# Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors

Michael Boyer*, David Tarjan*, Scott T. Acton†, and Kevin Skadron*
Departments of *Computer Science and †Electrical and Computer Engineering
University of Virginia, Charlottesville, VA 22904

## Abstract

*The availability of easily programmable manycore CPUs and GPUs has motivated investigations into how to best exploit their tremendous computational power for scientific computing. Here we demonstrate how a systems biology application—detection and tracking of white blood cells in video microscopy—can be accelerated by 200x using a CUDA-capable GPU. Because the algorithms and implementation challenges are common to a wide range of applications, we discuss general techniques that allow programmers to make efficient use of a manycore GPU.*

## 1. Introduction

The microprocessor industry has recently shifted from maximizing single-core performance to integrating multiple cores on a single processor die. The number of integrated cores is likely to continue increasing exponentially with Moore's Law for the foreseeable future. As core count grows rapidly while per-core performance grows more slowly, exploiting concurrency becomes essential for software to use multicore processors to their full potential and realize the benefits of semiconductor technology scaling.

Graphics Processing Units (GPUs) are notable because they contain many processing elements–up to 240 in NVIDIA's GTX 280–and provide a level of concurrency that cannot be found in any other consumer platform. Although GPUs have been designed primarily for efficient execution of 3D rendering applications, demand for ever greater programmability by graphics programmers has led GPUs to become general-purpose architectures, with fully featured instruction sets and rich memory hierarchies. Tools such as NVIDIA's CUDA have further simplified the process of developing general-purpose GPU applications. CUDA presents to the programmer a fairly generic abstraction of a manycore architecture supporting fine-grained parallelism. CUDA and the GPU therefore provide massive, general-purpose parallel computation resources with the potential for dramatic speedups.

It should be noted that GPUs only represent one possible direction for future manycore architectures. A variety of companies and researchers have described possible manycore architectures, most of which are currently envisioned as coprocessors on a chip or board separate from the main system CPU. Using CUDA and GPUs provides an opportunity to draw general lessons on how to best make use of manycore chips from the perspectives of both the programmer and the system architect.

This paper discusses our experiences in parallelizing a computationally intensive application from the field of systems biology: detection and tracking of rolling leukocytes in *in vivo* video microscopy of blood vessels [1], [2]. Tracking leukocytes provides researchers with important information about the inflammatory response of the vascular system. Unfortunately, manual tracking is a tedious process, requiring on the order of tens of hours of manual analysis for a single video [2]. Automated approaches to the detection [1] and tracking [2] of leukocytes obviate the need for manual analysis, but are computationally expensive, requiring more than four and a half hours to process one minute of video. Significantly reducing the runtime of these automated approaches would accelerate the process of developing anti-inflammatory medications.

This application was chosen partially because it demonstrates an urgent need for dramatic speedups. More importantly, however, it is a useful case study because it illustrates many of the issues that other applications will face in trying to use manycore systems. It is a complex application presenting nontrivial software engineering challenges as well as presenting a workload that is representative of a much broader class of applications. The application's runtime is dominated by a number of widely used operations, such as stencil-based operations (specifically feature extraction and image dilation) and an iterative solution procedure. These first two operations are widely used in image processing, and the last operation is widely used in high performance computing.

The detection and tracking algorithm was originally implemented in MATLAB, and re-implementing it in C resulted in a significant performance improvement. We further improved the performance by accelerating the most computationally demanding stages using CUDA and, for comparison, OpenMP. We achieved an overall speedup of 199.9x using a desktop system with an NVIDIA GeForce

GTX 280 GPU, compared to a speedup of 7.6x on the fastest available multicore CPU system. These speedups demonstrate the advantages of the throughput-oriented nature of GPUs. Additionally, we have identified a number of bottlenecks, in both hardware and software, whose elimination would enable even more significant speedups and simplify the development of efficient CUDA applications.

In addition to the substantial speedup, the main contribution of this paper is to describe in detail the algorithmic transformations needed to reduce the overheads associated with a separate coprocessor. These overheads are particularly acute with iterative algorithms such as iterative solvers. The best implementation required abandoning the canonical parallelization strategy suggested in the CUDA literature, in which each output value is computed by a separate thread. We also propose extensions to CUDA's software and hardware models that would provide better support for applications with fine-grained interleaving of serial and parallel regions.

## 2. CUDA

NVIDIA's CUDA [3] architecture allows programmers to use the C programming language to develop general-purpose applications exploiting fine-grained parallelism. CUDA is currently supported only on NVIDIA GPUs but recent work has shown the viability of compiling CUDA programs for execution on multi-core CPUs [4]. Nickolls *et al.* [3] provide a comprehensive overview of the CUDA programming model. We only touch on the most important features here and refer the reader to their work for more details.

CUDA consists of a runtime library and an extended version of C. The main abstractions on which CUDA is based are the notion of a *kernel* function, which is a single routine that is invoked concurrently across many thread instances; a software controlled scratchpad, which CUDA calls the "shared memory", in each SIMD core; and barrier synchronization. CUDA presents a virtual machine consisting of an arbitrary number of *streaming multiprocessors* (SMs), which appear as 32-wide SIMD cores with a total of up to 512 thread contexts (organized into *warps* of 32 threads each). Kernels are invoked on a 2D *grid* that is divided into as many as 64K 3D *thread blocks*. Each thread block is mapped in its entirety and executes to completion on an arbitrary SM. All thread blocks in a kernel run to completion before a subsequent kernel may start, providing an (expensive) global memory fence.

Once a kernel is launched, a hardware scheduler assigns each thread block to an SM with sufficient spare capacity to hold the entire thread block. If multiple (small) thread blocks fit onto a single SM, they will execute concurrently but cannot communicate with or even be aware of the existence of their co-resident thread blocks. Warps are multiplexed onto the SIMD hardware on a cycle-by-cycle granularity according to their execution readiness. Each thread is completely independent, scalar, and may execute arbitrary code and access arbitrary addresses. Execution is most efficient if all threads in a warp execute in lockstep; divergence is handled with a branch stack and masking. Similarly, memory access is more efficient if threads within a warp access contiguous memory locations.

## 3. Leukocyte Detection and Tracking

Leukocytes, or white blood cells, play an important role inside the body, acting as the body's defense mechanism against infection and cellular injury. Much effort has been invested in studying the way leukocytes carry out this role in the process of inflammation. The most commonly used statistic predicting the level of cell recruitment during inflammation is the velocity distribution of rolling leukocytes [5], [6]. This distribution can help researchers gain the requisite knowledge about the mechanisms behind leukocyte rolling and arrest to create effective inflammation treatments. As a result, researchers investigating anti-inflammatory drugs need a fast, accurate method of attaining these measurements to test the validity of their treatments.

Currently velocity measurements are taken manually. Researchers go through hours of video data frame-by-frame, marking the centers of rolling leukocytes [5], [6], [7] at an average rate of several minutes per leukocyte. To obtain a valid estimate of the leukocyte velocity distribution hundreds of cells must be tracked. This process requires many tiresome hours and, like any human action, involves a certain amount of observer bias. A method to automatically track cells would solve both these problems and allow researchers to focus more on the problem of creating treatments and less on the tabulation of data. Furthermore, the possibility of real-time leukocyte detection and tracking would give researchers the ability to immediately view the results of their experiments. This would enable a researcher to vary experimental parameters until appropriate results are obtained, instead of having to conduct many different experiments separated by periods of data tabulation and analysis.

Automatic detection is accomplished using a statistic called the Gradient Inverse Coefficient of Variation (GICOV) [1]. The GICOV computes, along a closed contour, the mean outward gradient magnitude divided by the standard deviation of this measure. In the implementation used in this paper, the contours are restricted to circles of a known range of radii. In the first image of a sequence, detection is performed on the whole image. Following the initial detection, subsequent detections only need to be performed in a small window at the entry side of the venule. After detection, an active contour (snake) algorithm is used to track the boundary from frame to frame using a statistic called the Motion Gradient Vector Flow (MGVF) [2]. The MGVF is a gradient field biased in the assumed direction

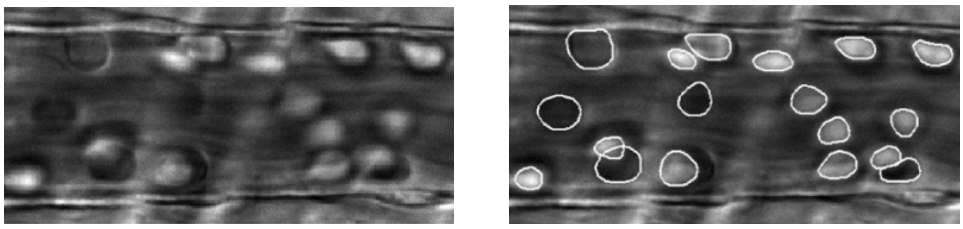

Figure 1. Still image from an intravital video of a mouse cremaster muscle [1]. On the left is the original image; on the right is the result of automatic detection with the leukocytes outlined.

of the movement of the leukocytes. This active contour method works well in the cluttered, contrast-varying scene encountered in intravital microscopy. The snake is tailored to the leukocyte model and is constrained to prefer circular shapes of a radius near the average radius for leukocytes of a given species.

The images used for detection and tracking are of leukocytes found in vivo, that is, within a living organism. The videos are made using intravital microscopy, filming the cremaster muscle of a mouse. This muscle is particularly thin, making it transparent, and is filled with post capillary venules. Part of a frame from such a movie is shown in Figure 1. These intravital images present a salient challenge for automated image analysis.

The particular video used in this work is a 640x480 uncompressed AVI file. The actual blood vessel being analyzed only occupies a third of each frame, so a frame is cropped to a 218x480 sub-frame before detection is performed. The cropping boundary is hard-coded for all performance measurements, although in practice it would be designated manually by the user in the first frame. During the tracking stage, only a small, fixed-sized area around each cell is analyzed, so the performance of the tracking stage is a function of the number of cells being tracked rather than the size of the frame. The video was recorded at 30 frames per second (FPS), so achieving real-time analysis would require processing each frame in 1/30th of a second.

## 4. Accelerating the Detection Stage

In order to automatically detect leukocytes in an image, three operations are performed. First, for each pixel in the image, the GICOV score is computed for circles of varying radii (stencils) centered on that pixel, and the maximum of those scores is output into a matrix. Second, this matrix is dilated, which simplifies the process of determining if the GICOV score at a given pixel is the maximum within that pixel's neighborhood. Third, for those pixels which have locally maximum GICOV scores, an active contour is used to refine the initial circle and more precisely determine both the location and the shape of the leukocyte.

Previous work implemented both the detection [1] and tracking [2] stages of the algorithm using MATLAB. In the detection stage of that implementation, the GICOV computation and dilation take 36.7% and 28.2% of the overall runtime, respectively. In our C implementation of the algorithm, these two operations further dominate the execution, taking 59.1% and 39.2% of the runtime, respectively. The C implementation is essentially a line-by-line translation of the MATLAB implementation and provides a speedup of 2.2x on the detection stage. To further improve the performance, we accelerated the critical operations using OpenMP and CUDA. The OpenMP acceleration was achieved with the introduction of two simple pragmas. The CUDA acceleration was more complex, starting with a straightforward translation and then applying increasingly complex optimizations. The speedups achieved by the C, OpenMP, and CUDA implementations of the detection stage are shown in Figure 2.

Runtimes for all implementations were measured on a machine running Ubuntu version 7.10 with a 3.2 GHz quad-core Intel Core 2 Extreme X9770 processor and an NVIDIA GeForce GTX 280 GPU, with NVIDIA driver version 177.67. The original implementation was executed in MATLAB version 7.6.0.324. The C code was compiled using GCC version 4.2.3 and the CUDA code was compiled using NVCC version 0.2.1221, CUDA Toolkit version 2.0, and CUDA SDK version 2.0 Beta2. The first access to the CUDA API incurs a non-negligible delay due to initialization overhead. Because a real-time implementation would initialize the API before the video capturing begins, and because this delay can vary significantly between different runs, we started measuring the runtime after a dummy call to the API.

### 4.1. Comparison to OpenMP

A popular approach to parallelizing programs on shared-memory machines is the OpenMP standard. In order to provide a point of comparison to our CUDA implementations, we have also parallelized the leukocyte detection using OpenMP. Specifically, the for loops in the GICOV computation and dilation functions that iterate over the pixels in the image were augmented with a `parallel`

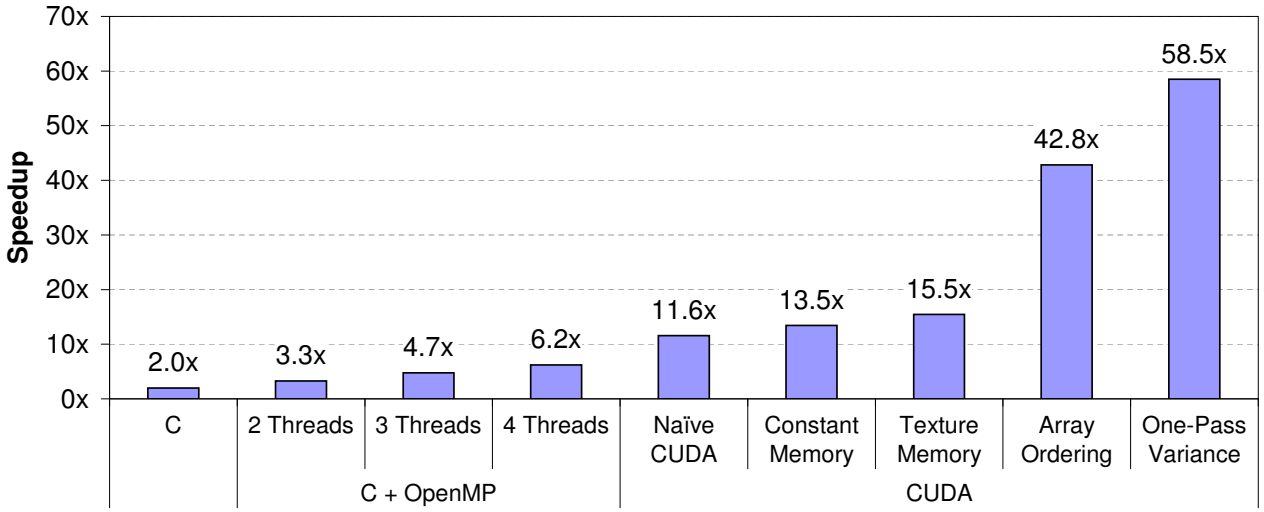


Figure 2. Speedup of the different implementations of the detection stage over the original MATLAB implementation.

pragma. This transformation was trivial because the for loops contain no inter-loop dependencies. The OpenMP speedups for two, three, and four threads are shown in Figure 2. With four threads, the OpenMP implementation achieves speedups of 6.8x and 3.1x over the original MATLAB and C implementations, respectively.

### 4.2. Naïve CUDA Implementation

The CUDA implementations parallelize exactly the same loops as in the OpenMP approach. The code inside the nested for loops in each function was converted directly into a kernel function, and the domains of the kernel functions were defined to be the pixels in the image. This straightforward CUDA implementation achieves a 5.9x speedup over the original C version.

### 4.3. CUDA Optimizations

Although the naïve CUDA implementation achieves a non-trivial speedup over the sequential version of the application, it was written without taking into account the unique architecture of the GPU. A number of optimizations were applied to the original CUDA implementation that significantly improved its performance. Each optimization is described in turn. Note that the optimizations are cumulative, meaning that once an optimization has been applied, it remains in effect in all subsequent optimizations. However, they are independent of each other and could be applied in any order. For each optimization, we also note its applicability to programs parallelized using OpenMP.

**Constant Memory:** Many of the arrays accessed by both kernels are read-only. Thus, they can be allocated in the GPU's constant memory address space, which allows them to be cached on-chip. Accomplishing this change in CUDA is trivial, but it allows the code to achieve a speedup of 6.8x over the original C version and 1.2x over the naïve CUDA version. This optimization is not applicable to the OpenMP version, since CPU architectures do not provide such special purpose address spaces.

**Texture Memory:** GPUs also employ another special-purpose address space for texture memory. Like constant memory, texture memory only supports read-only data structures. Data structures mapped to texture memory can take advantage of special hardware on the GPU which provides a small amount of on-chip caching and slightly more efficient access to off-chip memory. By moving the large arrays accessed by the two kernels into texture memory, the application achieves a speedup of 7.8x over the original C version and 1.2x over the previous CUDA version. As texture memory is an architectural feature of GPUs, this optimization is not applicable to the OpenMP version.

**Array Ordering:** The two largest arrays accessed by the GICOV kernel were originally allocated in row-major order. The memory access pattern of the kernel resulted in threads within the same warp accessing non-contiguous elements of the arrays. Allocating the arrays in column-major order allows threads within the same warp to access contiguous elements, which can significantly improve performance due to the GPU's ability to coalesce multiple contiguous memory accesses into one larger memory access. With this optimization, the detection stage achieves a speedup of 21.7x over the original C version and 2.8x over the previous CUDA version. Programs using OpenMP on manycore CPUs such as Sun Niagara 2 [8] and Intel Larrabee [9] can benefit from this optimization, as it makes more efficient use of the L1 data cache and memory bandwidth when working on many data points in parallel. It does not impact the running time of

our OpenMP implementation, however, as each heavyweight core processes a single data point at a time and can fully buffer the array in the L1 cache in both layouts.

**One-Pass Variance:** For each point in the image, the GICOV kernel computes the sum of a function at 150 different points and then computes the variance of the function across those same points. This two-pass approach is inefficient because it requires storing the 150 intermediate values, which requires spilling those values to global memory[1]. The variance can instead be computed in a single-pass using a relatively straightforward algorithm [10]. This optimization provides an overall speedup of 29.7x over the original C version and 1.4x over the previous CUDA version. We experimented with this optimization in the OpenMP version and observed no speedup, since the L1 cache in each CPU core is large enough to buffer the 150 intermediate values.

## 5. Accelerating the Tracking Stage

After the locations of leukocytes in frame $i$ have been determined by the detection stage of the algorithm, this information is used by the tracking stage to determine the new locations of those same leukocytes in frame $i+1$. These updated locations are then fed back into the tracking stage to determine the new cell locations in frame $i+2$. This process continues, with detection typically performed once every 10 frames.

In each frame, all cells can be processed independently. For each cell, the algorithm only analyzes a fixed-sized portion of the frame (41x81 pixels for the particular leukocytes studied in this work), centered around the cell's location in the previous frame. This explicitly limits the maximum velocity at which a cell can be successfully tracked. Within the sub-image of interest, two operations are performed. First, the Motion Gradient Vector Flow (MGVF) matrix is computed via an iterative Jacobian solution procedure. The solver iterates until it has met a convergence criterion, which is a function of all of the elements in the matrix. Second, an active contour minimizes an energy function defined on the MGVF matrix and computes the new location of the leukocyte.

In the original MATLAB implementation of the tracking stage, 93.5% of the execution time is spent in the iterative solver. In the C implementation, the iterative solver consumes essentially all (99.8%) of the overall runtime. The C implementation provides a speedup of 2.0x over the MATLAB implementation. As with the detection stage, the tracking stage was accelerated further using OpenMP and CUDA. The runtime of each of the different implementations was measured on the same system as described in Section 4. The speedups achieved by the C, OpenMP, and CUDA implementations over the original MATLAB implementation are shown in Figure 3.

1. Another option would be to spill to the on-chip shared memory. Because the shared memory is small, however, this approach reduces the number of threads per SM and reduces overall performance.

### 5.1. Comparison to OpenMP

Accelerating the tracking stage with OpenMP was a relatively straightforward transformation. Since each cell being tracked can be processed in parallel, we simply added a `parallel` pragma to the for loop that iterates over all of the cells. Because the number of cells is small (generally less than 50), this approach would not be effective if we attempted to scale the OpenMP implementation to much larger numbers of processors. For the hardware on which we benchmarked the implementation, however, the decomposition was good enough to achieve nearly linear scaling. The OpenMP speedups for two, three, and four threads are shown in Figure 3. With four threads, the OpenMP implementation achieves speedups of 7.7x and 3.8x over the original MATLAB and C implementations, respectively.

### 5.2. Naïve CUDA Implementation

Because the runtime of the tracking stage is dominated by calls to the iterative solver, which in turn is dominated by calls to a regularized version of the Heaviside function, the first CUDA implementation simply replaced each call to the Heaviside function with a call to a Heaviside CUDA kernel. In this implementation, each element in the output matrix is computed by a single thread. Although the overall kernel execution time is slightly less than one second, the memory allocation and copying overheads add more than eleven seconds to the overall runtime. Due to these overheads, this implementation achieves a 2.6x *slowdown* compared to the original C implementation (and is actually slower than the MATLAB implementation). Parallelizing the OpenMP implementation at the granularity of individual calls to the Heaviside function similarly resulted in a significant slowdown.

### 5.3. CUDA Optimizations

As with the detection stage, a number of optimizations were applied to the naïve CUDA implementation of the tracking stage in order to improve its performance. For each implementation, Figure 4 shows the overall runtime, as well as the fraction of the runtime devoted to kernel execution, memory copying, memory allocation, and non-CUDA related code. Note that the optimizations are again cumulative, but unlike in the detection stage, they are mostly dependent on one other, since they change how and when memory is allocated and when data is moved to and from the GPU.

**Larger Kernel:** In the naïve implementation, the Heaviside kernel is called eight times during each iteration of

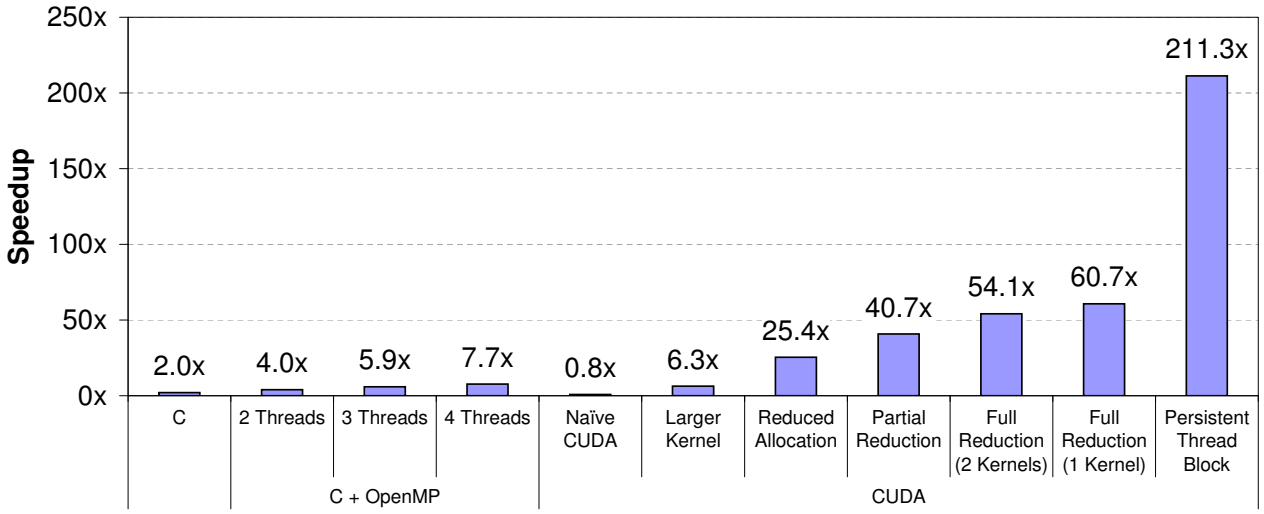


Figure 3. Speedup of the different implementations of the tracking stage over the original MATLAB implementation.

the solver. In order to reduce the memory allocation and copying overhead and the number of kernel calls and also increase the amount of useful work performed in each kernel call, the entire body of the inner loop was converted into a single CUDA kernel. As in the previous implementation, each element in the output matrix is computed by one thread. Applying this optimization yields an overall speedup of 3.1x over the original C implementation and 8.1x over the previous CUDA implementation. Recall that the OpenMP implementation is parallelized across the cells being tracked. If we instead parallelize that implementation across the individual matrix elements in the iterative solver, as is done here for the CUDA implementation, the OpenMP implementation actually becomes 24% slower.

**Reduced Allocation:** Allocating and deallocating memory on the CPU via the C standard library functions `malloc` and `free` is a very low overhead operation. Allocating and deallocating memory on the GPU via the CUDA library functions `cudaMalloc` and `cudaFree`, however, is considerably more expensive. On the system used in this study, we measured the overhead of `cudaMalloc` to be approximately 30-40 times greater than the overhead of `malloc` (and significantly higher for memory sizes larger than a few megabytes) and the overhead of `cudaFree` to be approximately 100 times greater than the overhead of `free`. This overhead is readily apparent in Figure 4 for both the naïve CUDA and larger kernel implementations, whose runtimes are dominated by memory allocation.

In order to minimize this overhead, instead of allocating and freeing memory on the GPU once each iteration of the solver, initialization and cleanup functions were added to allocate memory a single time at the start of the iterative solver and then free memory at the end. Applying this optimization yields an overall speedup of 12.6x over the original C implementation and 4.0x over the previous CUDA implementation. Note that even if the C standard library memory allocation functions were as expensive as the CUDA equivalents, this overhead would be negligible in the OpenMP implementation because it does not allocate memory within the iterative solver loop.

**Partial Reduction:** After each iteration of the solver, the average of the absolute value of the change at each pixel is computed in order to check for convergence. In the previous CUDA implementation, the entire MGVF matrix is copied back after each iteration, and the reduction is performed entirely on the CPU. In order to improve the performance of the reduction, the kernel was extended to perform a partial reduction, in which each thread block computes the sum of the absolute value of the change of each each pixel within that thread block. With a thread block size of $N$ threads, this reduces by a factor of $N$ both the amount of memory copied from the GPU to the CPU as well as the number of additions required by the CPU to perform the reduction. Since typical values of $N$ in CUDA applications are 128 and 256, performing a partial reduction on the GPU can result in a substantial performance improvement. In this application, applying the optimization yields an overall speedup of 20.3x over the original C implementation and 1.6x over the previous CUDA implementation. This and the next two optimizations do not apply to the OpenMP implementation because it does not transfer data between disjoint memory spaces.

**Full Reduction (2 Kernels):** In order to further reduce the reduction and memory copying overheads, a second CUDA kernel was added to complete the reduction on the GPU. This allows the copying of the partial sums to be replaced by the copying of a single Boolean value indicating whether or not the computation has converged. However,

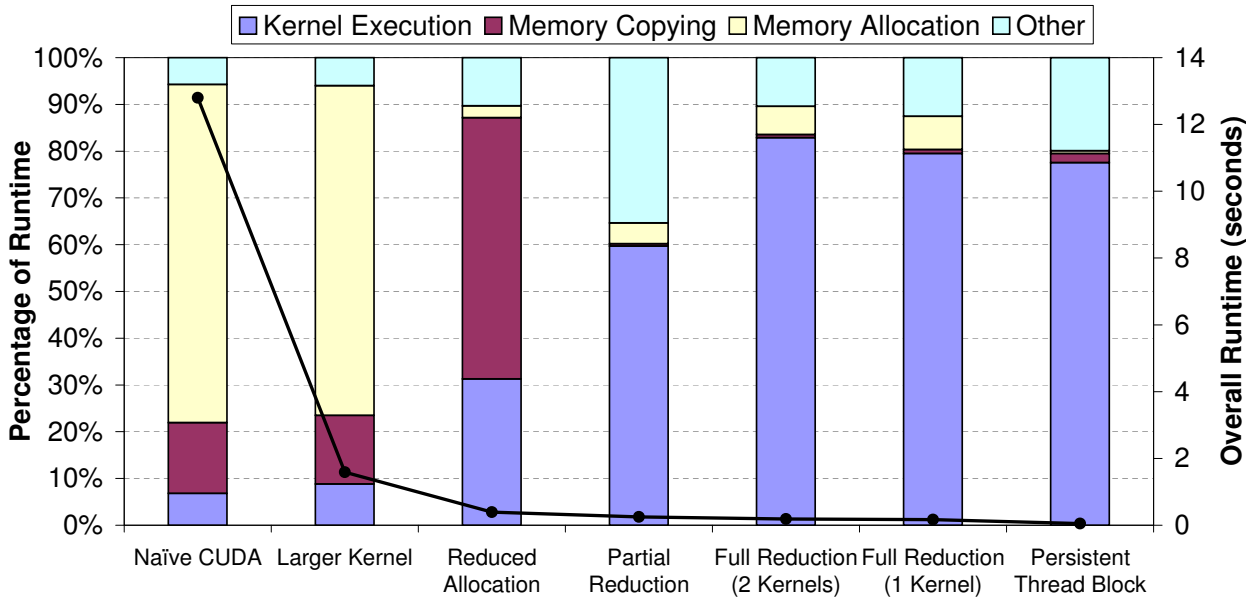


Figure 4. Impact of different overheads on the runtimes of the CUDA implementations. Each bar shows, starting at the bottom, the percentage of runtime due to: executing the CUDA kernels, transferring memory between the CPU and GPU, allocating memory on the GPU, and executing the other, non-CUDA related code. The line indicates the overall runtime of each implementation.

this approach does not improve performance significantly because, although it does reduce the amount of data copied, it does not reduce the number of copies performed. At data sizes less than about four kilobytes, the latency of a memory transfer is essentially constant regardless of the amount of data transferred.

Thankfully, performing the entire reduction on the GPU enables a further optimization. Instead of checking the convergence flag after each iteration, the computation kernel can be modified to check the value of the flag and exit if convergence has already been achieved. This allows the computation and reduction kernels to be called an as many times as desired without the need to explicitly copy the convergence flag and without impacting the correctness of the results. In other words, this allows the main loop of the iteration to be unrolled to an arbitrary degree. In our experiments, performing about 30 back-to-back kernel calls before copying the convergence flag resulted in the best performance. Applying these optimizations yields an overall speedup of 26.9x over the original C implementation and 1.3x over the previous CUDA implementation.

**Full Reduction (1 Kernel):** Although the previous optimization reduces the overall runtime, it actually increases the absolute runtime devoted to kernel execution, due to both the increase in computation performed by the kernels as well as the doubling of the number of kernel calls. To reduce the kernel overhead, the computation and reduction kernels can be merged into a single kernel. However, we must be careful about the ordering of the computation and reduction in the merged kernel. A seemingly reasonable approach would be to compute the updated MGVF matrix at the beginning of the kernel and then perform the reduction at the end of the kernel. Unfortunately, this would require the use of a global memory fence in order to ensure that all thread blocks had finished their computations before the reduction was performed, and CUDA does not provide such a fence except across kernel calls.

To avoid this potential deadlock, in each kernel call we first perform a reduction on the values produced by the previous kernel call. Only then do we proceed to compute the next iteration (if the computation has not already converged). Applying this optimization yields an overall speedup of 30.2x over the original C implementation and 1.1x over the previous CUDA implementation.

**Persistent Thread Block:** In the previous implementation, about 24% of the time spent by the application waiting for kernel execution is due to the overhead of kernel invocation, with only 76% of the time due to actually performing useful work on the GPU. To reduce the overhead of kernel execution, we can perform all of the iterations in a single kernel call. As mentioned earlier, CUDA only provides a per-thread-block memory fence, not a global memory fence. Thus, in order to perform all of the iterations in a single kernel call, we must perform all of the computation for one cell within a single thread block. Since a single thread block can contain no more than 512 threads, and there are

more than 3,000 elements in the MGVF matrix, we must abandon the one-to-one mapping between threads and matrix elements. Instead, within each iteration, the single thread block traverses the entire matrix, computing a subset of the matrix in each step.

If we simply modify the kernel to perform all of the iterations for a cell in a single kernel call but still process the individual cells sequentially, the application will not effectively take advantage of the GPU's parallel computation resources and the resulting performance will be significantly worse than the previous implementation. However, since each cell now only requires a single thread block, it makes sense to process all of the cells concurrently, with one thread block allocated for each cell. The entire tracking stage for one frame can then be completed with a single kernel call. Implementing this optimization yields an overall speedup of 105.2x over the original C implementation and 3.5x over the previous CUDA implementation. Note that the OpenMP parallelization uses essentially the same approach, but with only a single thread processing each cell rather than an entire thread block.

## 6. Discussion

The final CUDA implementation of the detection and tracking algorithm provides a speedup of 80.8x over the single-threaded C implementation[2]. Even assuming perfectly linear scaling, matching the performance of this CUDA implementation with the OpenMP implementation would require about 80 CPU cores equivalent to the cores used in our experiments. Given the choice to obtain the same speedup by purchasing either 20 quad-core processors (and associated hardware) or a single GPU, the most cost-effective choice is clearly the GPU. Of course, in practice we have been unable to achieve perfectly linear scaling with OpenMP on this problem due to the relatively small sizes of the computations involved, and we would be unable to match the performance of the GPU with any number of additional CPU cores.

All of the performance results presented so far have been expressed relative to the performance of other implementations. To provide a sense of how close each implementation comes to achieving real-time analysis, Figure 5 shows the number of frames of video that each implementation can process per second. While the MATLAB, C, and OpenMP implementations cannot even process a single frame per second, the CUDA implementation can process more than twenty. Given the increases in GPU performance expected in the next few years, real-time detection and tracking of leukocytes appears realizable in the near future with commodity hardware.

2. To compute the performance of the entire application, we assume that detection is performed once every ten frames. Thus, the average time to process one frame can be estimated by $(D + 9T)/10$, where $D$ and $T$ are the average times to perform detection and tracking, respectively, on a single frame.

### 6.1. Lessons for CUDA Developers

We encountered a number of significant bottlenecks while attempting to improve the performance of both the detection and tracking stages using CUDA. Most of the bottlenecks are a result of limitations of the underlying hardware and software and are not fundamental limitations of the CUDA programming model. Later we will suggest ways in which system architects can significantly reduce or even eliminate some of these bottlenecks, but here we focus exclusively on techniques that allow CUDA application developers to bypass these bottlenecks to some extent.

**Reduce Kernel Overhead:** We have shown earlier that the overhead of launching a kernel can severely impact the performance of a CUDA application. This is clearly evident when we compare the performance of the naïve CUDA implementations of the two different stages of the algorithm. In the detection stage, the most natural decomposition was at a coarse-grained level, resulting in only two kernel calls per frame. In the tracking stage, however, the most natural decomposition was at a much finer-grained level, resulting in approximately 50,000 kernel calls per frame. As a result, only 0.1% of the time spent waiting for the execution of the GICOV kernel in the detection stage is caused by the kernel invocation overhead, with 99.9% of the time spent performing actual computation on the GPU. Conversely, 73.1% of the time spent waiting for the execution of the Heaviside kernel in the tracking stage is caused by the kernel overhead, with only 26.9% of the time spent performing actual computation. Thus, regardless of how much we were able to improve the performance of the Heaviside kernel, we would not be able to reduce the overall runtime of the kernel by more than 26.9%. In order to reduce the impact of this overhead, developers should attempt to make their kernels as coarse-grained as is feasible, thereby increasing the amount of work performed in each kernel call and reducing the total number of kernel calls.

There is also a performance advantage due to launching many kernels back-to-back. For example, in both full reduction implementations of the tracking stage, the overhead of kernel invocation is significantly more severe without unrolling the iterative solver loop. This is because, in the most recent versions of the CUDA API, kernel invocations are asynchronous. With unrolling, multiple kernel calls are batched in the GPU driver, and the application can overlap kernel execution on the GPU with accessing the driver on the CPU. Without unrolling, there is an implicit synchronization when the convergence flag is copied back to the CPU after each kernel call, and there is no overlap between kernel execution and driver access.

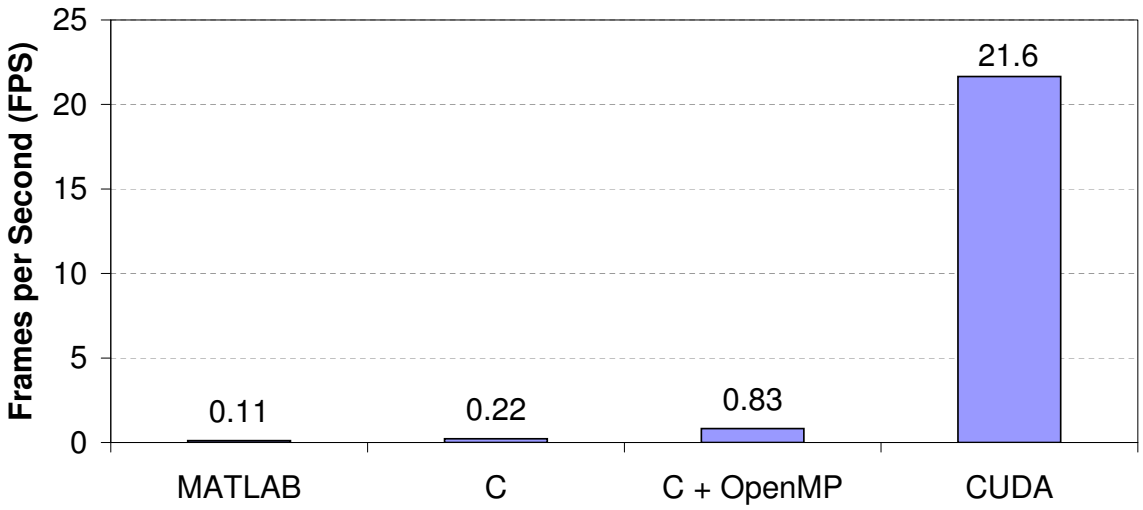


Figure 5. Overall rates at which the four implementations can detect and track leukocytes.

**Reduce Memory Management Overhead:** Although most programmers have learned that memory allocation is a relatively inexpensive operation, this assumption is no longer valid in the context of CUDA. As mentioned earlier, `cudaMalloc` and `cudaFree` are approximately 30-40 and 100 times more expensive than their equivalent C standard library functions `malloc` and `free`. The results for the naïve CUDA and larger kernel implementations of the tracking stage demonstrate this clearly. Allocating memory on the GPU consumes approximately 72% and 71%, respectively, of the runtimes of those two implementations. The solution here is straightforward: wherever possible, allocate GPU memory once at the beginning of an application and then reuse that memory in each kernel invocation.

**Reduce Memory Transfer Overhead:** Another inefficiency caused by the disjoint address spaces of the CPU and GPU is the need to explicitly transfer data between the two memories. The transfer overhead can be significant: in the reduced allocation implementation of the tracking stage, memory copying consumes 56% of the overall runtime. To reduce the severity of this overhead, developers should attempt to perform as much computation on the GPU as possible. For example, in the partial reduction implementation, the convergence condition is partially computed on the GPU in order to reduce the memory transfer overhead. With this change the number of elements transferred decreases from the number of elements in the matrix (generally 3,321) to the number of thread blocks (52 in this case). It is important for developers to understand that accelerating a computation using CUDA does not have to be an all-or-nothing proposition. Even if an entire computation cannot be (easily) implemented using CUDA, it is possible that offloading only a part of the computation will improve the overall performance.

Note also that moving a computation to the GPU may prove beneficial even if that computation would be more efficiently executed on the CPU. To further reduce the memory copying overhead of the partial reduction implementation, the two-kernel full reduction implementation uses a second kernel to finish summing the partially reduced values produced by the first kernel. Even though the second kernel is invoked with only a very small number of threads, which certainly perform the reduction significantly slower than would a CPU thread, overall the change improves the application's performance because the reduction in the memory transfer overhead outweighs the increase in computation time. CUDA implicitly encourages developers to fill the GPU with thousands of threads, so that they are trained to think that they are wasting the GPU's computational resources if they use only a small number of threads. However, as we have seen here, it is sometimes advantageous to accept computational inefficiency in exchange for a reduction in memory transfer overhead.

**Understand Memory Access Patterns:** CPUs are designed to reduce the effective memory access latency through extensive caching. Thus, a slightly irregular memory access pattern, such as the one exhibited by the stencil operation used in the GICOV computation, can be successfully captured by the CPU's caches. However, that same access pattern may be irregular enough to prevent efficient utilization of the GPU's memory bandwidth, because the restrictions on access patterns that must be met in order to achieve good memory performance are much more strict on a GPU than they are on a CPU. This is evident in the GICOV kernel of the detection stage. In the original implementation, the input matrices are allocated in row-major order, so access would be most efficient if neighboring threads access neighboring elements from the same row.

However, the access pattern actually exhibited by the kernel is that neighboring threads access neighboring elements from the same *column*. This explains why allocating the input matrices in column-major order provides a 2.8x speedup. The same change in the CPU version does not significantly impact the runtime because the caches are large enough to capture the entire stencil regardless of the order of traversal.

These access pattern restrictions can be partially relaxed by taking advantage of the GPU's special-purpose address spaces. Both constant and texture memory provide small on-chip caches that allow threads to take advantage of fine-grained spatial and temporal locality. In addition, texture memory relaxes the alignment requirements that must be met in order for multiple memory accesses from within the same warp to be coalesced. Another effective approach is to use the software-controlled shared memory as an explicitly managed cache, which can significantly improve performance when data elements are frequently reused among threads in the same thread block.

**Tradeoff Computation and Memory Access:** The GICOV and dilation kernels used in the detection stage perform relatively simple computations across a large number of data elements. Thus, their performance is more a function of the GPU's memory system performance than its processing performance. It can be beneficial for such memory-bound kernels to decrease the number of memory accesses required by increasing the complexity of the computation. Such a case arises in the GICOV kernel, which at each pixel and for each stencil computes the variance of a function across the 150 sample points within that stencil. The original CUDA implementation computes the variance in two passes. Since computing each point in the function requires accessing global memory, implementing a single-pass algorithm for computing the variance essentially halves the number of memory accesses. Even though the single-pass algorithm significantly increases the complexity of the variance computation, it provides a 1.4x speedup over the two-pass algorithm because the impact of the reduction in memory usage far outweighs the impact of the increased computational complexity. Similar transformations are likely to be possible for other memory-bound kernels.

**Avoid Global Memory Fences:** As discussed earlier, CUDA does not provide a global, inter-thread-block memory fence. Thus, if multiple thread blocks need to communicate, they must do so across kernel calls. This would not present a problem if the overhead of kernel invocation were not so high. In the two-kernel full reduction implementation of the tracking stage, a global memory fence is needed in each iteration between the matrix computation and the convergence check. This fence is implemented by creating separate kernels for the two steps. Unfortunately, this doubles the number of kernel calls, which limits the overall performance. As described earlier, one technique for reducing the number of kernel calls is to switch the order of the two steps and combine them into a single kernel, so that the convergence check occurs before the matrix computation in each iteration. Although this introduces redundant computation, since the final step in the reduction is performed by each thread block instead of by a single thread block, the reduction in the kernel overhead produces an overall speedup of 1.1x over the two-kernel implementation. This technique is generally applicable to any iterative solver that uses a convergence criterion for early exit.

Although this approach reduces the number of kernel calls by a factor of two, it still requires the use of a global memory fence after each iteration. This is because there is a one-to-one mapping between threads and matrix elements, and the number of matrix elements is larger than the maximum size of a thread block. The thread mapping scheme used here is typical in CUDA programs, because CUDA developers are encouraged to make their threads as fine-grained as possible in order to fully utilize the GPU's vast computational resources. However, abandoning this canonical thread mapping and instead using only a single thread block allows an arbitrary number of iterations to be computed in a single kernel call without the need for a global fence. As long as there are enough independent computations (corresponding to individual cells in this work) to occupy most or all of the SMs, this approach can provide significant speedups. Note also that the performance advantage increases as the number of iterations of the solver increases. Thus, the slower the computation converges, the more advantageous it becomes to use a single, persistent thread block for each independent computational unit.

## 6.2. Lessons for System Architects:

As shown above, there exist techniques for avoiding many of the performance bottlenecks that a CUDA developer may encounter. A more effective approach, however, would be for system designers to avoid introducing such bottlenecks altogether, or at least reduce the impact of those bottlenecks. We suggest a number of approaches that a system architect can take, at both the hardware and software levels, to reduce the amount of effort required for developers to obtain satisfactory performance. Removing some of the barriers to high performance will help speed the adoption of CUDA and other GPGPU programming models.

**Streamline Memory Management:** Perhaps the simplest bottleneck to address would be the slow memory management provided by the CUDA API. As noted earlier, the `cudaMalloc` and `cudaFree` functions are significantly slower than the equivalent C standard library functions, `malloc` and `free`. If the CUDA memory allocation functions were as fast as the equivalent C standard library functions, the larger kernel implementation of the tracking stage would provide a 2.5x speedup over the best OpenMP implementation instead of the 1.2x *slowdown* that it actually

provides. Thus, with a relatively straightforward translation to CUDA and without any complex optimizations, this CUDA implementation would have been adequate to provide better performance than the best CPU implementation. Reducing the overhead of memory management would both simplify the process of achieving satisfactory speedups with simple implementations and enable even more impressive speedups with complex implementations.

The inefficiency of memory allocation may be a byproduct of the fact that most graphics applications tend to allocate memory both in large chunks and on an infrequent basis. Thus, there traditionally has been little incentive for the authors of graphics drivers to optimize the memory management functions. With increased adoption of CUDA and other GPGPU programming models, it becomes more important to address these inefficiencies. Since the allocation functions cannot be executed on the GPU but instead must be executed on the CPU, the driver on the CPU should be able to maintain tables of allocated and available memory without any interaction with the GPU. Thus, there seems to be no fundamental reason that the CUDA functions and the C standard library functions cannot be implemented in the same way and achieve the same level of performance.

**Provide a Global Memory Fence:** CUDA's lack of an inter-thread-block global memory fence forced us to use a non-intuitive implementation strategy in order to achieve the most significant speedup on the tracking stage. The use of a persistent thread block runs counter to the standard CUDA development strategy of making threads as fine-grained as possible. If CUDA provided an inter-thread-block memory fence, the full reduction implementation could have achieved significantly better performance without the need to abandon the one-to-one mapping between threads and matrix elements. Assuming that the overhead of the fence would be negligible in comparison to the overhead of the computation itself, using a memory fence in the full reduction implementation instead of multiple kernel calls would speed up that implementation by 1.3x.

Without detailed knowledge of the GPU's microarchitecture, it is difficult to assess the complexity of implementing a global memory fence. One required change is clear, however. In the general case, implementing a global fence in CUDA would require thread blocks that reach the fence to yield to thread blocks that are still waiting to begin execution, in order to ensure forward progress when there are more thread blocks than can execute concurrently on the GPU. Yielding a thread block would require each thread to write its current state to memory. For small numbers of thread blocks, this would be relatively inexpensive. However, the CUDA specification allows a kernel to be invoked across more than four billion thread blocks of up to 512 thread blocks each. Clearly the GPU's memory would not be large enough to store the state for so many threads, and thus an application using a global memory fence would require a much lower limit on the number of threads per kernel invocation. For many applications, this would be an acceptable tradeoff.

**Add Caches:** The GPU's use of on-chip caches for the constant and texture memory spaces allows developers to achieve good memory performance even with kernels whose memory access patterns are slightly irregular. Unfortunately, in order to achieve good memory performance with data structures allocated in the global memory space, the access pattern restrictions are much more severe. Thus, for data structures that need to be updated and which are unsuitable for the on-chip shared memory, there is a significant burden placed upon developers to meet those restrictions. The introduction of a relatively modest amount of on-chip cache for the read-write global memory space would substantially reduce the burden on developers of ensuring the regularity of a kernel's memory accesses at the expense of raising coherence issues.

**Add a Control Processor:** A more substantial architectural change would be to add to the GPU a small control processor that provides higher single-thread performance than the underlying throughput-oriented PEs. If this core were able to launch kernels, then the overhead of kernel invocation would be significantly decreased since the latency between the control processor and the parallel substrate would be much lower than the latency between the CPU and the substrate. Additionally, applications with non-trivial sequential phases could be efficiently supported in a more straightforward manner. For example, the reduced allocation implementation of the tracking stage performs one iteration of the solver on the GPU and then transfers the current state of the matrix back to the CPU to perform the reduction and check for convergence. Copying the matrix from the GPU to the CPU consumes more time than the actual kernel execution. If the serial reduction could instead be executed on the GPU's control processor, almost all of the memory transfer overhead could be avoided.

## 7. Related Work

The availability of cheap, high-performance GPUs which can be programmed using a familiar programming abstraction has led a large number of developers to port their applications to CUDA. Garland *et al.* [11] provide a good overview of the experiences and speedups achieved in a number of application domains. Many developers are working with applications that are more naturally ported to CUDA because they consist of kernels that perform huge amounts of work. These developers do not encounter many of the overheads associated with fine-grained kernels that we explore in this work. Only a few have fully explored the optimizations necessary to obtain significant speedups.

Automating the exploration of CUDA configurations in order to optimize performance was explored by Ryoo *et al.* [12]. The authors of that work do not consider mapping

major data structures to different memory spaces in CUDA or reorganizing their memory layout to achieve higher performance, and do not explore more complex optimizations such as trading off the amount of computation done on the CPU and on the GPU. The optimization strategies of multiple applications and the use of CUDA's rich memory hierarchy were explored by Che *et al.* [13]. However, they focus on applications which have a large amount of work per kernel call, and thus do not have to deal with the system bottlenecks explored in this work.

## 8. Conclusions

We have shown that leukocyte detection and tracking can benefit greatly from using a CUDA-capable GPU. The algorithms used in the detection and tracking stages, stencil computations and iterative solvers, are also used in a wide range of other application domains, which can all benefit from the optimizations we have discussed. Overall, the best CUDA implementation provides speedups of 58.5x and 211.3x on the detection and tracking stages, respectively, over the original MATLAB implementation and 9.4x and 27.5x over the best OpenMP implementation. While the MATLAB implementation takes more than four and a half hours to process one minute of video, the CUDA implementation can process that same video in less than one and a half *minutes*. Put another way, while the MATLAB implementation can detect and track leukocytes at 0.11 FPS, the CUDA implementation operates at 21.6 FPS. For video recorded at 30 FPS, continued scaling of hardware resources mean that real-time analysis is now within reach for inexpensive workstations.

While straightforward CUDA implementations can achieve substantial benefits, especially with a modest amount of tuning, significant programmer effort can be required to make full use of the GPU's potential when irregular memory access patterns or small kernels are present. Despite this extra effort required to realize the potential of the GPU, the benefits can be dramatic. Our experiences with CUDA show the power of the GPU as a parallel platform, and help demonstrate how the variety of manycore platforms that we expect to see in the future will transform computational science.

## Acknowledgments

This work was supported in part by NSF grant IIS-0612049, SRC grant 1607.001, a GRC AMD/Mahboob Khan Ph.D. fellowship, an NVIDIA research grant, and equipment donated by NVIDIA. The authors would like to thank Saurav Basu for his help with the original MATLAB implementation, as well as Leo Wolpert, Donald Carter, and Drew Gilliam for their prior work on implementing the leukocyte detection and tracking algorithm.

## References

[1] G. Dong, N. Ray, and S. T. Acton, "Intravital leukocyte detection using the gradient inverse coefficient of variation," *IEEE Transactions on Medical Imaging*, vol. 24, no. 7, pp. 910–924, July 2005.

[2] N. Ray and S. T. Acton, "Motion gradient vector flow: An external force for tracking rolling leukocytes with shape and size constrained active contours," *IEEE Transactions on Medical Imaging*, vol. 23, no. 12, pp. 1466–1478, December 2004.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[4] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[5] N. Ray, S. T. Acton, and K. Ley, "Tracking leukocytes in vivo with shape and size constrained active contours," *IEEE Transactions on Medical Imaging*, vol. 21, pp. 1222–1235, 2002.

[6] S. B. Forlow, E. J. White, S. C. Barlow, S. H. Feldman, H. Lu, G. J. Bagby, A. L. Beaudet, D. C. Bullard, and K. Ley, "Severe inflammatory defect and reduced viability in CD18 and E-selectin double-mutant mice," *Journal of Clinical Investigation*, vol. 106, pp. 1457–1466, 2000.

[7] S. T. Acton and K. Ley, "Tracking leukocytes from in vivo video microscopy using morphological anisotropic diffusion," in *IEEE International Conference on Image Processing*, 2001, pp. 300–303.

[8] U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill, "Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip," *Journal of Solid State Circuits*, vol. 43, no. 1, pp. 6–20, 2008.

[9] L. Seiler *et al.*, "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1–15, 2008.

[10] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[11] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *IEEE MICRO*, vol. 28, no. 4, pp. 13–27, 2008.

[12] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2008, pp. 195–204.

[13] 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," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.