Experiences Accelerating MATLAB Systems Biology Applications

Lukasz G. Szafaryn, Kevin Skadron, Jeffrey J. Saucerman
University of Virginia
{lgs9a, ks7h, jjs3g}@virginia.edu

***Abstract*— Systems biology seeks to develop an understanding of the myriad interacting components of full biological systems, often in order to help treat or prevent diseases. System biology relies heavily on computation for parameterization of systems and their simulation. Although MATLAB is a convenient programming environment of choice for most scientists, its performance suffers because of single-threaded interpreted execution model. Optimizations of code and architectures are required in order for scientists to perform large-scale simulations on desktop machines. This paper presents a study of accelerating two typical systems biology applications, Heart Wall Tracking and Cardiac Myocyte Simulation, which result in 2.4x and 5.0x speed-ups, respectively. Our optimizations include improving single thread performance by using compiled (vs. interpreted) code as well as taking advantage of multiple thread execution by offloading parts of code via multi-core CPU or GPU. We show that each of the applications presents different programming and architectural challenges related to code structure and degree of parallelism exhibited. GPUs have the potential to provide the best speed-up if overhead due to the driver and data transfer can be eliminated. The feasibility of optimizations is analyzed in terms of the tradeoff between coding effort, degree of code modification and achievable speed-up. The paper also discusses code and architecture changes that would result in better acceleration.**

***Index Terms*—Biological Systems, Simulation, Parallel Processing**

I. Introduction

The field of systems biology uses computational methods to process large amounts of data describing full biological systems in order to develop models of physical processes taking place inside of them. One type of systems biology task involves processing of biomedical images that provide a rich source of data for building and testing models, such as in the Heart Wall Tracking application. On the other hand, biological systems that have already been parameterized, such as the one in the Cardiac Myocyte Simulation, are simulated to reconstruct their behavior.

Processing of large amounts of image data as well as simulating accurate models in MATLAB is usually time-consuming. Although MATLAB provides a convenient high-level environment for expressing algorithms, its performance is limited by its single-threaded interpreted execution model. Since the availability of results in reasonable time is critical to scientists' ability to conduct research, biological models are often simplified and simulation intervals shortened at the expense of fidelity.

The easiest way to optimize MATLAB application is to turn parts of it into compiled code. This can be done using the original MATLAB code (via convenient tools such as Embedded MATLAB Compiler) or by using the equivalent hand-written C code (via MATLAB MEX Compiler). While the former often requires some changes to the code (nested structure and variable sizes), it is a convenient solution overall. The latter, on the other hand, can provide better performance at the expense of sometimes considerable coding effort.

Extraction of parallelism with multi-core processors, on the other hand, requires parallel programming skills and the knowledge of new languages. However, the use of many high-level MATLAB functions makes it difficult to extract parallelism and translate code to another language. Also, the moderate amount of parallelism in many applications requires taking into account the significant communication overhead when using a co-processor such as the GPU. Existing parallel processing packages for MATLAB are limited to a subset of functions and require a large degree of parallelism to justify the cost of offloading.

As a result, acceleration of applications is currently a demanding task that is usually done manually by skilled programmers. The drastic optimizations necessitate time consuming modifications to code structure that sacrifice modularity, which in turn can make the code difficult to use for the scientist. Moreover, many scientists are unwilling to accept such sacrifices unless they yield at least an order of magnitude speed-up.

This paper introduces our work in progress, still developmental. We make the following contributions:

- Describe experiences in accelerating two typical MATLAB systems biology applications by optimizing original code and rewriting parts of it for multi-threaded CPU and GPU.
- Characterize the degree of parallelism in each application and the feasibility of optimization.
- Illustrate common architecture and language difficulties faced during the optimization process.
- Investigate how to best obtain code acceleration while maintaining the original MATLAB code structure.

The ultimate goal of our research is to develop tools for analysis, compilation and automatic offloading of parallel computation to make acceleration available to an ordinary MATLAB programmer.

II. Related Work

The early attempts to optimize MATLAB targeted utilization of clusters with MPI algorithms as back-end engines for computation. Software such as ParMatlab, Matpar, MALTAB Parallel Toolbox and StarP use this approach. Wide availability of desktop machines motivated parallelization approaches for multi-core shared memory processors. Some of the above mentioned software supports this approach as well. Packages such as Otter, FALCON, and Menhir link compiled MATLAB code with parallel numerical libraries that can run on heterogeneous architectures [1]. Jacket is an example of applying this approach to GPUs. It uses precompiled GPU-enabled versions of popular MATLAB functions and compiles simple structures such as loops [2]. The MEX interface provided in MATLAB allows for linking various types of hardware-accelerated code [3]. While the popular OpenMP standard is still not supported, packages with multi-threaded replacements for common MATLAB functions are available. Recent introduction of an API for general-purpose GPU computation provides a more convenient way to use GPUs as accelerators for MATLAB code. Throughput-oriented GPUs provide considerable speedup even over quad-core CPUs for a range of applications [4].

III. Experiment Setup and Methodology

Several techniques were used for optimizing applications described in this paper. Parts of applications were compiled to MEX files either from the original MATLAB code (using Embedded MATLAB) or the equivalent hand-written C code (using MEX Compiler). Only performance of the latter (~20% better than the former on average) is reported. MATLAB differential equation solver in Cardiac Myocyte Simulation was replaced by compiled CVODE solver [5]. We used the .NET thread standard for CPU multi-threaded code and CUDA [6] for GPU multi-threaded code. Our desktop machine was equipped with Intel Core 2 CPU (2 cores clocked at 1.86 GHz each), NVIDIA GeForce GTX 280 GPU (240 streaming processors clocked at 1296 MHz and 1024 MB of RAM) and 2 GB of RAM. All code was compiled using MS Visual Studio 2005 compiler and tested under Windows XP SP3. MATLAB profiler and a high-resolution C timer were used for performance measurements.

IV. Heart Wall Tracking

A. Application Description

This application tracks the movement of a mouse heart over a sequence of 100 609x590 ultrasound images to observe response to the stimulus. In its initial stage, the program performs image processing operations on the first image to detect initial, partial shapes of inner and outer heart walls. These operations include: edge detection, SRAD despeckling [7], morphological transformation and dilation. In order to reconstruct approximated full shapes of heart walls, the program generates ellipses that are superimposed over the image and sampled to mark points on the heart walls (Hough Search). In its final Tracking stage, program tracks movement of surfaces by detecting the movement of image areas under sample points as the shapes of the heart walls change throughout the sequence of images.

B. Algorithm and Optimizations

Only three parts of the application (Table 1) have enough parallelism and significant contribution to overall run time to justify optimization efforts. Because of the complexity of code, we could not use automated compilation tools. We avoided converting all parts of the application to C (with expected 3x speed-up) in order to focus on exposing GPU optimization potential which was of more interest to us for this application. The first two parts, SRAD and Hough Search, were simple enough to be entirely converted to C which resulted in 2.46x and 2.41x speed-ups, respectively. Since these two functions operate on entire images, there was sufficient data to distribute calculations across several parallel processors, thus justifying the use of GPU. The independent nature of operations with little need for synchronization and sufficient work per kernel minimized GPU overhead ultimately resulting in speed-ups of 9.90x and 7.45x for the two functions, respectively.

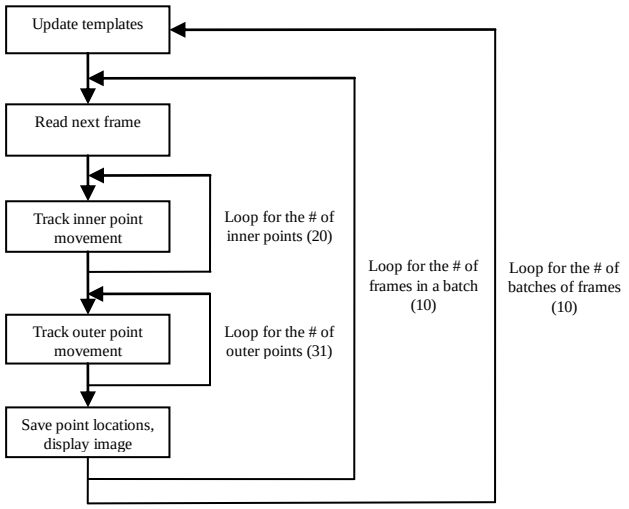

Fig. 1. Tracking part of the Heart Wall Tracking application.

TABLE I
Performance of Heart Wall Tracking Application

Application Part	MATLAB run time [s]	C run time [s] / speed-up [x]	CUDA run time [s] / speed-up [x]	*CUDA' run time [s] / speed-up [x]*
SRAD	8.71	3.54 / 2.46	0.88 / 9.90	*0.24 / 36.66*
Hough Search	15.87	6.59 / 2.41	2.13 / 7.45	*0.58 / 27.60*
Tracking – convolution	94.63	---	37.30 / 2.54	*10.07 / 9.40*
Tracking - algebraic	20.94	---	8.95 / 2.34	*2.42 / 8.67*
Tracking – statistical	13.70	---	9.32 / 1.47	*2.52 / 5.44*
Tracking - all	129.28	---	55.97 / 2.31	*15.11 / 8.55*
All Parts	187.39	---	78.40 / 2.39	*21.17 / 8.85*

The tracking part of the application consists of multiple nested loops (Fig. 1) that process batches of sample points from the image. There is a sequential dependency between processed frames. Because of the substantial coding effort involved, we only attempted to optimize individual operations on a fine-grained level. These consist of a large number of small serial steps with interleaved control statements. Each of the steps involves a small amount of data processing performed only on a subset of entire image. Because of this, the utilization of GPU was limited only to three streaming processors and GPU overhead (data transfer and kernel launch) became significant. Since the run time of the entire application was by far dominated by the Tracking stage, we had to resort to more drastic GPU optimization techniques that sacrificed modularity in order to further improve performance. These techniques included combining unrelated functions and data transfers in single kernels, which ultimately resulted in 2.39x speed-up. The last column in Table 1 illustrates that the best speed-up could be achieved with GPU if its overhead was eliminated by using a combined CPU-GPU chip architecture. This estimate is based on our measurements of kernel launch and data transfer times for offloaded operations, which was 73% on average.

V. Cardiac Myocyte Simulation

A. Application Description

This simulation models the behavior of a cardiac myocyte (heart muscle cell) according to the work by Saucerman and Bers [8]. The model integrates cardiac myocyte electrical activity with the calcineurin pathway, which is a key aspect of the development of heart failure. The model spans large number of temporal scales to reflect how changes in heart rate as observed during exercise or stress contribute to calcineurin pathway activation, which ultimately leads to the expression of numerous genes that remodel the heart's structure. It can be used to identify potential therapeutic targets that may be useful for the treatment of heart failure. Biochemical reactions, ion transport and electrical activity in the cell are modeled with 91 ordinary differential equations (ODEs) that are determined by more than 200 experimentally validated parameters. The application feeds differential equations into the solver to obtain results for a specified time interval (5 s for the simulation described here). Since the ODEs are stiff (exhibit fast rate of change within short time intervals), they need to be simulated at small time scales with an adaptive step size solver.

B. Algorithm and Optimizations

The execution time of the application is almost entirely dominated by the MATLAB ODE solver (~25%) and the evaluations of the model (~70%). Therefore only the structure of the solver (Fig. 2) and the input files are of interest to us. The process of ODE solving is based on the causal relationship between values of ODEs at different time steps, thus it is mostly sequential. At every dynamically determined time step, the solver evaluates the model consisting of a set of 91 ODEs ("1x91 evaluation") and 480 supporting equations to determine behavior of the system at that particular time instance. If evaluation results are not within the expected tolerance (usually as a result of incorrect determination of the time step), the recovery process takes place. As a part of this process, MATLAB solver recalculates the Jacobian, which results in 91 evaluations of the model ("91x91 evaluation").

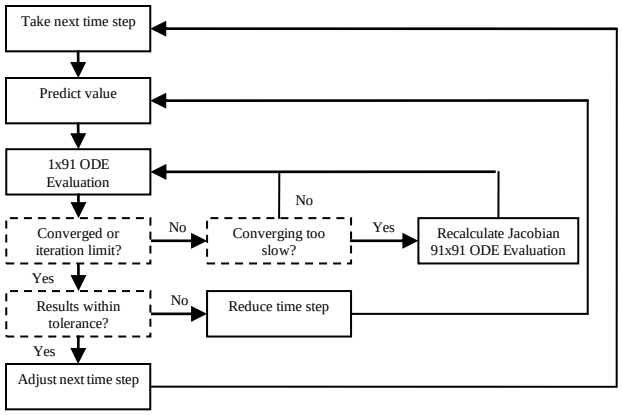

Fig. 2. Main part of MATLAB ODE solver in Cardiac Myocyte Simulation.

Our first optimization included performing the 1x91 and 91x91evaluations in C code, which resulted in 2.94x speed-up. In our second attempt we optimized the 91x91 evaluation by multi-threading and using multiple CPU cores, which resulted in 3.31x speed-up for 2 cores. We also coded the same 91x91 evaluation for GPU, however, data transfer overhead brought the speed-up down to 1.48x. The 1x91 evaluation did not have enough parallelism to justify offloading to either multi-threaded CPU or GPU. Finally, we replaced the interpreted MATLAB solver with an external, compiled CVODE solver. The compiled C version of the model evaluated with CVODE solver resulted in the speed-up of 5.04x. Rows 4 and 9 in Table 1 show that the best speed-up could be achieved with the use of GPU if its overhead was eliminated by using a combined CPU-GPU chip architecture.

TABLE II
Performance of Cardiac Myocyte Simulation

	Application Part		Run time [s] / speed-up [x]
	ODE Solver	ODE Model	
1	MATLAB	MATLAB	10.3 / 1.00
2	MATLAB	C	3.50 / 2.94
3	MATLAB	CUDA (GPU)	6.96 / 1.48
4	*MATLAB*	*CUDA (GPU) '*	*3.02 / 3.41*
5	MATLAB	C (Multi-threaded CPU)	3.11 / 3.31
6	CVODE	MATLAB	7.21 / 1.43
7	CVODE	C	2.04 / 5.04
8	CVODE	CUDA (GPU)	5.52 / 1.87
9	*CVODE*	*CUDA (GPU) '*	*1.61 / 6.39*

VI. Discussion

The process of optimizing the Heart Wall Tracking application illustrates the diverse nature of a typical image processing application. While simple functions such as SRAD and Hough Search are nicely parallel, the structure of the Tracking function that dominates the application is largely sequential. In the former case, the speed-up obtained with GPU is limited by the size of images still commonly used by scientists. In the latter case, the parallelism is limited to within a single image with the complexity of code requiring significant changes in order to parallelize, which in turn limits the effectiveness of the offloading of individual operations. Currently, we are in the process of modifying coarse-grained loop structure of the Tracking code to achieve better speedups at a significant programming cost.

In case of the Heart Wall Tracking application, the increased performance allows faster processing of images and higher throughput. However, the accuracy of processing is not affected as it is limited by the fixed size of images obtained from the medical equipment. In case of the Cardiac Myocyte Simulation, on the other hand, both the amount and the accuracy of results can benefit from faster processing. Cardiac function spans from millisecond-scale dynamics of electrical activity to the remodeling of gene expression and heart structure over weeks. The ability to develop predictive models is severely limited by computational resources. As a result of this, most current models of cardiac myocytes are limited to several minutes of simulated time, and comprehensive analyses of these models are not generally performed. Accelerated simulations provide an opportunity to expand the aspects of heart function accessible to modeling, as well as providing opportunities to more thoroughly analyze the properties of current models. Such advances are crucial for modeling the slow, adaptive process of heart failure and other heart diseases.

The process of optimizing Cardiac Myocyte Simulation was challenging because of the sequential time-step nature of the ODE solving process which limits parallelism to within a single time step. Since the MATLAB ODE solver is an encapsulated function call, its replacement with CVODE did not alter the structure of user's code. The best speedup of 5.04x that we finally obtained (with compiled solver and compiled model) could be further increased by eliminating MATLAB environment entirely. Our optimization allows reducing the simulation time and/or including more details in the model.

Results obtained for both applications considered in this paper suggest that GPU would offer the greatest potential for code acceleration if its overhead due to the driver and data transfer was eliminated. Future-generation chips that combine heterogeneous CPU-GPU cores in one package [9] should overcome this problem. The Heart Wall Tracking application would ultimately benefit more from the use of GPU because of the larger degree of parallelism (determined by each individual operation performed on subsets of 609x590 images). In the Cardiac Myocyte Simulation, on the other hand, the achievable GPU speedup is limited by the serial structure of the solver of and limited workload (only about 480 equations) in the evaluation of the model.

VII. Conclusions and Future Work

Our experiences with accelerating two representative systems biology applications, Heart Wall Tracking and Cardiac Myocyte Simulation, allow us to conclude:

- Many important applications remain difficult to reorganize for scalable parallelism.
- Retaining structure and extensibility of the code further limits parallelism, but major structural changes are unacceptable to many users.
- Improvement in application performance is mainly proportional to the coding effort.
- The use of software with persistent CPU threads is required to eliminate thread launch overhead for small workloads.
- Combined CPU-GPU chips should eliminate current overhead and make GPUs the most successful accelerators even for medium and small workloads.
- Packages such as Jacket fail for this type of applications because they lack support for functions that dominate execution time and they assume large parallelism.

With the current programming paradigm, a scientist attempting to use compilation for speed-up is either required to write compliant MATLAB code or to generate C code manually. Also, the specifics of the particular accelerator need to be known to determine the benefit of offloading. A promising alternative could be an approach based on compiler assistance. Compiler should analyze a particular section of source code, determine whether it would benefit from accelerating, compile it and/or automatically offload it to an appropriate back-end engine transparently to the user. A more advanced compiler also should derive dependency tree that could be used to batch accelerator offload requests.

Interesting directions for future work include automated compiler analysis within the MATLAB runtime to perform the necessary restructuring transparently while preserving the overall MATLAB programming "look and feel". This includes automatic analysis of whether to optimize and/or offload a particular section of code or run it natively on a CPU. Techniques to cope with tightly coupled serial-parallel steps using alternative accelerators are another possibility.

References

[1] R. Choy and A. Edelman. Parallel MATLAB: Doing it right. IEEE Proceedings. Volume 93: 331-341, Issue 2, February 2005.

[2] Accelereyes. Jacket. http://www.accelereyes.com/overview.php, 2008.

[3] Mathworks. Using MEX-Files to Call C and Fortran Programs. MATLAB Documentation. http://www.mathworks.com/access/helpdesk.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A Performance Study of General Purpose Applications on Graphics Processors using CUDA. JPDC, Elsevier, June 2008.

[5] Lawrence Livermore National Laboratory. Sundials: CVODE. https://computation.llnl.gov/casc/sundials/main.html.

[6] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable Parallel Programming with CUDA. ACM Queue, 6(2):40-53, Mar.-Apr. 2008.

[7] Y. Yongjian and S.T. Acton. Speckle reducing anisotropic diffusion. IEEE Transactions on Image Processing. Volume 11, November 2002.

[8] J. J. Saucerman and D. M. Bers. Calmodulin Mediates Differential Sensitivity of CaMKII and Calcineurin to Local Ca^{2+} in Cardiac Myocytes. Biophysical Journal 95:4597-4612, 2008.

[9] AMD. The Industry-Changing Impact of Accelerated Computing. http://www.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf. 2008.