Parallel Simulation for Parameter Estimation of Optical Tissue Properties

Mihai Duta[1], Jeyarajan Thiyagalingam[1], Anne Trefethen[1],
Ayush Goyal[2], Vicente Grau[2], and Nic Smith[2]

[1] Oxford e-Research Centre
[2] Computing Laboratory
University of Oxford, Oxford, UK
jeyarajan.thiyagalingam@oerc.ox.ac.uk

Abstract. Several important laser-based medical treatments rest on the crucial knowledge of the response of tissues to laser penetration. Optical properties are often localised and are measured using optically active fluorescent microspheres injected into the tissue. However, the measurement process combines the tissue properties with the optical characteristics of the measuring device which in turn requires numerically intensive mathematical simulations for extracting the tissue properties from the data.

In this paper, we focus on exploiting the algorithmic parallelism in the bio-computational simulation, in order to achieve significant runtime reductions. The entire simulation accounts for over 30,000 spatial points and is too computationally demanding to run in a serial fashion. We discuss our strategies of parallelisation at different levels of granularity and we present our results on two different parallel platforms. We also emphasise the importance of retaining a high level of code abstraction in the application to benefit both agile coding and interdisciplinary collaboration between research groups.

Keywords: tissue optics, parallel simulation, CUDA, parallel MATLAB.

1 Introduction

This paper describes the acceleration of a computationally intensive biomedical application through the exploitation of parallelism at different levels of granularity while retaining a high level of code abstraction. The biomedical application involves a large number of computer simulations, which are based on a mathematical model that is subject to a continuous refinement process as a result of research. This research is an interdisciplinary effort, drawing on results from both biology and mathematics. Thus, the computational performance demanded by the computer simulations is conflicting with the need of high-level abstractions in the implementation, which facilitates the rapid inter-collaborative development of the mathematical model.

Software performance and programming abstraction are two contradicting goals of software engineering. Resolving these contradictions in the case of an interdisciplinary research effort is aggravated by the likely lack of expertise in using conventional high-performance programming languages (*e.g.* FORTRAN or C++) or of numerical function libraries. The typical challenges to address in such a case are:

- the ability to cope with rapidly changing prototypes (agile code development);
- the retention of a high level of abstraction in order to enable the easy knowledge transfer across disciplines;
- the ability to exploit parallelism without trading off abstraction.

The approach followed in this paper towards maintaining the programming abstraction is the exploitation of parallelism at multiple levels of granularity. Thus, independent computer simulations on which the biomedical application is based are run on conventional multi-core computer hardware or multi-node clusters in a parallel fashion controlled via an abstracted description using Parallel MATLAB [1]. At the same time, parts of each individual simulation are accelerated on general-purpose graphics cards (GPUs) using nVidia CUDA [2].

This paper makes the following contributions:

- we develop a parallel computational model for the biomedical application, which effectively exploits parallelism at different levels of granularity;
- we demonstrate significant computing speedups of the application while retaining the high-level abstraction, which is essential for an agile interdisciplinary code development.

The rest of this paper is organised as follows. We first provide a background on tissue optics and introduce the underlying mathematical model of the biomedical application in Section 2. We then describe our approach to addressing the challenges raised by this interdisciplinary application in Section 3. Following that, we give the details of parallelising the simulation in Section 4. This is followed by a discussion of the performance results in Section 5. The conclusions are drawn in Section 6, where the generality of the present approach is also assessed.

2 Tissue Optics

Recent medical advances have witnessed the development of a host of minimally invasive laser-based treatments, *e.g.* the detection of bacterial pathogens in living hosts, the measurement of neoplasmic tumour growth, the pharmacokinetic study of drug effects or photo-dynamic cancer therapy [3]. A crucial factor in these treatments is the optical tissue properties, which vary throughout the body [4] and which determine how laser light is scattered, diffused and attenuated. As a consequence of varying optical properties, tissues are exposed to varying energy levels of incident light [5] and the treatment must avoid extended exposures to high energy levels. A good understanding of the optical properties of tissue is, therefore, very important.

Of the several applications, the specific medical research application of this paper is optical fluorescence imaging, which is used for obtaining high-contrast images of coronary vasculature. This application concerns measuring radii of blood vessels in the coronary vasculature images, which requires knowledge of the optical properties of the tissue. To estimate these optical properties, fluorescent microspheres are first injected into the tissue vascular network. The microspheres embedded in the tissue are excited by filtered light which in response emit light of a spectrum distinct from the incident light. The emitted light is recorded by a CCD camera every time a slice (in the micrometre range) of the tissue is sectioned off [6]. The result is a set of digitised 2D images that make up, slice-by-slice, the 3D tissue volume.

This set of images does not directly characterise the optical properties of the tissue as the recorded photographic data mixes the sought properties with the optics of the recording device. The prevalent way to describe the recording process is via point-spread functions (PSF) [7], which are mathematical models of how optical media blurs an imaged object. In essence, a PSF describes the response of an optically active system to a point light source, so its precise definition is that of a Green's (or impulse response) function. In our application, two distinct PSF models are used to represent the optical properties of both the recording device and biological tissue. Thus, the recorded digital image I of a single microsphere is modelled as the three-dimensional (3D) image convolution (denoted by $\otimes$) of the original microsphere image I_m with the PSF f_1 of the biological tissue and the PSF f_2 of the recording device:

$$I = I_m \otimes f_1 \otimes f_2 \qquad (1)$$

For each microsphere, f_1 is parameterised by

- the coefficients of excitation and emission tissue attenuation, μ_{ex} and μ_{em}, respectively.
- location of the microsphere (denoted by s) inside the slice, which affects the detected intensity of the microsphere.

The parameters that characterise f_1 are not known and therefore estimating them constitutes part of the calculation of tissue properties. The parameter estimation represents a fitting problem, where a simulated image I_s is synthetically generated and the parameters are adjusted such that the difference between the simulated image and the observed image I is minimised in a least squares sense. The algorithm is iterative, starting with initial parameter values and updating them (and thus generating a new image every time) until the least squares difference is minimised. The overall algorithm for the computational model is depicted in Figure 1. In the model, as a first step, a set of the synthetic microspheres M are initialised with known synthetic values via the function *Initialise_Synthetic*. *InitGuess* initialises the parameters with initial values obtained from the literature [8]. *GetProps* extracts the *a-priori* known parameter values for a given microsphere, *SignalSpread* performs expensive signal convolution between single point source and surrounding simulated tissues and the lens, as specified in Equation 1. As part of this operation, the images being convolved need to be resized through interpolation along one of the axis, the *imresize* operation. The *Optimise* function is an optimiser which optimises the values of μ_{em}, μ_{ex} and s satisfying the constraints. These newly searched values become the current parameters, which are set by the *SetProps* function. The search is continued until the mean-squared error, calculated by *MSE*, between the actual image and the simulated image (formed with the optimised parameter values) is minimised. The optimiser may rely on any algorithm and in our case this is based on the Newton-Raphson gradient descent algorithm [9]. The search is constrained such that parameters μ_{ex}, μ_{em} and s can assume only a sub-set of values and being inter-dependent [10]. Furthermore, the algorithm is computationally expensive

```
$M \leftarrow$ Initialise_Synthetic()
for $m \in M$
    $[\mu'_{ex}, \mu'_{em}, s'] \leftarrow$ InitGuess()
    $err \leftarrow \infty$
    $[\mu_{ex}, \mu_{em}, s] \leftarrow$ GetProps($m$)
    while $(err > \epsilon)$
        $V \leftarrow$ SignalSpread($\mu'_{ex}, \mu'_{em}, s'$)
        $err \leftarrow$ MSE($V, m$)
        $[\mu'_{ex}, \mu'_{em}, s'] \leftarrow$ Optimise($\mu'_{ex}, \mu'_{em}, s', err$)
        SetProps($\mu'_{ex}, \mu'_{em}, s'$)
    end while
end for
```

Fig. 1. Sequential computational model of the problem

— the cost of computation depends directly on the number of microspheres involved, which is typically in the order of tens of thousands.

3 Challenges and Approach

3.1 Challenges

One of the pressing factors of this study is the *time-to-solution*. This includes the code development time, time spent on simulation and time spent on the interpretation of results. The simulation is highly time consuming and the setting should be such that scientists should be able to map their domain-specific knowledge without ample programming expertise, let alone parallel programming experience. This need for agile development leads us to consider a solution whereby we should be able to develop the model with minimal effort while exploiting high performance computing resources.

These issues collectively demand a solution with the ability to represent high-level abstractions with the facility to utilise them in a parallel environment. Libraries or high-level programming languages such as Fortran or C++ may not yield that level of abstractions, especially with parallel constructs. Furthermore, the need for rapid prototyping and simulation discourages such a choice. Modelling tools such as Simulink [11] may be a useful approach but the abstractions within Simulink still need to be developed.

3.2 Our Approach

In realising a solution, we favour an array programming language, MATLAB [1]. Array programming languages offer high level of abstractions, especially when combined with their own libraries. They offer an easier route for developing short, but yet powerful prototypes. By the same token, Octave, Python and SciLab are potential alternatives to MATLAB. However, we were motivated to choose MATLAB on the basis of interactiveness, core functionalities, popularity, support and scalability of applications.

MATLAB is an integrated, modular development and visualisation platform. Abstractions within MATLAB goes beyond the notion of libraries. While delivering Fortran-90 like array-level abstractions, MATLAB specialises on different domains using toolboxes, which are implementations of very common domain-specific operations. These toolboxes bridge the communication gap between scientists across different disciplines and enables easier knowledge transfer across domains.

MATLAB supports parallel computing through a specialised toolbox — Parallel Computing Toolbox [12]. This toolbox enables writing parallel loops or processing large arrays with the notion of single program multiple data. Functionalities of Parallel Computing Toolbox does not scale beyond a single node and this is augmented by a Distributed Computing Toolbox, which is transparently run on the master node, the distributed computing engine. All these collectively enable utilising a cluster of workstations in a transparent manner. The fact that MATLAB is a just-in-time compiled language often leads to suboptimal runtime performances. To overcome this, we selectively modify and use the CUDA architecture wherever possible, without breaking any abstractions.

4 Parallelisation and Computational Considerations

In view of identifying parallelism within the parameter fitting algorithm depicted in Figure 1, the following observations are useful.

1. The task of optimising the parameter space for one microsphere is independent of another; this allows the microspheres to be processed in parallel.
2. The fitting algorithm relies on an optimiser to find the point (μ_{ex}, μ_{em}, s) in the parameter space that minimises the *SignalSpread* cost function. Numerous types of optimisers require repeated and independent evaluations of the *SignalSpread* cost function, which can be obtained in a parallel manner.
3. The *SignalSpread* function performs 3D image convolution operation on three different images as given in Equation 1. Following the standard approach to numerical convolution, the images are first transformed to the frequency domain, followed by a frequency-domain multiplication and the transformation of this result back to the image domain. This involves two discrete Fourier transforms, for which the popular numerical algorithm is the Fast Fourier Transform (FFT). Parallel multi-threaded implementations of this exist, which exploit a fine-grained level of parallelism.
4. To avoid the large memory penalty of convolving the 512^3 resolution image volume within the *SignalSpread* function, we convolve a down-sampled image and then resize the result via interpolation. This operation exposes fine-grained parallelism. The negligible loss of accuracy in the interpolated image does not affect the tuning of optical parameters.

The spatial resolution of a microsphere image varies between 32^3 and 512^3 type of double, corresponding to 256KB and 1GB respectively. However, to avoid strong aliasing effects in converting an image to the frequency domain, the doubling (at a minimum) of the original array size along all three dimensions. Additionally, the entire convolution operation requires three 3D arrays for the reasons stated in Section 2, which raises the memory requirement for a single operation to 6MB and 24GB, respectively. This highlights clearly the need for image resize manipulation, which reduces the memory imprint of convolution to manageable dimensions.

In light of the demands posted in Section 3, the representation of the parallelism should be such that it remains at a higher level as much as possible. High level parallelism is clearly expressed by the observations (1) and (2). This is easily exploited in an abstracted way within Parallel MATLAB, as described in the next section.

At the other end of the spectrum, fine-grained parallelism as expressed by the FFT and image resize operations (observations (3) and (4)) can take advantage of multi-threading, and adds extra performance on appropriate hardware. In this work, programmable GPUs were used at a great advantage; their specialised programming is in contradiction with the overall requirement for abstraction but is of limited extent within the application and hidden within MATLAB functions.

4.1 Coarse-Grained Parallelism Using Parallel MATLAB

MATLAB provides parallelism with the notion of *workers*. Each worker is responsible for a MATLAB session and a subscription mechanism does exist to govern the granularity of the worker. It is very common to subscribe a single worker per core of a microprocessor. In the case of a cluster of workstations, multiple workers may exist and their execution are orchestrated by the distributed computing engine. The distributed computing engine may use any scheduler and provides the transparency across workers. In other words, the application does not differentiate between workers which span across different nodes and workers residing within the same host.

Using the Message Passing Interface as the underlying communication mechanism between workers, MATLAB provides two different forms of high-level constructs to perform parallel operations: `parfor` and `spmd`. The `parfor` construct represents a parallel loop, whose enclosed statements are executed in parallel across workers in a lock-stepped fashion. The data distribution is automatic where the array is sliced based on the access patterns. The `spmd` construct is a single program, multiple data construct, where all workers perform the same operation but on different data. The data distribution can be performed manually depending on the requirements. In this work, we chose the `spmd` construct to achieve the parallelism at the microsphere level.

Evaluation of the parameter space for each microsphere is independent from each other, but may vary in their runtime depending on the convergence rate of the optimiser. This implies that microspheres cannot be processed in a locked-stepped fashion without incurring additional delays. The `spmd` construct of Parallel MATLAB allows this condition to be met: while enabling several MATLAB workers to perform similar operation but on different data without the need for an explicit synchronisation. The original data space, here a set of microspheres, is partitioned across distributed workers to enable workers to deal with different data. Since no communications are required across workers for the evaluation of the parameter space, this is an ideal case for `spmd`. The modified algorithm accounting this is shown below in Figure 2. In addition to the original notions in the sequential model, M is distributed across workers using the *CoDistribute*. Placeholder result arrays $(\mu^d_{ex}, \mu^d_{em}, s^d)$ are created across all workers in a distributed fashion using *CreateDistributed*. The spmd block guarantees that the all

```
$M \leftarrow$ Initialise_Synthetic()
$M^d \leftarrow$ CoDistribute(M,N)
$\mu_{ex}^d \leftarrow$ CreateDistributed(N)
$\mu_{em}^d \leftarrow$ CreateDistributed(N)
$s^d \leftarrow$ CreateDistributed(N)
spmd (N)
        for $m \in M^d$
            $[\mu_{ex}^{d'}, \mu_{em}^{d'}, s^{d'}] \leftarrow$ InitGuess()
            $err \leftarrow \infty$
            $[\mu_{ex}, \mu_{em}, s] \leftarrow$ GetProps($m$)
            while $(err > \epsilon)$
                $V \leftarrow$ SignalSpread($\mu'_{ex}, \mu'_{em}, s'$)
                $err \leftarrow$ MSE$(V, m)$
                $[\mu'_{ex}, \mu'_{em}, s'] \leftarrow$ Optimise$(\mu'_{ex}, \mu'_{em}, s', err)$
                SetProps($\mu'_{ex}, \mu'_{em}, s'$)
            end while
            $\mu_{ex}^d(m) \leftarrow \mu'_{ex}$
            $\mu_{em}^d(m) \leftarrow \mu'_{em}$
            $s^d(m) \leftarrow s'$
        end for
        $\mu_{ex} \leftarrow$ gather($\mu_{ex}^d$)
        $\mu_{em} \leftarrow$ gather($\mu_{em}^d$)
        $s \leftarrow$ gather($s^d$)
end
```

Fig. 2. Parallel computational model of the problem

workers perform the same operation on their own data. Finally, the data are gathered and consolidated to non-distributed arrays μ_{ex}, μ_{em} and s using the *gather* construct. Each worker may reach a path in the program at any point in time.

4.2 Fine-Grained Parallelism Using Graphics Hardware Acceleration

The fine-grained parallelism inherent in the 3D convolution and image resizing can be accelerated on commodity GPU hardware relatively easily, using nVidia CUDA [13]. CUDA is a general purpose high-level parallel architecture that enables programmers to control the execution of (numerically intensive) code on GPUs via the C language with nVidia-specific extensions. GPU programming using CUDA is determined by two important hardware aspects:

– the highly parallel structure of GPU hardware (represented by a large number of specialised processing cores) is best exploited through a massively multi-threaded SIMD programming approach, with minimal conditional branching;
– the global memory available on GPU hardware is relatively limited and memory bandwidth between the host and the GPU is too limiting to allow graphics

computing to access the host memory in a way that overcomes the graphics memory limitation.

The convolution operations use the optimised FFT transforms implemented by the CUFFT library from nVidia. In using the CUFFT library, there are a number of factors to consider:

– The CUFFT library processes FFT transforms in an interleaved data format for the real and imaginary parts, whereas MATLAB holds the real and imaginary parts separate. The data format transformation back and forth is best carried out on the GPU itself but that obviously requires extra device memory.
– Best transform performance (in terms of both absolute speed and scaling with the data size) is achieved for transform sizes that are powers of 2; this is a general feature of the FFT algorithm in any implementation (including FFTW [14], used by MATLAB itself). The divide-and-conquer approach of FFT under performs when data size is a prime or an unbalanced factorisation containing a large prime. Another advantage of powers of two is that transform results from CUFFT and the more standard FFTW are equal to within machine accuracy, in contrast to prime sizes on which results can have a relative difference of order 1.

Apart from the FFT transforms themselves, the frequency-domain data array multiplication and scaling also benefited from GPU acceleration. Beside the 3D convolution, the parallelism inherent in the image resizing (independent row and/or column operations) also benefits from the massively multi-threaded acceleration on GPUs to great effect. Finally, to retain the high abstraction level in the application, the CUDA programming is hidden from the rest of the application via MEX interface functions directly usable from MATLAB.

5 Performance Results

Since the time for performing the entire computation is an ongoing process, we report our runtime performance for a limited set of microspheres for resolutions from 32 through 128. To enable the measurement of speedups against different versions, we fixed the number of iterations of the optimiser or wherever this was not possible, we calculated the speedups based on per iteration basis. This is perfectly valid, as the runtimes between iterations vary very marginally. Our simulations are based on a cluster of workstations and a GPU-based system whose configurations are as follows:

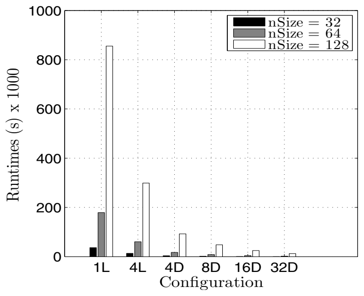
– A Microsoft cluster running Microsoft Windows 2008 (64bit) with 32-nodes, interconnected using Gigabit Ethernet. Each node is equipped with a quad-core IA-64, Xeon processor and 8GB of RAM running MATLAB (2009b, 64bit). The MATLAB is configured to use single worker-per-core and the native scheduler.
– A Tesla-S1070 GPU cluster, running on Linux (Kernel 2.6.18) with 8 compute nodes *shared* across 4 Tesla S1070 units. Each compute node is equipped with a 2 quad-core Xeon Processor and with 8GiB memory. Each GPU device is equipped with 4GB of device memory shared across 240 cores per processor and devices are interconnected using PCIe x16. The CUDA version is 2.3.

– Hosts used for local worker/single worker experiments have the same configuration of quad-core Xeon processor running Windows 2008 (64bit) and 32GB of RAM. Both the hosts are running the same version of MATLAB as the cluster. Results are averaged for these.

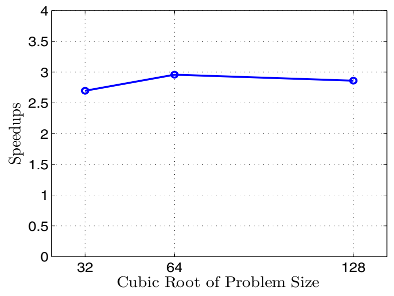
The initial speedups by simply using Parallel MATLAB and CUDA are very attractive. There are a number observations that can be made in the reported performance results.

Speedups from parallel workers: We show our runtime performance in Figure 3(a). An immediate observation, which we investigate further below, is the performance difference between local and distributed workers. In MATLAB, local workers are confined to a single node whereas distributed workers may span across any number of nodes. We report our speedups in two different stages: speedup of local workers against a single worker (4L vs 1L) and speed up of distributed workers against local workers (xD vs 4L, where $x = 4, 8, 16, 32$). When using multiple local workers, we see substantial but yet suboptimal speedups against a single worker. However, speedups of distributed workers against 4-Local workers are very attractive (and thus against 1L). Distributed workers outperform corresponding number of local workers. It is worth noticing that MATLAB does not have the notion of shared memory model, and instead it relies on the message passing model. Hence all communications are treated as messages which may create a bus contention. To investigate this further, we performed a synthetic data transfer between local and distributed workers. In the case of distributed workers, we customised the worker allocation to ensure that the workers are from distinct nodes to avoid any local bus contention. Figure 3(d) shows the ratio of transfer times of local workers to distributed workers. Transfers between distributed workers are faster than transfers between local workers by several fold. We ascribe these observations to different characteristics between local bus and network interconnects and to the schedulers. Local workers rely on the operating system whereas distributed workers rely on the scheduling policy of the distributed computing engine.

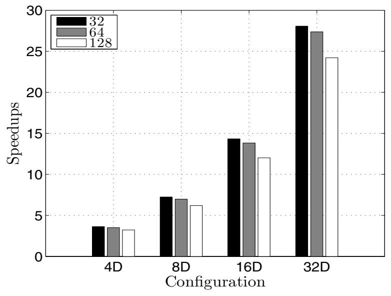
Speedups from CUDA: There are two fine-grained operations, 3D-convolution (FFT) and image resizing (ImResize), which were time consuming in the original application, with neither of them benefiting from threading within MATLAB. We parallelised each of these using the CUDA. We report our findings, again, in two stages: performance of these kernels as standalone versions and after integrating these kernels within the application. The integration of the CUDA kernels is tested in three configurations: MATLAB version with a single worker with and without CUDA and 4-local workers without CUDA . The speedups are obtained by comparing their runtimes against a single worker without CUDA (using the standard FFT and ImResize functions). The results are shown in Figures 3(e) and 3(f), respectively. In all cases, the performance figures include data transfer and transformation costs. In the case of standalone kernels, we observe that both the kernels vary in their performance with data sizes. Initial rise in speedups are obtained by amortising the cost of transfer and transformation costs by computation time. For increased data sizes, however, the transfer costs begins to dominate again and the overall speedups begin decreasing. In the case of integrated version, parallel MATLAB offers significant speedups compared to the single worker but single worker with CUDA outperforms the parallel version on a single host. Performance varies with problem sizes,



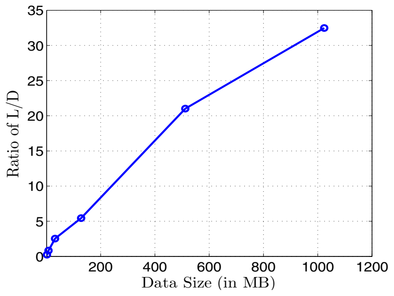
(a) Runtimes for different configurations



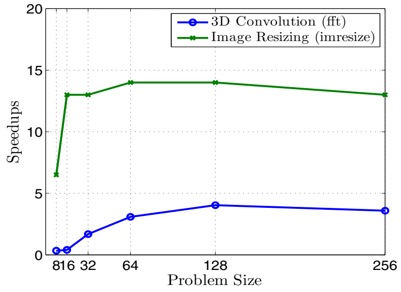
(b) Speedups with four local workers



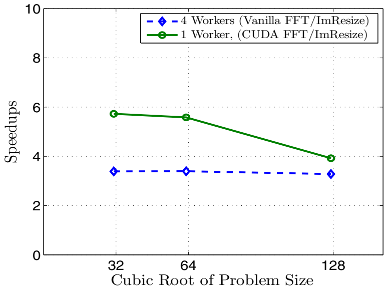
(c) Speedups with distributed workers



(d) Ratio of local to distributed data transfer times



(e) 3D Convolution/Imresize speedups



(f) Combined speedups

Fig. 3. Runtimes/Speedups for Parameter Searching. The search is carried out for three different sizes (32, 64 and 128) on different worker configurations: single worker (denoted by 1L), four workers localised to a single node (4L), four, eight, 16 and 32 workers distributed across the cluster (denoted by 4D, 8D,16D and 32D respectively) (Figures 3(a)-3(c)). The ratio of transfer times between local and distributed workers are presented in Figure 3(d). On CUDA, both 3D-Convolution and ImResize were run in isolation and after the integration with the overall solution (Figures 3(e),3(f)). See text for further information.

replicating the original effects found in standalone versions. Furthermore, the current limitation in the CUDA driver, in terms of using CUDA-based routines in MATLAB, requires constant flushing of CUDA device memory when CUDA kernels are repetitively called within MATLAB. Our explanation is that such constant memory management may trigger the MATLAB Just-in-Time compiler, which may offset some of the benefits due to CUDA parallelisation. Furthermore, runs on higher-data sizes cannot be performed as the overall memory requirements of the overall application exceeds that of the device. Albeit all these complexities, the overall speedups are substantial. Furthermore, the configuration of the GPU cluster is such that two nodes share the S1070 device and thus parallel workers will only create a contention/race condition and will not offer any speedups. The absence of any hardware with MATLAB distributed computing engine and CUDA impedes us from making use of the fine-grained parallelism within each worker.

The scalability of the solution: We observe that when increasing the number of distributed workers, the speedup increases linearly with respect to the minimum number of distributed workers (4D). In the case of CUDA, the scalability exists only for a subrange of problem sizes and this is the case even when including data transfer and transformation costs. However, integration of the kernel and associated overheads leads to sub-optimal scalability.

Programmability: What is more promising is the level of detail at which the parallelism is expressed. The same sequential algorithm becomes readily parallelised without any efforts relating to the experience of C++. However, utilising the CUDA FFT and ImResize required some expertise in understanding and in transforming the data formats within the satellite MEX functions. However, this is only a one-time effort and does not affect the level of abstraction represented by the application.

6 Conclusions

This paper outlines the solution adopted in the parallelisation of a biomedical application, in which a high-level of programming abstraction is a crucial criterion for both the rapid code development as well as for the underlying inter-disciplinary research. The key to maintaining abstraction is the exploitation of parallelism at multiple levels of granularity; at a high level, multiple simulations of the core mathematical model on which the application is based are run in a parallel fashion on a conventional computer hardware while the simulations themselves are accelerated at a low level on GPUs.

The high-level parallelism is facilitated by the use of Parallel MATLAB, which strikes a good balance between performance and a high level of abstraction, with the added advantage of its prevalence in the academic community. Parallelism at the low-level granularity benefits from remarkable acceleration on graphics cards, via CUDA programming. Both levels of parallelism are relatively flexible in terms of the type of hardware the application can be run on.

By combining Parallel MATLAB programming with low-level CUDA programming for the graphics card acceleration, remarkable performance benefits were demonstrated on two different parallel hardware configurations. The paper emphasises the importance of the high-level programming abstraction for a simple knowledge transfer across different scientific disciplines. The generality of the separation between high-level parallel programming from the low-level acceleration on graphics devices renders the approach applicable in similar cases, where inter-disciplinarity couples with agile code development.

References

1. Sharma, G., Martin, J.: MATLAB ®: A Language for Parallel Computing. International Journal of Parallel Programming 37(1), 3–36 (2009)
2. Harris, M.: Many-Core GPU Computing with nVidia CUDA. In: Proceedings of the 22nd Annual International Conference on Supercomputing, p. 1. ACM, New York (2008)
3. Shah, K., Weissleder, R.: Molecular Optical Imaging: Applications Leading to the Development of Present Day Therapeutics. Journal of the American Society for Experimental NeuroTherapeutics 2(2), 215–225 (2005)
4. Steyer, G.J., Roy, D., Salvado, O., Stone, M.E., Wilson, D.L.: Removal of Out-of-Plane Fluorescence for Single Cell Visualization and Quantification in Cryo-Imaging. Annals of Biomedical Engineering 37(8), 1613–1628 (2009)
5. Steven, L.J., Prahl, S.A.: Modeling Optical and Thermal Distributions in Tissue During Laser Irradiation. Lasers in Surgery and Medicine 6, 494–503 (2008)
6. Spaan, J.A.E., ter Wee, R., van Teeffelen, J.W.G.E., Streekstra, G., Siebes, M., Kolyva, C., Vink, H., Fokkema, D.S., VanBavel, E.: Visualisation of Intramural Coronary Vasculature by an Imaging Cryomicrotome Suggests Compartmentalisation of Myocardial Perfusion Areas. Medical and Biological Engineering and Computing 43(4), 431–435 (2005)
7. Rolf, P., ter Wee, R., van Leeuwen, G.T., Spaan, J., Streekstra, G.: Diameter Measurement from Images of Fluorescent Cylinders Embedded in Tissue. Medical and Biological Engineering and Computing 46(6), 589–596 (2008)
8. Cheong, W., Prahl, S., Welch, A.: A Review of the Optical Properties of Biological Tissues. IEEE Journal of Quantum Electronics 26, 2166–2185 (1990)
9. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer, New York (1999)
10. Goyal, A., van den Wijngaard, J., van Horssen, P., Grau, V., Spaan, J., Smith, N.: Intramural Spatial Variation of Optical Tissue Properties Measured with Fluorescence Microsphere Images of Porcine Cardiac Tissue. In: Proceedings of the Annual IEEE International Conference on Engineering in Medicine and Biology Society, pp. 1408–1411 (2009)
11. Mathworks: MATLAB Simulink, Simulation and Model-Based Design, `http://www.mathworks.co.uk/products/simulink/` (Last accessed June 1, 2010)
12. Mathworks: MATLAB Parallel Computing Toolbox, `http://www.mathworks.co.uk/products/parallel-computing/` (Last accessed June 1, 2010)
13. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High Performance Discrete Fourier Transforms on Graphics Processors. In: Proceedings of the SuperComputing 2008 (Electronic Media). ACM/IEEE, Austin (2008)
14. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: Proceedings of International Conference on Acoustics, Speech and Signal Processing, vol. 3, pp. 1381–1391 (1998)