# GPU Acceleration of Optical Mapping Algorithm for Cardiac Electrophysiology

Pingfan Meng, Ali Irturk, Ryan Kastner, Andrew McCulloch, Jeffrey Omens, and Adam Wright

*Abstract*— **Optical mapping is an increasingly popular tool for experimentally analyzing the electrical activity in the heart. The optical mapping algorithm is computationally intense and consumes a considerable amount of time even with a highly optimized program running on a state-of-the-art multi-core microprocessor. For example, one second of data requires approximately 5 minutes of computation time (3.66 FPS) with a C++ program parallelized by OpenMP running on a 3.4GHz Quad-Core CPU. This article presents a GPU implementation of the optical mapping algorithm. Our result indicates that the GPU implementation is capable of processing the optical mapping video at 578 FPS which achieves 157.92X speed against the OpenMP optimized CPU implementation.**

## I. Introduction

Optical fluorescence imaging of cardiac electrical impulses on the surface of isolated animal hearts has been proven as an effective tool [1][2] in cardiac physiology for studying the mechanisms of rhythm disturbances (dysrhythmia), which cause heart attacks. The optical mapping technique can record heart motion without interfering with it since the technique does not require physical contact. Moreover,the optical mapping technique can provide spatial electrical activity maps that include the information across the entire heart surface which provides more information for medical study than the conventional electrode technique does.

Unfortunately, optical mapping analysis in cardiac physiology is computationally intensive due to two factors: (1) high input data rate; (2) high accuracy requirement for the study of depolarization properties. Firstly, for a typical experimental setup, a high-speed camera (about $1,000$ FPS, $100 \times 100$ pixels/frame in our case) feeds the video data to the optical mapping processing system. Secondly, in order to conserve the depolarization properties, a complex phase detection process is required. In the phase detection process, the video data is interpolated temporally. Typically, the interpolation factor is 10. Thus, the phase detection process expands the data throughput to $10,000$ FPS. With such a high throughput, the optical mapping algorithm takes high performance multi-core processors hours to process just a few seconds of data. This attribute makes the optical mapping technique impractical to use in application such as high throughput screening, immediate experimental feedback and real-time feedback control [2].

Manuscript received March 29, 2012

P. Meng, A. Irturk and R. Kastner are with Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Dr. La Jolla, CA 92093

A. McCulloch, J. Omens and A. Wright are with Department of Bioengineering, University of California, San Diego, 9500 Gilman Dr. La Jolla, CA 92093

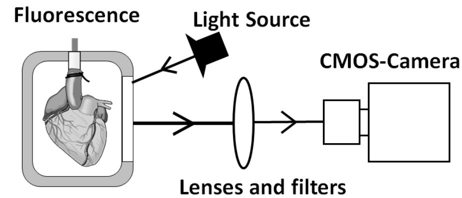


Fig. 1. Experimental setup for optical mapping video data capturing.

Graphic Processing Units (GPUs) employ many-core architectures which highly parallelize the operations and hide the memory access latencies. GPUs are proven to be effective in accelerating high throughput applications in many fields [3][4][5]. Computer Unified Device Architecture (CUDA) is an environment provided by NVIDIA for GPU programming. Although the CUDA programming environment is easy to manage, it still requires the designer to have a certain level of knowledge about the GPU architecture to achieve a high performance. In the optical mapping algorithm, the ultra high throughput causes the high performance CUDA development to be even more challenging. Moreover, the optical mapping algorithm has several different processing stages which have different computational characteristics and memory access patterns temporally and spatially (e.g. temporal phase shift followed by the spatial Gaussian filter). Due to the complexity of the algorithm, a naive CUDA implementation prevents the GPU hardware from distributing its computing resources efficiently and hiding the memory access latency effectively. Thus, optimizations on each accelerating kernel and a high entire system throughput design are needed. Ultimately, the complex temporal and spatial computational features of the optical mapping algorithm and the entire system throughput design strategy distinguish our implementation from other image processing acceleration works [6][7].

In this article, we present a high performance implementation (578 FPS processing speed) of the optical mapping algorithm using GPU for cardiac electrophysiology. The performance of the GPU implementation is evaluated by comparison to a Matlab CPU implementation, a serial C++ CPU implementation and an OpenMP C++ CPU implementation.

## II. Optical Mapping Algorithm

In this section, we briefly introduce the optical mapping image processing algorithm for cardiac physiology.

The video input data is captured with an experimental setup depicted in figure 1. In the experimental setup, with

the aid of a set of optical lenses and filters, a CMOS camera records a rabbit heart which is filled with voltage-sensitive fluorescent dye. The fluorescence intensity, which represents transmembrane potential, is converted to digital 8-bit images at $1,000$ FPS with a spatial resolution of $100 \times 100$ pixels.

The raw video data, which is collected by the high speed camera, is covered by noise. It is impossible to extract any biological information for medical studies with this noisy video data. Thus, the optical mapping technique applies an image processing algorithm to remove the noise on the raw video data [8]. The effect of the image processing is shown in figure 2.

We measured the performance of the CPU implementation of the optical mapping image processing algorithm using both Matlab and C++ programming language on a 3.40 GHz Quad-Core Intel Core i7-2600 CPU. To process 1 second of input video data, the Matlab program took 39 minutes; the serial C++ program took 22 minutes; the multi-core CPU C++ program with OpenMP (an API for multi-core parallel programming in C/C++) took 4.6 minutes. The optimized C++ program also used direct access tables to avoid computation such as trigonometric functions and the FFT output indices. However, even with multi-core parallelization and direct access tables ($O(1)$), the best CPU implementation could only achieve 3.66 FPS. This processing rate is only a tiny fraction of the 1,000 FPS camera data capturing rate. At this processing rate, a biomedical researcher has to spend hours and even days on processing a reasonable amount of useful data.

The major stages in the image processing are demonstrated in figure 2. Firstly, the image data passes through a phase shift spatial filter which includes interpolation by a factor of 10, phase shifting and spatial filtering. The phase shifting operation, which is implemented by FFT, conjugation multiplication and IFFT, is used to correct the temporal differences among pixels before any filtering in order to conserve the depolarization properties [8]. The phase shifted data is spatially filtered by a $5 \times 5$ Gaussian ($\sigma = 1.179$) window to cancel the noise. Finally, a temporal median filter is performed on the spatially filtered data to preserve the steep upstroke of the optical action potential.

## III. GPU IMPLEMENTATION OF THE OPTICAL MAPPING ALGORITHM

In this section, we present our GPU implementation of the optical mapping algorithm. First, we discussed how we partitioned the optical mapping algorithm for the design. Second, we present how we implemented and optimized the CUDA kernels. Third, we described how the optical mapping program runs on the GPU.

### A. Application Partition

The overview of the GPU implementation of the optical mapping algorithm is illustrated in figure 3. The optical mapping algorithm consists of several sequential stages of operations. We studied the data throughput for each stage. We partitioned the optical mapping algorithm into three types

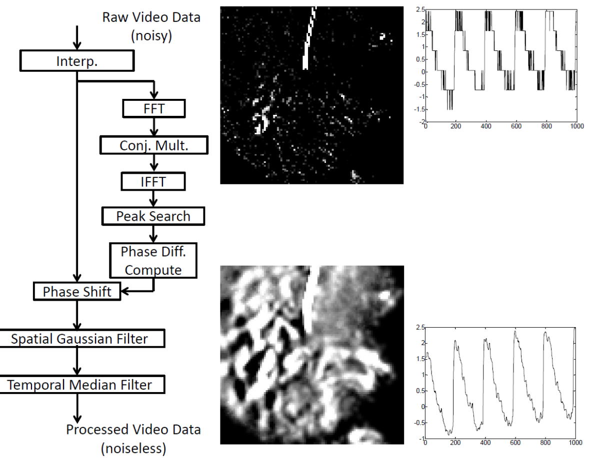


Fig. 2. The image processing of the optical mapping technique. And effect on the real images after the image processing.

of modules: CPU function, CUDA accelerating kernel and CUDA throughput kernel.

$CPU\ function\ module$ : the phase difference compute is the only module of this type in the optical mapping algorithm as shown in figure 3. This module computes the absolute phase differences by using the relative phase differences produced by the temporal peak search stage. The phase differences compute module has two features: (1) high data dependency (e.g. Given $t_{ab}$, $t_{ac}$ and $t_{bd}$ by the peak search module to calculate $t_{dc}$ in the phase difference compute module. We get two computations/threads: $t_{bc} = t_{ab} - t_{ac}$ and $t_{dc} = t_{bd} - t_{bc}$. Obviously, the second computation/thread depends on the output of the first one. One can argue to replace $t_{bc}$ with $t_{ab} - t_{ac}$. this will force the second thread to complete one more calculation, which will result in the same latency as waiting for the completion of the first thread.); (2) instruction branches (middle pixels need to calculate 23 phase differences while the edge ones are given 5 and only need to calculate 19 phase differences). With these two features, this stage is completely unsuitable for CUDA kernel implementation. Moreover, the input throughput and the output throughput are both significantly low. This means it will not cause intensive data transferring between the CPU and the GPU if it is implemented on the CPU. Therefore, we implemented this stage as a CPU function.

$CUDA\ accelerating\ kernel$ : these modules highly occupy the GPU computing resource to accelerate the complex computations in the optical mapping algorithm. These modules usually have complex computational characteristics and memory access patterns that need to be specifically optimized in the design. Once the appropriate optimization techniques are applied on these kernels, they usually gain tremendous speedups. For example, in the optical mapping algorithm, the highly optimized accelerating kernels such as $FFT$, $IFFT$ and $peak\ search$ gained hundreds of times of speedups against the CPU implementation of these functions.

$CUDA\ throughput\ kernel$ : these modules focus on low latency instead of high GPU computing resources occupancy

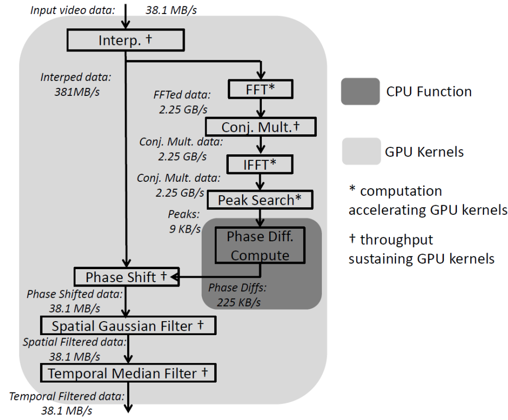


Fig. 3. GPU implementation overview. For each stage, the data throughput is demonstrated. For the FFT stage, the high data throughput is due to the padded zeros and the complex number data type.

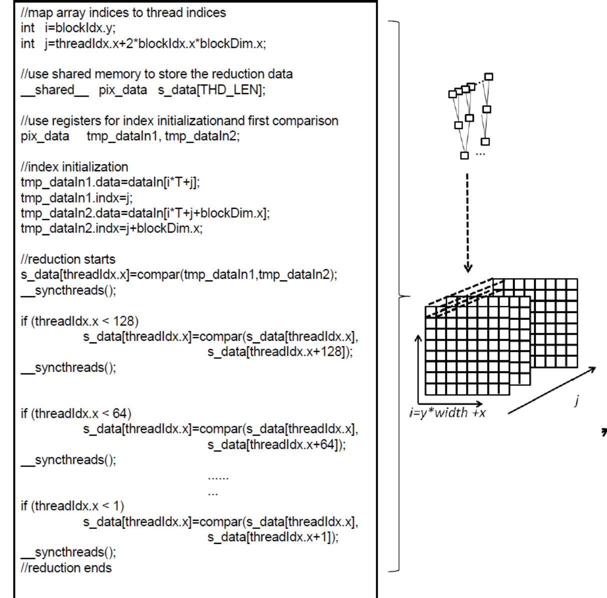


Fig. 4. The temporal peak search kernel demonstration and its CUDA pseudo code.

in order to sustain the high throughput of the entire system. For example, the spatial Gaussian filter kernel consumes a high throughput input (800MB/s) by downsizing it to 36MB/s as shown in figure 3. Similarly, the kernels that upsize the throughput, such as the interpolation kernel, serve the system as a high speed data generator. Usually, the computations in these kernels are relatively simple ones that do not require any characteristic specific optimization. With the aid of these throughput kernels, the GPU implementation doubled or even tripled the speedups gained by the accelerating kernels.

In our design, with these three types of modules, a significant amount of data is generated and consumed within the GPU; the complex computations with high parallelism are accelerated on the GPU; the operations with high data dependency or the operations filled with branches are executed on the CPU. The GPU to CPU communication is minimized to the inevitable data transferring such as the original input video data loading and the final processed data storing.

### *B. CUDA Kernels*

In this section, we discuss how we implement and optimize the CUDA accelerating kernels. The principles of GPU acceleration are (1) provide the GPU enough threads; (2) maximize the operation to memory access ratio. The first principle is to ensure that the GPU always has a sufficient amount of threads. The second principle is to ensure that the GPU is able to hide the memory access [9].

In the implementation, we developed accelerating kernels for the computationally complex stages in the algorithm according to their specific computational characteristics and memory access patterns. Using this design methodology, each accelerating kernel can achieve high performance individually. For example, we highly optimized the temporal peak search kernel according to its temporal reduction and spatial independency characteristics. The temporal peak search kernel is based on the CUDA reduction method [10]. However, the conventional reduction method only computes an individual array for one final result. In the temporal peak search kernel, there are multiple peak values to be obtained spatially. Each peak value is computed by a CUDA reduction method. Therefore, we assigned multiple reductions operating on the GPU concurrently. As shown in figure 4, index $i$ maps the kernel to the video data spatially while index $j$ does so temporally. Most of the general reduction optimizations, such as using shared memory, hard code reductions and memory access coalescing (i.e. adjacent threads accessing adjacent memory locations), are still effective in our temporal peak search kernel. Thus, we applied these optimizations on the temporal peak search kernel as described in the CUDA pseudo code in figure 4. The optimized peak search CUDA kernel gains $11\times$ speedup against the non-optimized one.

### *C. Running on the GPU*

Due to the size limitation of the memories on GPU cards (usually 512MB-6GB), the input data is not able to be entirely transferred to the GPU memory at one time. For example, 1 second of video data (1024 frames, $100 \times 100$ pixels/frame) needs 2.25GB for the input array and another 2.25GB for the output array at the same time in the FFT stage. The GPU memory cannot provide such a large space for an entire 1 second of video data to be processed. Therefore, we divided the video data into several smaller chunks so that each chunk of data is able to stay on the GPU and be processed as shown in figure 5. Then the processed chunks will be assembled to produce a complete output video. After each chunk is processed, the graphic memory is released and loaded with the next chunk of data.

In this approach, we selected the appropriate chunk size to ensure that each chunk occupies the GPU cores efficiently. With multiple experiments, we discovered that the optimal

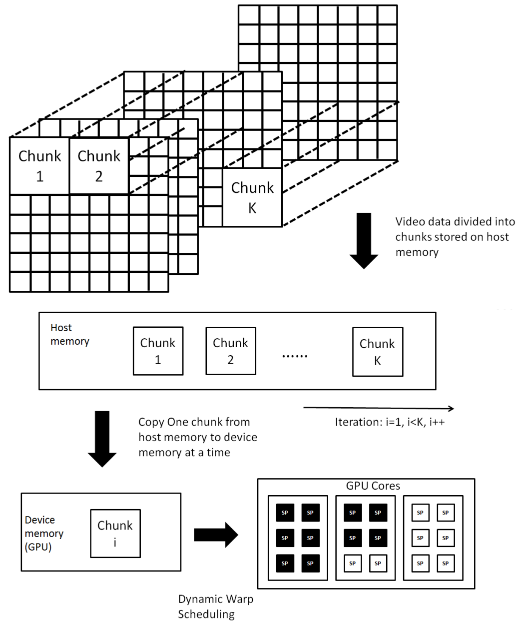


Fig. 5. Divide, process and assemble method.

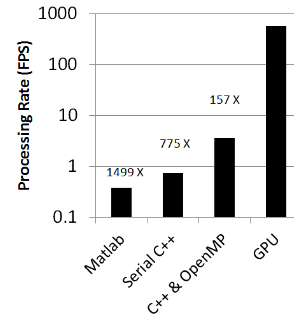


Fig. 6. The performance of the GPU implementation in comparison to the Matlab CPU implementation, the serial C++ implementation and the OpenMP C++ implementation.

chunk for our target GTX590 GPU is $40 \times 40 \times 1024$.

Furthermore, since the chunks are completely independent of each other, it is easy to adapt this approach on a multi-GPU system with a high scalability.

## IV. Performance

In this section, we present the performance of our GPU implementation. We report the performance of the GPU implementation in comparison to that of three different CPU implementations.

We chose NVIDIA Geforce GTX590 as the hardware to test our GPU implementation. Since GTX590 has the most CUDA cores among all the recent NVIDIA GPUs, it matches our high throughput design goal. The GTX590 graphic card has two GPU chips. In this implementation, we only utilized one GPU chip. However, our GPU implementation is potentially able to be ported on multi-GPU systems to gain even more speedup as discussed in section III-C.

The performance of our design is shown in figure 6. The results indicate that the GPU implementation has a significant speed up against the CPU implementations. Even in comparison to the OpenMP parallelized CPU implementation, the GPU implementation still performs $157.92\times$ faster.

## V. Conclusion

We have presented a GPU implemenation of the optical mapping algorithm for cardiac electrophysiology. We have reported a test of our implementation on the GTX590 GPU. The result indicates that the GPU implementation is significantly faster than the parallelized multi-core CPU implementation. We developed the CUDA kernels by considering the computational characteristics of the individual accelerating kernels and the features of the entire system throughput. Running the CUDA program on the GPU, we used a divide, process and assemble method to avoid the GPU memory space limitation.

## References

[1] S. Iravanian and D. J. Christini, Optical mapping system with real-time control capability, American Journal of Physiology - Heart and Circulatory Physiology, 2007, pp. H2605-H2611.

[2] H. N. Pak, Y. B. Liu, H. Hayashi, Y. Okuyama, P. S. Chen, and S. F. Lin, Synchronization of ventricular fibrillation with real-time feedback pacing: implication to low-energy defibrillation, American Journal of Physiology - Heart and Circulatory Physiology, 2003, pp. H2704-H2711.

[3] J. Fung, S. Mann, Using graphics devices in reverse: GPU-based Image Processing and Computer Vision, IEEE International Conference on Multimedia and Expo, 2008, pp. 9-12.

[4] M.C. Schatz, C. Trapnell, A.L. Delcher, A. Varshney, High-throughput sequence alignment using Graphics Processing Units, BMC Bioinformatics, 8:474, 2007.

[5] S.S. Stonea, J.P. Haldarb, S.C. Tsaoa, W.-m.W. Hwua, B.P. Suttonc and Z.-P. Liangb, Accelerating advanced MRI reconstructions on GPUs, Journal of Parallel and Distributed Computing Volume 68, Issue 10, October 2008, pp. 1307-1318.

[6] A. Ruiz, M. Ujaldon, J.A. Andrades, J. Becerra, Kun Huang, T. Pan, J. Saltz, The GPU on biomedical image processing for color and phenotype analysis, Bioinformatics and Bioengineering, 2007, pp. 1124-1128.

[7] Timothy D. R. Hartley, Umit Catalyurek, Antonio Ruiz, Francisco Igual, Rafael Mayo, Manuel Ujaldon, Biomedical image analysis on a cooperative cluster of GPUs and multicores, In ICS '08: Proceedings of the 22nd annual international conference on Supercomputing (2008), pp. 15-25.

[8] D. Sung, J. Somayajula-Jagai, P. Cosman, R. Mills, and A. D. McCulloch, Phase shifting prior to spatial filtering enhances optical recordings of cardiac action potential propagation, Ann Biomed Eng, 2001, vol. 29, pp. 854-61.

[9] S. Hong and H. Kim, An analytical model for GPU architecture with memory-level and thread-level parallelism awareness, In Proc. International Symposium on Computer Architecture, 2009

[10] D. Kirk and W. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, 2010.