TensorFlow Implementation

TensorFlow is a free and open-source software library primarily designed by the Google Brain team [114,115] for internal use. It was released for public use later under Apache 2.0 open-source licence on November 9, 2015 [116]. It has

been used by a number of companies, including Airbnb, AIRBUS, Coca-Cola, Google, Intel, PayPal and Qualcomm [117] in both research and production. TensorFlow can be used on single as well as multiple CPUs and GPUs. Once the target device is chosen, the parallelization is carried out automatically by the TensorFlow engine. TensorFlow provides extensive features to be used for machine learning and deep neural network applications [114,115,118]. Hence, it can be considered an attractive choice for model-based machine-learning environments where machine-learning algorithms can be trained using a dynamical numerical model. A number of groups in the cardiac community have embraced TensorFlow [119–127].

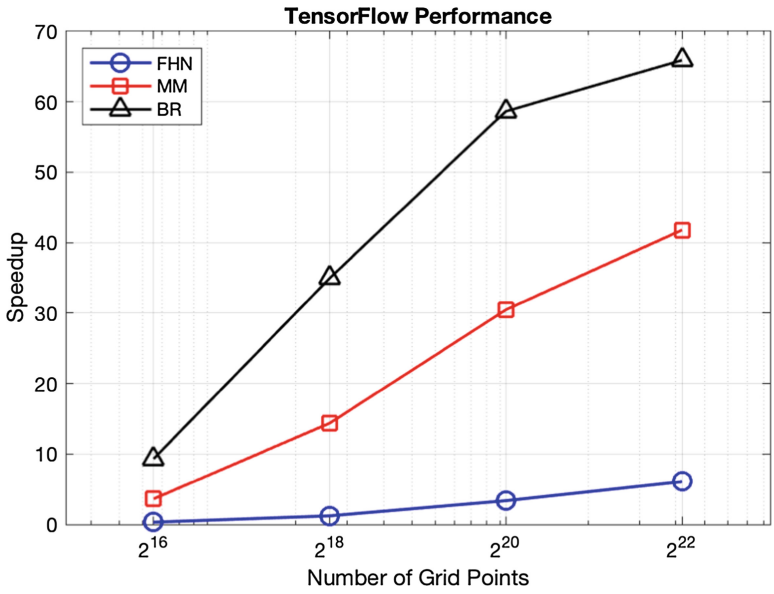

Fig. 5. Speedup of models vs. grid size for the FHN, MM, and BR models using TensorFlow.

Speedup results for the three models using Tensor-flow are given in Fig. 5. The measurements for TensorFlow were made using single-precision variables, as some of the functions did not have a double-precision implementation for GPU parallelism at the time of coding the TensorFlow programs. In this case, speedup is quite limited compared to the other approaches considered, with the maximum speedup (attained for the BR model on the largest grid) still well below 100. The speedups are the result of just choosing the target device to be the GPU. No optimization such as using convolutions was used here. We would say the effort required for parallelism on the GPU was minimal compared to other languages. Given the minimal effort required for achieving parallelism, programmers are encouraged to use the GPU as their target device for all models, especially more complex ones.

3.5 WebGL and Abubu.js Implementation

WebGL or the Web Graphics Library is a royalty-free JavaScript application programming interface standard that provides low-level 2D and 3D rendering capabilities in modern web browsers without the need to install any plug-ins through the HTML5 canvas element [128]. This means that WebGL applications can run on any modern web browser and on any major operating system (such as Microsoft Windows, macOS, Linux, Android, or even iOS), and at the same time harness the computational power of the available GPU on that device. WebGL applications are automatically compiled at run-time for the particular user's graphic cards. Therefore, the WebGL applications do not need to be compiled by the developers for all the intended GPUs and operating systems. This also means that WebGL applications are capable of harnessing the computational power in various GPU devices from various vendors, unlike some of the languages such as NVIDIA CUDA, which can only run on specific hardware.

The heart of the WebGL applications is written in OpenGL Shading Language (GLSL), which is a high-level programming language with a syntax based on the C programming language [129]. GLSL supports most of the C/C++ familiar structural components, such as if statements, for loops, etc. It also has a number of built-in functions for mathematical, vector, and matrix operations as well as texture access [130]. The only drawback for using WebGL is that currently it only supports single-precision variables and textures. Therefore, for applications that must use double-precision floats, WebGL is not suitable at present. When using WebGL for parallelism, usually texture memory is utilized as the basic data structure for the input and output of the programs [131]. However, the WebGL language can have a high learning curve for novice programmers or those who are not well versed in graphics programming. The Abubu.js programming library is used to address this issue and remove the hurdles of GPGPU programming with WebGL [6]. Using Abubu.js, WebGL has been shown to be capable of solving a wide range of problems from studying fractals, solitons and chaos [132] to cardiac dynamics, fluid mechanics, and crystal growth [6].

In this work, we followed the methods proposed in [6] to implement the FHN, the MM, and the BR model in WebGL using Abubu.js. Figure 6 shows performance gains using our WebGL implementation, which generally outperforms all other implementations. In particular, the speedup for the MM is now above 1000 for the largest grid size, and for the BR model speedup exceeds 2000 for a grid size of 2^{20}. However, performance for the BR model is more variable, with a dropoff in speedup at the largest grid size in contrast to monotonic increases with grid size in all other cases. In addition, WebGL performance for the smallest grid size is typically no greater than that seen in OpenACC and MATLAB with CUDA kernels. That is due to the fact that there is a minimum overhead time to launch the WebGL applications in each time step. The performance drop in the BR model for larger grid sizes could be due to memory access bottlenecks and how the data is stored on the GPU. It should be noted that even with the performance drops, the WebGL applications outperform all other implementations by a large margin for larger domains.

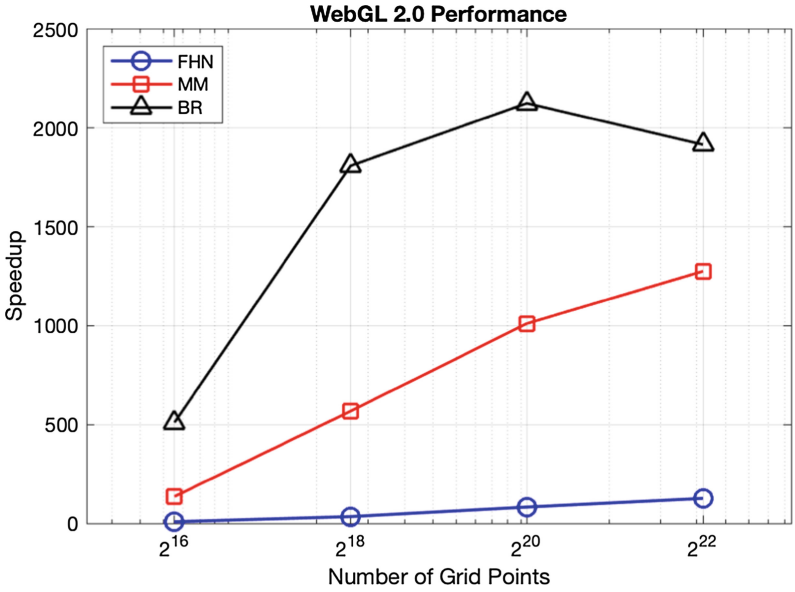

Fig. 6. Speedup of models vs. grid size for the FHN, MM, and BR models using WebGL together with the Abubu.js library.

3.6 NVIDIA CUDA Implementation

One of the most popular platforms to solve PDEs in parallel using GPUs is CUDA. CUDA is a parallel platform developed by NVIDIA that allows the user to execute programs on the GPU of a personal computer. This allows faster processing and visualization of large data sets that fulfill certain characteristics that will be discussed below. Since its launch in 2007, CUDA has helped to extend the use of GPU technology to the scientific community. Specifically, the CUDA platform has been applied in several scientific and engineering fields such as fluid dynamics [133,134], machine learning and neural networks [135–138], astrophysics [139–142], the Lattice Boltzmann method [143–145], molecular dynamics [146–148], clinical applications [149,150], and recently in the cardiac modeling community [151–155]. CUDA has also been successfully used for teaching purposes, including in undergraduate workshops [156]. Like all computational tools, it has advantages and disadvantages.

The CUDA platform is an extension of other programming languages, i.e., it is a set of functions added to a preexisting platform that allows the user to communicate with the GPU. This implies that most of the base language characteristics and logic will be inherited by the parallel functions. There are several versions of CUDA, mainly C/CUDA (CUDA for the C language), PyCUDA (CUDA for Python) and CUDA Fortran. We decided to develop our solvers in C/CUDA because it is the most supported version. The description below is valid regardless of the version chosen.

In some cases, CUDA is able to launch millions of processing threads simultaneously, which can increase the speed of computations and save many hours and possibly days of processing time (). The speedup depends mainly on the type of algorithm implemented to process the data and the structure of the data. To understand better how these factors affect the speed of the computations, it is important to understand the interaction between the software and hardware, particularly the interaction between the CPU (commonly referred to as the host) and the GPU (commonly referred to as the device). All programs start at the host level, meaning that they are all managed by the host and all the data is held in CPU RAM or the hard drive. Meanwhile, sections of memory in the GPU are reserved to hold the data that needs to be processed. Once everything is ready, the CPU calls specific functions to be executed on the GPU. After the GPU processes the data, it must be sent back to the CPU so that it can be post-processed by the user. In most programs, there is a constant exchange of data between host and device. As a rule of thumb, the programmer should try to reduce the number of memory transactions between both ends mainly due to bandwidth limitations. Other factors to be considered are the GPUs memory capacity and frequency of kernel calls (functions called by the CPU that execute on the GPU and hold the bulk of the processing algorithm). In addition to adequately controlling the data flow, the program must manage the data in a parallel-friendly arrangement, specifically, we must determine the way that data will be read and written. As commonly observed in programming languages with arrays of two or more dimensions, the data layout and memory access patterns need to be aligned to achieve maximum performance. More specifically, CUDA requires the data layout to adapt to a single instruction multiple data processing structure, which means that all processing threads must be performing the same instructions simultaneously to avoid thread divergence.

In addition to the memory transactions and layout mentioned above, CUDA requires the programmer to adapt the data to a specific hierarchical structure of threads. In general, threads are organized into blocks, which can be one-, two-, or three-dimensional. Sets of blocks are then organized in a grid. Again, the grid can be arranged in all three dimensions. More information can be found in [157] and [158]. The dimension of these objects refers to how they will be accessed, not how they are physically stored in memory. The user can adapt this structure to increase the performance of their computations. In our particular case, two-dimensional blocks and grid resemble very well the 2D domains in which we are solving the PDEs. Still, different memory access patterns will influence the speed of our computations. Other factors to be considered when building a CUDA program are coalesced memory patterns, in which multiple threads can receive data through a single combined memory access, and the use of the various types of internal memories and the interaction among them. These are just some of many considerations that are important to keep in mind. It is also worth noting that if a task is not inherently parallelizable due to dependencies across loop iterations without substantial work within each iteration (such as a Fibonacci sequence calculation) or if the number of threads is small (typically on the order

of hundreds or lower), CUDA will perform worse than most standard serial implementations due to overhead associated with launching kernels and moving data between the host and device.

In our implementations, we used global memory. Both single- and double-precision implementations were tested and the variations in performance were limited to less than 10% on this particular GPU. The results presented here are generated using double-precision variables. One-dimensional arrays were used to represent the 2D domain. The data could be arranged in either a row-major or a column-major fashion in the one-dimensional arrays. In the row-major structure, the matrix is stored in the 1D array one row after another until the entire matrix is stored. In the column-major order, the same procedure is followed for the matrix columns. Both versions were tested to observe the performance differences. The row-major version of the data-structures performed consistently better than the column-major structures. This could be due to the fact that the row-major structure was more compatible with hardware, possibly due to the way that warps are organized on this GPU. Different results might be expected for different GPUs and the users should be aware of such differences. It should be noted that this should not be confused with the loop access of multi-dimensional arrays in CPUs. Here, in both cases, the data structures are one-dimensional and the central difference algorithm for the diffusion term imposes a symmetry condition on both directions.

We also decided against using shared-memory implementations such as those suggested in earlier studies [152]. The use of shared memory requires copying the variables from the global memory into shared memory, performing calculations from shared memory in registered memory (implicit), then writing data into shared memory, and then to global memory. These steps are required in each time step as no data can be retained between time steps. However, the use of global memory would require bringing the data to register for calculations and writing the data back to global memory. It is evident that using global memory for this type of problems involves fewer memory transactions compared to the shared memory implementations but the same number of global memory accesses and thus is expected to be faster. Additionally, our goal in this study is to compare the simplest implementations in each language as the targeted programmers are scientists whose primary expertise is not GPU programming. The use of texture memory instead of one-dimensional arrays could also change the performance of the applications. However, any performance improvements could be hardware-dependent and would also depend on the problem size and complexity. In favor of simplicity, we chose the use of a global memory implementation.

We used a 16×16 thread size for the CUDA implementations, and each direction was then divided by 16 to get the block size. Smaller thread sizes led to lower performance and larger thread sizes did not improve the performance on our particular GPU.

Figure 7 shows the speedup achieved for each of the three models using our CUDA implementation. CUDA slightly outperforms MATLAB with CUDA kernels and OpenACC, especially for smaller grids, but overall the performance

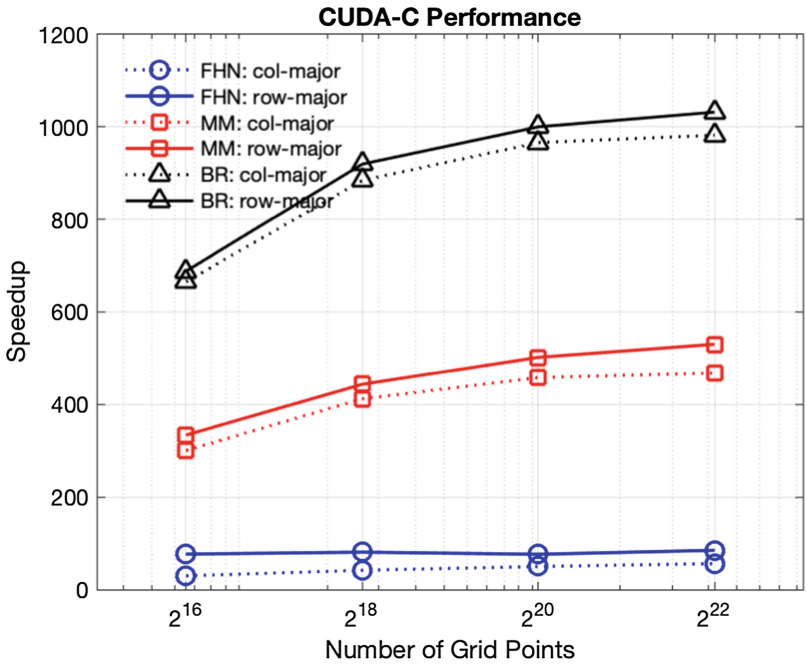

Fig. 7. Speedup of models vs. grid size for the FHN, MM, and BR models using CUDA with column-major data structures in dashed lines and row-major data structures in solid lines.

is fairly comparable for these three implementations. WebGL maintains better performance for all grid sizes. Note that speedup seems to have saturated for the FHN model and appears to be close to saturating for the other models. In addition, it should be noted that we could potentially observe a performance saturation similar to those observed in WebGL implementations. Moreover, due to the limitations in the GPU memory size, there is a limit to the problem size that can be handled on a single GPU so that using multiple GPUs for larger problems becomes inevitable. While using multiple GPUs can be useful for handling larger domains due to memory constraints, it should be noted the required communication between the multiple GPUs will impose performance penalties on the parallel GPU codes.

4 Discussion and Conclusion

Figure 8 shows the comparison between the speedup gains for each of the GPU implementations of the three different models with different grid sizes. It can be seen that the WebGL applications outperform all other implementations for all cases except for the smallest grid sizes and the FHN model. As soon as the workload on the GPU is "large" enough to take full advantage of concurrency, WebGL provides the best performance. All implementations performed

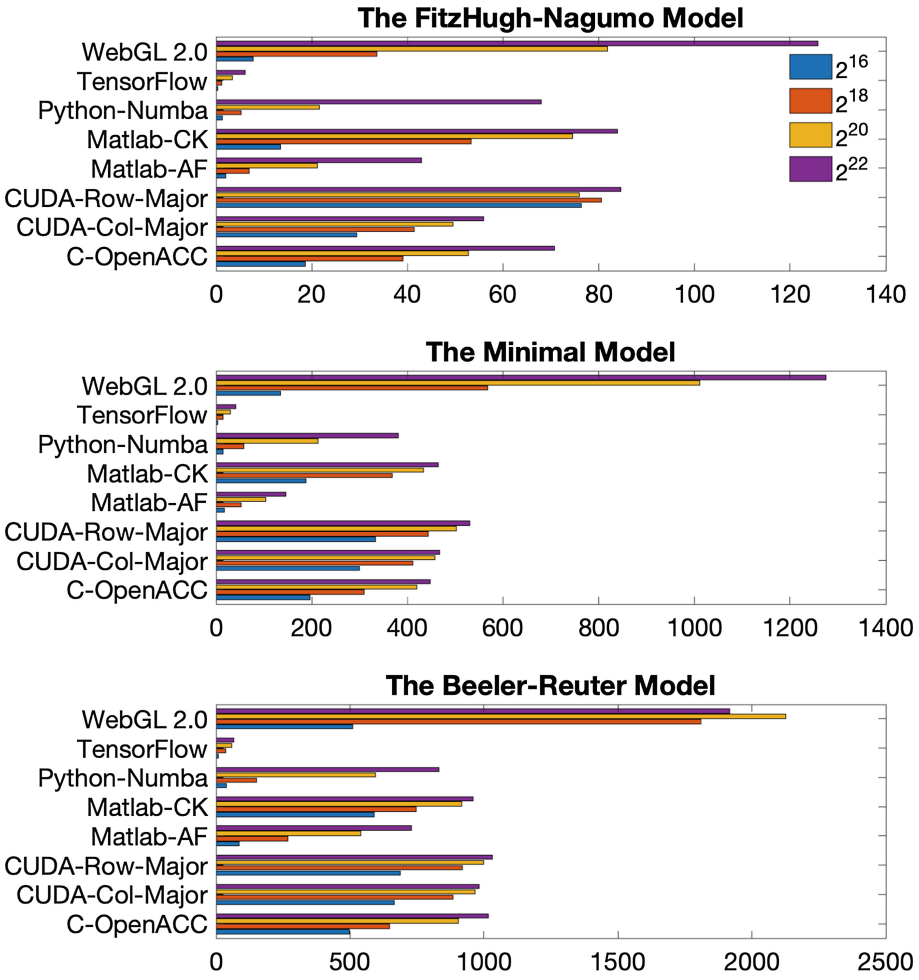

Fig. 8. Speedup comparison for various implementations of the FHN, MM, and BR models. Each color corresponds to a different grid size.

better with larger grid sizes and more complicated models, with the BR model implementations providing the best performances among all models. Another notable observation is that almost all GPU implementations provided performance comparable to that of the NVIDIA CUDA implementations with minor differences with the exception of TensorFlow. Therefore, we can conclude that almost all languages considered in this study are ready to make effective use of GPU hardware to reduce program runtimes. The least effort for achieving parallelism in the languages was required by TensorFlow, C-OpenACC, MATLAB arrayfun, and then Python Numba implementations. However, writing the serial code in TensorFlow was the most convoluted of all the approaches tested.

Nevertheless, moving from the serial code to the accelerated GPU code was as simple as just choosing the target device. C-OpenACC was the most natural for a novice programmer, which could provide the best performance with the least programming effort. However, MATLAB and Python Numba provide built-in visualization tools.

Acknowledgements. This work was supported in part by the National Science Foundation under grants CNS-1446312(EMC) and by CMMI-1762553 (FHF and AK). EMC, AK, YS, and FHF, also collaborated while at Kavli Institute for Theoretical Physics (KITP) and thus research was also supported in part by NSF Grant No. PHY-1748958, NIH Grant No. R25GM067110, and the Gordon and Betty Moore Foundation Grant No. 2919.01.

References

1. Benjamin, E.J., et al.: Heart disease and stroke statistics-2017 update: a report from the American Heart Association. Circulation **135**(10), e146–e603 (2017)
2. Winfree, A.: Electrical turbulence in three-dimensional heart muscle. Science **266**(5187), 1003–1006 (1994)
3. Gray, R.A., et al.: Mechanisms of cardiac fibrillation. Science **270**(5239), 1222–1226 (1995)
4. Gray, R.A., Pertsov, A.M., Jalife, J.: Spatial and temporal organization during cardiac fibrillation. Nature **392**(6671), 75 (1998)
5. Fenton, F.H., Cherry, E.M., Glass, L.: Cardiac arrhythmia. Scholarpedia **3**(7), 1665 (2008)
6. Kaboudia, A., Cherry, E.M., Fenton, F.H.: Real-time interactive simulations of large-scale systems on personal computers and cell phones. Sci. Adv. **5**, eaav6019 (2019)
7. Zahid, S., et al.: Feasibility of using patient-specific models and the "minimum cut" algorithm to predict optimal ablation targets for left atrial flutter. Heart Rhythm **13**(8), 1687–1698 (2016)
8. Dutta, S., et al.: Optimization of an in silico cardiac cell model for proarrhythmia risk assessment. Front. Physiol. **8**, 616 (2017)
9. Cavero, I., Holzgrefe, H.: CiPA: ongoing testing, future qualification procedures, and pending issues. J. Pharmacol. Toxicol. Methods **76**, 27–37 (2015)
10. Fenton, F.H., Cherry, E.M.: Models of cardiac cell. Scholarpedia **3**(8), 1868 (2008)
11. Fenton, F.H., Cherry, E.M., Hastings, H.M., Evans, S.J.: Real-time computer simulations of excitable media: JAVA as a scientific language and as a wrapper for C and FORTRAN programs. Biosystems **64**(1–3), 73–96 (2002)
12. Barkley, D.: EZ-spiral: a code for simulating spiral waves (2002). https://homepages.warwick.ac.uk/~masax/. Accessed 29 Apr 2019
13. FitzHugh, R.: Impulses and physiological states in theoretical models of nerve membrane. Biophys. J. **1**(6), 445–466 (1961)
14. Nagumo, J., Arimoto, S., Yoshizawa, S.: An active pulse transmission line simulating nerve axon. Proc. IRE **50**(10), 2061–2070 (1962)
15. Winfree, A.T.: Varieties of spiral wave behavior: an experimentalist's approach to the theory of excitable media. Chaos Interdiscip. J. Nonlinear Sci. **1**(3), 303–334 (1991)

16. Bueno-Orovio, A., Cherry, E.M., Fenton, F.H.: Minimal model for human ventricular action potentials in tissue. J. Theor. Biol. **253**(3), 544–560 (2008)
17. Cherry, E.M., Ehrlich, J.R., Nattel, S., Fenton, F.H.: Pulmonary vein reentry–properties and size matter: insights from a computational analysis. Heart Rhythm **4**(12), 1553–1562 (2007)
18. Lombardo, D.M., Fenton, F.H., Narayan, S.M., Rappel, W.-J.: Comparison of detailed and simplified models of human atrial myocytes to recapitulate patient specific properties. PLoS Comput. Biol. **12**(8), e1005060 (2016)
19. Beeler, G.W., Reuter, H.: Reconstruction of the action potential of ventricular myocardial fibres. J. Physiol. **268**(1), 177–210 (1977)
20. Courtemanche, M.: Complex spiral wave dynamics in a spatially distributed ionic model of cardiac electrical activity. Chaos Interdiscip. J. Nonlinear Sci. **6**(4), 579–600 (1996)
21. Courtemanche, M., Winfree, A.T.: Re-entrant rotating waves in a beeler-reuter based model of two-dimensional cardiac electrical activity. Int. J. Bifurc. Chaos **1**(02), 431–444 (1991)
22. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. J. Physiol. **117**(4), 500–544 (1952)
23. Streeter Jr., D.D., Spotnitz, H.M., Patel, D.P., Ross Jr., J., Sonnenblick, E.H.: Fiber orientation in the canine left ventricle during diastole and systole. Circ. Res. **24**(3), 339–347 (1969)
24. Peskin, C.S.: Fiber architecture of the left ventricular wall: an asymptotic analysis. Commun. Pure Appl. Math. **42**(1), 79–113 (1989)
25. Ji, Y.C., Fenton, F.H.: Numerical solutions of reaction-diffusion equations: application to neural and cardiac models. Am. J. Phys. **84**(8), 626–638 (2016)
26. Rush, S., Larsen, H.: A practical algorithm for solving dynamic membrane equations. IEEE Trans. Biomed. Eng. **4**, 389–392 (1978)
27. Halpern, D., Wilson, H.B., Turcotte, L.H.: Advanced Mathematics and Mechanics Applications using MATLAB. Chapman and Hall/CRC, Boca Raton (2002)
28. Pozrikidis, C.: Introduction to Finite and Spectral Element Methods Using MATLAB. CRC Press, Boca Raton (2005)
29. Quarteroni, A., Saleri, F., Gervasio, P.: Scientific Computing with MATLAB and Octave, vol. 2. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-642-45367-0
30. Strang, G.: Computational Science and Engineering, vol. 791. Wellesley-Cambridge Press, Wellesley (2007)
31. Aarnes, J.E., Gimse, T., Lie, K.-A.: An introduction to the numerics of flow in porous media using MATLAB. In: Hasle, G., Lie, K.A., Quak, E. (eds.) Geometric Modelling, Numerical Simulation, and Optimization, pp. 265–306. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-68783-2_9
32. Li, J., Chen, Y.-T.: Computational Partial Differential Equations Using MATLAB. Chapman and Hall/CRC, Boca Raton (2008)
33. Anderson, D., Tannehill, J.C., Pletcher, R.H.: Computational Fluidmechanics and Heat Transfer. CRC Press, Boca Raton (2016)
34. Elsherbeni, A.Z. Demir, V.: The finite-difference time-domain method for electromagnetics with MATLAB simulations. The Institution of Engineering and Technology (2016)
35. Kwon, Y.W., Bang, H.: The Finite Element Method Using MATLAB. CRC Press, Boca Raton (2018)

36. Martin, N., Gorelick, S.M.: MOD_FreeSurf2D: a Matlab surface fluid flow model for rivers and streams. Comput. Geosci. **31**(7), 929–946 (2005)
37. Gholami, A., Bonakdari, H., Zaji, A.H., Akhtari, A.A.: Simulation of open channel bend characteristics using computational fluid dynamics and artificial neural networks. Eng. Appl. Comput. Fluid Mech. **9**(1), 355–369 (2015)
38. Irving, J., Knight, R.: Numerical modeling of ground-penetrating radar in 2-D using MATLAB. Comput. Geosci. **32**(9), 1247–1258 (2006)
39. Tzanis, A., et al.: MATGPR: a freeware MATLAB package for the analysis of common-offset GPR data. In: Geophysical Research Abstracts, vol. 8 (2006)
40. Xuan, C., Channell, J.E.: UPmag: MATLAB software for viewing and processing u channel or other pass-through paleomagnetic data. Geochem. Geophys. Geosyst. **10**(10), 1–12 (2009). https://doi.org/10.1029/2009GC002584
41. Lesage, P.: Interactive MATLAB software for the analysis of seismic volcanic signals. Comput. Geosci. **35**(10), 2137–2144 (2009)
42. Battaglia, M., Cervelli, P.F., Murray, J.R.: dMODELS: a MATLAB software package for modeling crustal deformation near active faults and volcanic centers. J. Volcanol. Geoth. Res. **254**, 1–4 (2013)
43. Charpentier, I., Sarocchi, D., Sedano, L.A.R.: Particle shape analysis of volcanic clast samples with the MATLAB tool MORPHEO. Comput. Geosci. **51**, 172–181 (2013)
44. Valade, S., Harris, A.J., Cerminara, M.: Plume ascent tracker: interactive MATLAB software for analysis of ascending plumes in image data. Comput. Geosci. **66**, 132–144 (2014)
45. Ofek, E.O.: MATLAB package for astronomy and astrophysics. Astrophysics Source Code Library (2014)
46. Ahmad, I., Raja, M.A.Z., Bilal, M., Ashraf, F.: Bio-inspired computational heuristics to study lane-emden systems arising in astrophysics model. SpringerPlus **5**(1), 1866 (2016)
47. Loredo, T., Scargle, J.: Time series exploration in Python and MATLAB: unevenly sampled data, parametric modeling, and periodograms. In: AAS/High Energy Astrophysics Division, vol. 17 (2019)
48. Kamel, N.A., Selman, A.A.-R.: Automatic detection of sunspots size and activity using MATLAB. Iraqi J. Sci. **60**(2), 411–425 (2019)
49. Guo, G.: Electromechanical feed control system in chemical dangerous goods production. Chem. Eng. Trans. **71**, 1039–1044 (2018)
50. Esche, E., Bublitz, S., Tolksdorf, G., Repke, J.-U.: Automatic decomposition of nonlinear equation systems for improved initialization and solution of chemical engineering process models. Comput. Aided Chem. Eng. **44**, 1387–1392 (2018)
51. Fitzpatrick, D., Ley, S.V.: Engineering chemistry for the future of chemical synthesis. Tetrahedron **74**(25), 3087–3100 (2018)
52. Eastep, C.V., Harrell, G.K., McPeak, A.N., Versypt, A.N.F.: A MATLAB app to introduce chemical engineering design concepts to engineering freshmen through a pharmaceutical dosing case study. Chem. Eng. Educ. **53**(2), 85 (2019)
53. Svoboda, T., Kybic, J., Hlavac, V.: Image Processing, Analysis and Machine Vision-A MATLAB Companion. Thomson Learning, Toronto (2007)
54. Dhawan, A.P.: Medical Image Analysis, vol. 31. Wiley, Hoboken (2011)
55. Schneider, C.A., Rasband, W.S., Eliceiri, K.W.: NIH image to imagej: 25 years of image analysis. Nat. Methods **9**(7), 671 (2012)

56. Rodríguez-Cristerna, A., Gómez-Flores, W., de Albuquerque-Pereira, W.C.: BUSAT: a MATLAB toolbox for breast ultrasound image analysis. In: Carrasco-Ochoa, J.A., Martínez-Trinidad, J.F., Olvera-López, J.A. (eds.) MCPR 2017. LNCS, vol. 10267, pp. 268–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59226-8_26
57. Cho, J.I., Wang, X., Xu, Y., Sun, J.: LISA: a MATLAB package for longitudinal image sequence analysis. arXiv preprint arXiv:1902.06131 (2019)
58. Vedaldi, A., Lenc, K.: MatConvNet: convolutional neural networks for MATLAB. In: Proceedings of the 23rd ACM international conference on Multimedia, pp. 689–692. ACM (2015)
59. Novikov, A., Podoprikhin, D., Osokin, A., Vetrov, D.P.: Tensorizing neural networks. In: Advances in Neural Information Processing Systems, pp. 442–450 (2015)
60. Vardhana, M., Arunkumar, N., Lasrado, S., Abdulhay, E., Ramirez-Gonzalez, G.: Convolutional neural network for bio-medical image segmentation with hardware acceleration. Cogn. Syst. Res. **50**, 10–14 (2018)
61. Molitor, S.C., Tong, M., Vora, D.: Matlab-based simulation of whole-cell and single-channel currents. J. Undergrad. Neurosci. Educ. **4**(2), A74 (2006)
62. Cardin, J.A., et al.: Driving fast-spiking cells induces gamma rhythm and controls sensory responses. Nature **459**(7247), 663 (2009)
63. Kirkton, R.D., Bursac, N.: Engineering biosynthetic excitable tissues from unexcitable cells for electrophysiological and cell therapy studies. Nat. Commun. **2**, 300 (2011)
64. Kodandaramaiah, S.B., Franzesi, G.T., Chow, B.Y., Boyden, E.S., Forest, C.R.: Automated whole-cell patch-clamp electrophysiology of neurons in vivo. Nat. Methods **9**(6), 585 (2012)
65. Prassl, A.J., et al.: Automatically generated, anatomically accurate meshes for cardiac electrophysiology problems. IEEE Trans. Biomed. Eng. **56**(5), 1318–1330 (2009)
66. O'Hara, T., Virág, L., Varró, A., Rudy, Y.: Simulation of the undiseased human cardiac ventricular action potential: model formulation and experimental validation. PLoS Comput. Biol. **7**(5), e1002061 (2011)
67. Cusimano, N., Bueno-Orovio, A., Turner, I., Burrage, K.: On the order of the fractional laplacian in determining the spatio-temporal evolution of a space-fractional model of cardiac electrophysiology. PLoS ONE **10**(12), e0143938 (2015)
68. Elshrif, M.M., Shi, P., Cherry, E.M.: Representing variability and transmural differences in a model of human heart failure. IEEE J. Biomed. Health Inform. **19**(4), 1308–1320 (2015)
69. Passini, E., et al.: Human in silico drug trials demonstrate higher accuracy than animal models in predicting clinical pro-arrhythmic cardiotoxicity. Front. Physiol. **8**, 668 (2017)
70. Cusimano, N., del Teso, F., Gerardo-Giorda, L., Pagnini, G.: Discretizations of the spectral fractional laplacian on general domains with Dirichlet, Neumann, and Robin boundary conditions. SIAM J. Numer. Anal. **56**(3), 1243–1272 (2018)
71. Handa, B.S., et al.: Interventricular differences in action potential duration restitution contribute to dissimilar ventricular rhythms in ex-vivo perfused hearts. Front. Cardiovasc. Med. **6**, 34 (2019)
72. Kraus, J., Schlottke, M., Adinetz, A., Pleiter, D.: Accelerating a C++ CFD code with OpenACC. In: 2014 First Workshop on Accelerator Programming using Directives, pp. 47–54. IEEE (2014)

73. Blair, S., Albing, C., Grund, A., Jocksch, A.: Accelerating an MPI lattice Boltzmann code using OpenACC. In: Proceedings of the Second Workshop on Accelerator Programming Using Directives, p. 3. ACM (2015)
74. Huismann, I., Stiller, J., Fröhlich, J.: Two-level parallelization of a fluid mechanics algorithm exploiting hardware heterogeneity. Comput. Fluids **117**, 114–124 (2015)
75. Lou, J., Xia, Y., Luo, L., Luo, H., Edwards, J.R., Mueller, F.: OpenACC directive-based GPU acceleration of an implicit reconstructed discontinuous Galerkin method for compressible flows on 3D unstructured grids. In: 54th AIAA Aerospace Sciences Meeting, p. 1815 (2016)
76. Raj, A., Roy, S., Vydyanathar, N., Sharma, B.: Acceleration of a 3D immersed boundary solver using OpenACC. In: 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW), pp. 65–73. IEEE (2018)
77. Gallovic, F., Valentova, L., Ampuero, J.-P., Gabriel, A.-A.: Bayesian dynamic finite-fault inversion: 1. Method and synthetic test (2019)
78. Kan, G., He, X., Ding, L., Li, J., Liang, K., Hong, Y.: A heterogeneous computing accelerated SCE-UA global optimization method using OpenMP, OpenCL, CUDA, and OpenACC. Water Sci. Technol. **76**(7), 1640–1651 (2017)
79. Liu, C., Yang, H., Sun, R., Luan, Z., Qian, D.: SWTVM: exploring the automated compilation for deep learning on Sunway architecture. arXiv preprint arXiv:1904.07404 (2019)
80. Cavuoti, S., et al.: Astrophysical data mining with GPU. A case study: genetic classification of globular clusters. New Astron. **26**, 12–22 (2014)
81. Rosenberger, S., Haase, G.: Pragma based GPU parallelizations for cardiovascular simulations. In: 2018 International Conference on High Performance Computing and Simulation (HPCS), pp. 1022–1027. IEEE (2018)
82. Campos, J., Oliveira, R.S., dos Santos, R.W., Rocha, B.M.: Lattice boltzmann method for parallel simulations of cardiac electrophysiology using GPUs. J. Comput. Appl. Math. **295**, 70–82 (2016)
83. Canal Noguer, P.: Modeling human atrial electrodynamics and arrhythmias through GPU parallel computing: from cell to tissue. B.S. thesis, Universitat Politècnica de Catalunya (2016)
84. History and license. https://docs.python.org/3/license.html. Accessed 28 Apr 2019
85. Fangohr, H.: A comparison of C, MATLAB, and Python as teaching languages in engineering. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3039, pp. 1210–1217. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25944-2_157
86. Goldwasser, M.H., Letscher, D.: Teaching an object-oriented CS1-: with Python. ACM SIGCSE Bull. **40**, 42–46 (2008)
87. Vallisneri, M., Kanner, J., Williams, R., Weinstein, A., Stephens, B.: The LIGO open science center. J. Phys. Conf. Ser. **610**, 012021 (2015)
88. Rodriguez, C.L., Morscher, M., Pattabiraman, B., Chatterjee, S., Haster, C.-J., Rasio, F.A.: Binary black hole mergers from globular clusters: implications for advanced LIGO. Phys. Rev. Lett. **115**(5), 051101 (2015)
89. Aasi, J., et al.: Advanced LIGO. Class. Quantum Gravity **32**(7), 074001 (2015)
90. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
91. Raschka, S.: Python Machine Learning. Packt Publishing Ltd., Birmingham (2015)
92. Goodman, D.F., Brette, R.: Brian: a simulator for spiking neural networks in Python. Front. Neuroinform. **2**, 5 (2008)

93. Yen, D.C., Huang, S.-M., Ku, C.-Y.: The impact and implementation of XML on business-to-business commerce. Comput. Stand. Interfaces **24**(4), 347–362 (2002)
94. Lehmann, G., et al.: Towards dynamic planning and guidance of minimally invasive robotic cardiac bypass surgical procedures. In: Niessen, W.J., Viergever, M.A. (eds.) MICCAI 2001. LNCS, vol. 2208, pp. 368–375. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45468-3_44
95. Lehmann, G., Habets, D., Holdsworth, D.W., Peters, T., Drangova, M.: Simulation of intra-operative 3D coronary angiography for enhanced minimally invasive robotic cardiac intervention. In: Dohi, T., Kikinis, R. (eds.) MICCAI 2002. LNCS, vol. 2489, pp. 268–275. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45787-9_34
96. Azaouzi, M., Makradi, A., Petit, J., Belouettar, S., Polit, O.: On the numerical investigation of cardiovascular balloon-expandable stent using finite element method. Comput. Mater. Sci. **79**, 326–335 (2013)
97. Herman, W.H., et al.: Early detection and treatment of type 2 diabetes reduce cardiovascular morbidity and mortality: a simulation of the results of the Anglo-Danish-Dutch study of intensive treatment in people with screen-detected diabetes in primary care (addition-Europe). Diabetes Care **38**(8), 1449–1455 (2015)
98. Itani, M.A., et al.: An automated multiscale ensemble simulation approach for vascular blood flow. J. Comput. Sci. **9**, 150–155 (2015)
99. Ramachandra, A.B., Kahn, A.M., Marsden, A.L.: Patient-specific simulations reveal significant differences in mechanical stimuli in venous and arterial coronary grafts. J. Cardiovasc. Transl. Res. **9**(4), 279–290 (2016)
100. Updegrove, A., Wilson, N.M., Merkow, J., Lan, H., Marsden, A.L., Shadden, S.C.: Simvascular: An open source pipeline for cardiovascular simulation. Ann. Biomed. Eng. **45**(3), 525–541 (2017)
101. Kuo, S., Ye, W., Duong, J., Herman, W.H.: Are the favorable cardiovascular outcomes of empagliflozin treatment explained by its effects on multiple cardiometabolic risk factors? A simulation of the results of the EMPA-REG OUTCOME trial. Diabetes Res. Clin. Pract. **141**, 181–189 (2018)
102. Myers, C.R., Sethna, J.P.: Python for education: computational methods for nonlinear systems. Comput. Sci. Eng. **9**(3), 75–79 (2007)
103. Niederer, S.A., et al.: Verification of cardiac tissue electrophysiology simulators using an N-version benchmark. Philos. Trans. R. Soc. A Math. Phys. Eng. Sci. **369**(1954), 4331–4351 (2011)
104. Burton, R.A., et al.: Optical control of excitation waves in cardiac tissue. Nat. Photonics **9**(12), 813 (2015)
105. Hurtado, D.E., Castro, S., Gizzi, A.: Computational modeling of non-linear diffusion in cardiac electrophysiology: a novel porous-medium approach. Comput. Methods Appl. Mech. Eng. **300**, 70–83 (2016)
106. Cherubini, C., Filippi, S., Gizzi, A., Ruiz-Baier, R.: A note on stress-driven anisotropic diffusion and its role in active deformable media. J. Theor. Biol. **430**, 221–228 (2017)
107. Gizzi, A., et al.: Nonlinear diffusion and thermo-electric coupling in a two-variable model of cardiac action potential. Chaos Interdiscip. J. Nonlinear Sci. **27**(9), 093919 (2017)
108. Mane, R.S., Cheeran, A., Awandekar, V.D., Rani, P.: Cardiac arrhythmia detection by ECG feature extraction. Int. J. Eng. Res. Appl. **3**, 327–332 (2013)
109. Conferences and workshops. https://www.python.org/community/workshops/. Accessed 28 Apr 2019

110. Project jupyter. https://jupyter.org/. Accessed 29 Apr 2019
111. Numpy. https://www.numpy.org/. Accessed 29 Apr 2019
112. Numba. http://numba.pydata.org/. Accessed 29 Apr 2019
113. 1.1. a 5 minute guide to Numba. http://numba.pydata.org/numba-doc/latest/user/5minguide.html. Accessed 29 Apr 2019
114. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 265–283 (2016)
115. Girija, S.S.: TensorFlow: large-scale machine learning on heterogeneous distributed systems (2016). Software tensorflow.org
116. Google just open sourced TensorFlow, its artificial intelligence engine. https://www.wired.com/2015/11/google-open-sources-its-artificial-intelligence-engine/. Accessed 29 Apr 2019
117. Case studies and mentions. https://www.tensorflow.org/about/case-studies/. Accessed 29 Apr 2019
118. Cheng, H.-T., et al.: Wide & deep learning for recommender systems. In: Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, pp. 7–10. ACM (2016)
119. Lieman-Sifry, J., Le, M., Lau, F., Sall, S., Golden, D.: FastVentricle: cardiac segmentation with ENet. In: Pop, M., Wright, G.A. (eds.) FIMH 2017. LNCS, vol. 10263, pp. 127–138. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59448-4_13
120. Warrick, P., Homsi, M.N.: Cardiac arrhythmia detection from ECG combining convolutional and long short-term memory networks. In: 2017 Computing in Cardiology (CinC), pp. 1–4. IEEE (2017)
121. Biswas, S., Aggarwal, H.K., Poddar, S., Jacob, M.: Model-based free-breathing cardiac MRI reconstruction using deep learned & storm priors: MoDL-storm. In: 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6533–6537. IEEE (2018)
122. Kamaleswaran, R., Mahajan, R., Akbilgic, O.: A robust deep convolutional neural network for the classification of abnormal cardiac rhythm using single lead electrocardiograms of variable length. Physiol. Meas. **39**(3), 035006 (2018)
123. Iravanian, S.: fib-tf: a TensorFlow-based cardiac electrophysiology simulator. J. Open Source Softw. **3**(26), 719 (2018)
124. Kang, S.-H., Joe, B., Yoon, Y., Cho, G.-Y., Shin, I., Suh, J.-W.: Cardiac auscultation using smartphones: pilot study. JMIR mHealth uHealth **6**(2), e49 (2018)
125. Savalia, S., Emamian, V.: Cardiac arrhythmia classification by multi-layer perceptron and convolution neural networks. Bioengineering **5**(2), 35 (2018)
126. Teplitzky, B.A., McRoberts, M.: Fully-automated ventricular ectopic beat classification for use with mobile cardiac telemetry. In: 2018 IEEE 15th International Conference on Wearable and Implantable Body Sensor Networks (BSN), pp. 58–61. IEEE (2018)
127. Bello, G.A., et al.: Deep-learning cardiac motion analysis for human survival prediction. Nat. Mach. Intell. **1**(2), 95 (2019)
128. WebGL overview. https://www.khronos.org/webgl/. Accessed 29 Apr 2019
129. OpenGL shading language. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language. Accessed 29 Apr 2019
130. WebGL 2.0 API quick reference guide. https://www.khronos.org/files/webgl20-reference-guide.pdf. Accessed 29 Apr 2019
131. Owens, J.D., et al.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26–1**, 80–113 (2007)

132. Kaboudian, A., Cherry, E.M., Fenton, F.H.: Large-scale interactive numerical experiments of chaos, solitons and fractals in real time via GPU in a web browser. Chaos Solitons Fractals **121**, 6–29 (2019)
133. Jacobsen, D., Thibault, J., Senocak, I.: An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In: 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, p. 522 (2010)
134. Ladický, L., Jeong, S., Solenthaler, B., Pollefeys, M., Gross, M.: Data-driven fluid simulations using regression forests. ACM Trans. Graph. **34**(6), 1–9 (2015)
135. Jang, H., Park, A., Jung, K.: Neural network implementation using CUDA and OpenMP. In: 2008 Digital Image Computing: Techniques and Applications, pp. 155–161. IEEE (2008)
136. Nageswaran, J.M., Dutt, N., Krichmar, J.L., Nicolau, A., Veidenbaum, A.V.: A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Netw. **22**(5–6), 791–800 (2009)
137. Sierra-Canto, K., Madera-Ramirez, F., Uc-Cetina, V.: Parallel training of a back-propagation neural network using CUDA. In: 2010 Ninth International Conference on Machine Learning and Applications, pp. 307–312. IEEE (2010)
138. Ciresan, D.C., Meier, U., Masci, J., Gambardella, L.M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: Twenty-Second International Joint Conference on Artificial Intelligence (2011)
139. Nylons, L.: Fast n-body simulation with CUDA (2007)
140. Belleman, R.G., Bédorf, J., Zwart, S.F.P.: High performance direct gravitational n-body simulations on graphics processing units II: an implementation in cuda. New Astron. **13**(2), 103–112 (2008)
141. Glinskiy, B.M., Kulikov, I.M., Snytnikov, A.V., Romanenko, A.A., Chernykh, I.G., Vshivkov, V.A.: Co-design of parallel numerical methods for plasma physics and astrophysics. Supercomput. Front. Innov. **1**(3), 88–98 (2015)
142. Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., Taiji, M.: 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–12. IEEE (2009)
143. Obrecht, C., Kuznik, F., Tourancheau, B., Roux, J.-J.: A new approach to the lattice Boltzmann method for graphics processing units. Comput. Math. Appl. **61**(12), 3628–3638 (2011)
144. Rinaldi, P.R., Dari, E., Vénere, M.J., Clausse, A.: A lattice-Boltzmann solver for 3D fluid simulation on GPU. Simul. Model. Pract. Theory **25**, 163–171 (2012)
145. Obrecht, C., Kuznik, F., Tourancheau, B., Roux, J.-J.: Scalable lattice Boltzmann solvers for cuda GPU clusters. Parallel Comput. **39**(6–7), 259–270 (2013)
146. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. **227**(10), 5342–5359 (2008)
147. Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W.: Accelerating molecular dynamics simulations using graphics processing units with CUDA. Comput. Phys. Commun. **179**(9), 634–641 (2008)
148. Stone, J.E., Vandivort, K.L., Schulten, K.: GPU-accelerated molecular visualization on petascale supercomputing platforms, pp. 1–8 (2013)

149. Reichl, T., Passenger, J., Acosta, O., Salvado, O.: Ultrasound goes GPU: real-time simulation using CUDA. In: Medical Imaging 2009: Visualization, Image-Guided Procedures, and Modeling, vol. 7261, p. 726116, International Society for Optics and Photonics (2009)
150. Alsmirat, M.A., Jararweh, Y., Al-Ayyoub, M., Shehab, M.A., Gupta, B.B.: Accelerating compute intensive medical imaging segmentation algorithms using hybrid CPU-GPU implementations. Multimedia Tools Appl. **76**(3), 3537–3555 (2017)
151. Dawes, T.J.W., et al.: Machine learning of three-dimensional right ventricular motion enables outcome prediction in pulmonary hypertension: a cardiac MR Imaging Study. Radiology **283**(2), 161315 (2017)
152. Bartocci, E., Cherry, E.M., Glimm, J., Grosu, R., Smolka, S.A., Fenton, F.H.: Toward real-time simulation of cardiac dynamics. In: Proceedings of the 9th International Conference on Computational Methods in Systems Biology, pp. 103–112. ACM (2011)
153. Berg, S., Luther, S., Parlitz, U.: Synchronization based system identification of an extended excitable system. Chaos **21**(3), 033104 (2011)
154. Sato, D., Xie, Y., Weiss, J.N., Qu, Z., Garfinkel, A., Sanderson, A.R.: Acceleration of cardiac tissue simulation with graphic processing units. Med. Biol. Eng. Comput. **47**(9), 1011–1015 (2009)
155. Landoni, M., Genoni, M., Riva, M., Bianco, A., Corina, A.: Application of cloud computing in astrophysics: the case of Amazon web services. In: Software and Cyberinfrastructure for Astronomy V, vol. 10707, p. 107070G. International Society for Optics and Photonics (2018)
156. Bartocci, E., et al.: Teaching cardiac electrophysiology modeling to undergraduate students: laboratory exercises and GPU programming for the study of arrhythmias and spiral wave dynamics. Adv. Physiol. Educ. **35**(4), 427–437 (2011)
157. Fallis, A.: CUDA by example, vol. 53 (2013)
158. NVIDIA: NVIDIA CUDA C Programming Guide Version 4.2, p. 173 (2012)