

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

Michael Boyer, David Tarjan,
Scott T. Acton, and Kevin Skadron
University of Virginia

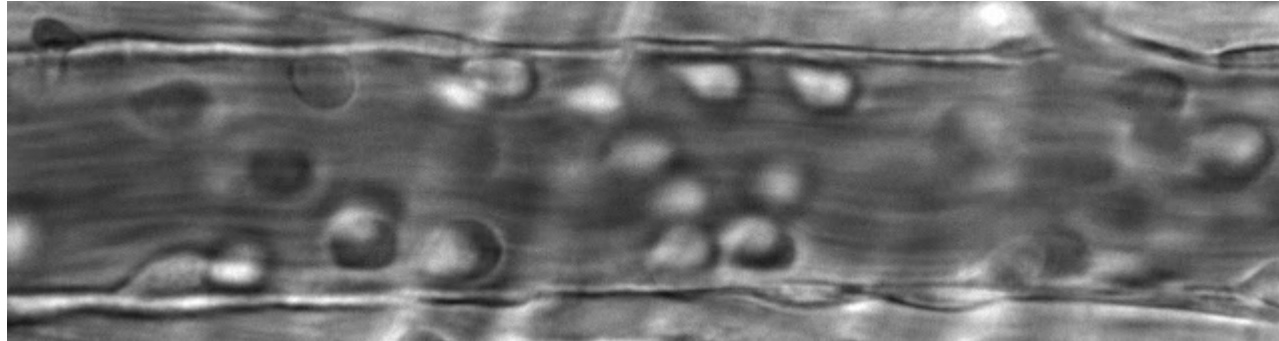
IPDPS 2009



Outline

- Leukocyte tracking:
 - Problem
 - Current approaches
- Acceleration using CUDA:
 - Bottlenecks
 - Optimization techniques
 - Performance impact

Leukocyte Tracking



- Velocity of rolling leukocytes (white blood cells) provides important information about the inflammatory response
- Velocity measured by tracking leukocytes through multiple frames

Leukocyte Tracking: Approaches

- Manual analysis
 - Researcher marks leukocyte centers frame-by-frame
 - Process 1 minute of video in tens of hours
- Automated analysis using MATLAB
 - Removes manual effort and observer bias
 - Process 1 minute of video in >4.5 hours

Goal: Leverage CUDA and a GPU to accelerate leukocyte tracking to near real-time speeds

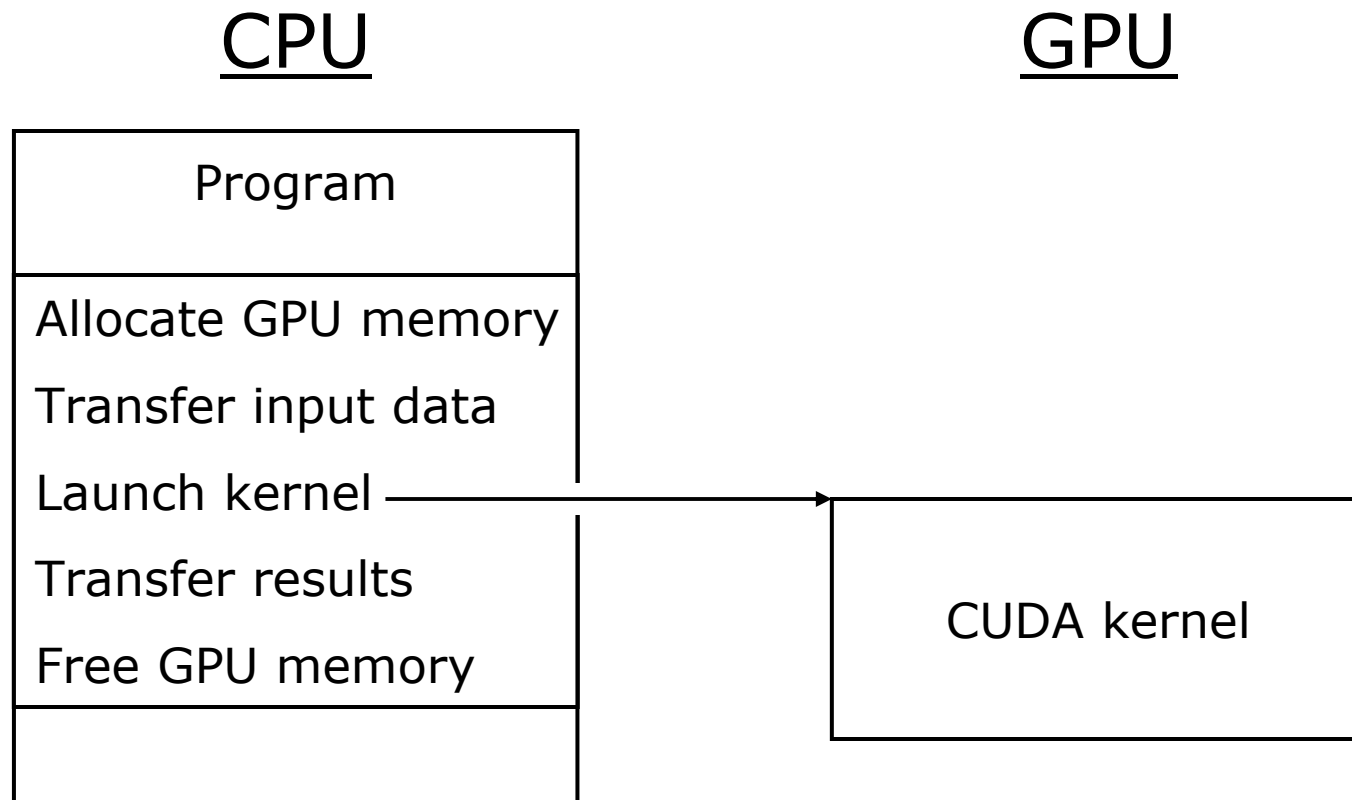
Acceleration

1. Translation: convert MATLAB code to C
2. Parallelization:
 - OpenMP for multi-core CPU
 - CUDA for GPU
- Experimental setup:
 - CPU: 3.2 GHz quad-core Intel Core 2 Extreme X9770
 - GPU: NVIDIA GeForce GTX 280 (PCIe 2.0)

CUDA

- Programming model for running general-purpose applications on NVIDIA GPUs
- Based on C, with some minor extensions
- Main CUDA abstraction: *kernel* function
 - Scalar program invoked across many threads
 - Threads grouped into thread blocks
 - Communication only allowed among threads within the same thread block

Acceleration using CUDA



Step 1: Determine which code to offload to the GPU as a CUDA kernel

Step 2: Write the CPU-side CUDA code **We focus on these two steps**

Step 3: Write and optimize the GPU kernel

Tracking Algorithm

Inputs: Video frame
Location of cells in previous frame

Output: Location of cells in current frame

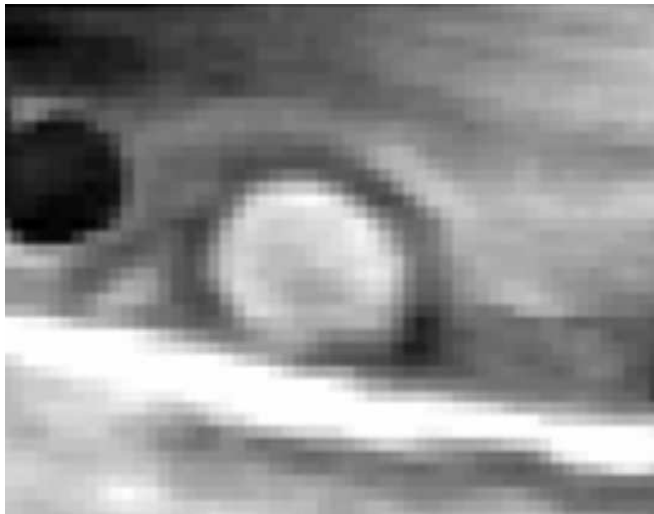
For each cell:

- Extract sub-image near cell's old location
- Compute MGVF matrix over sub-image → 99.8% of total runtime
- Evolve active contour using MGVF matrix

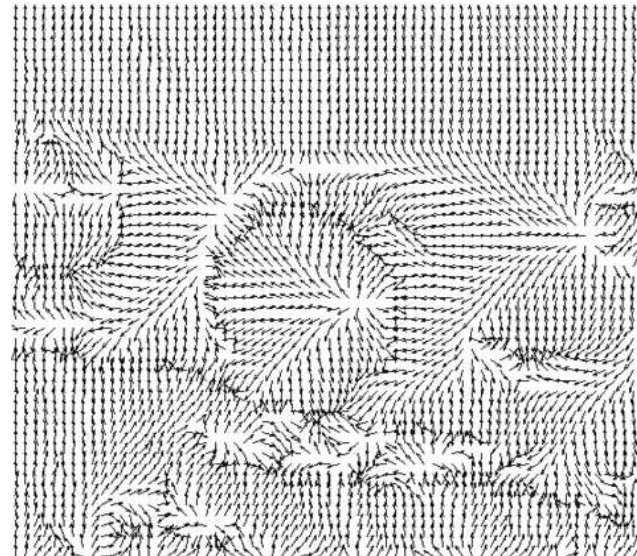
Computing the MGVF Matrix

- Motion Gradient Vector Flow
- Gradient vector field biased in the assumed direction of motion
- MGVF matrix is approximated via an iterative solution procedure

Sub-image near cell



Corresponding MGVF



MGVF Pseudo-code

MGVF = normalized sub-image gradient

do {

 Compute the difference between each element and its eight neighbors

 Compute the regularized Heaviside function across each matrix

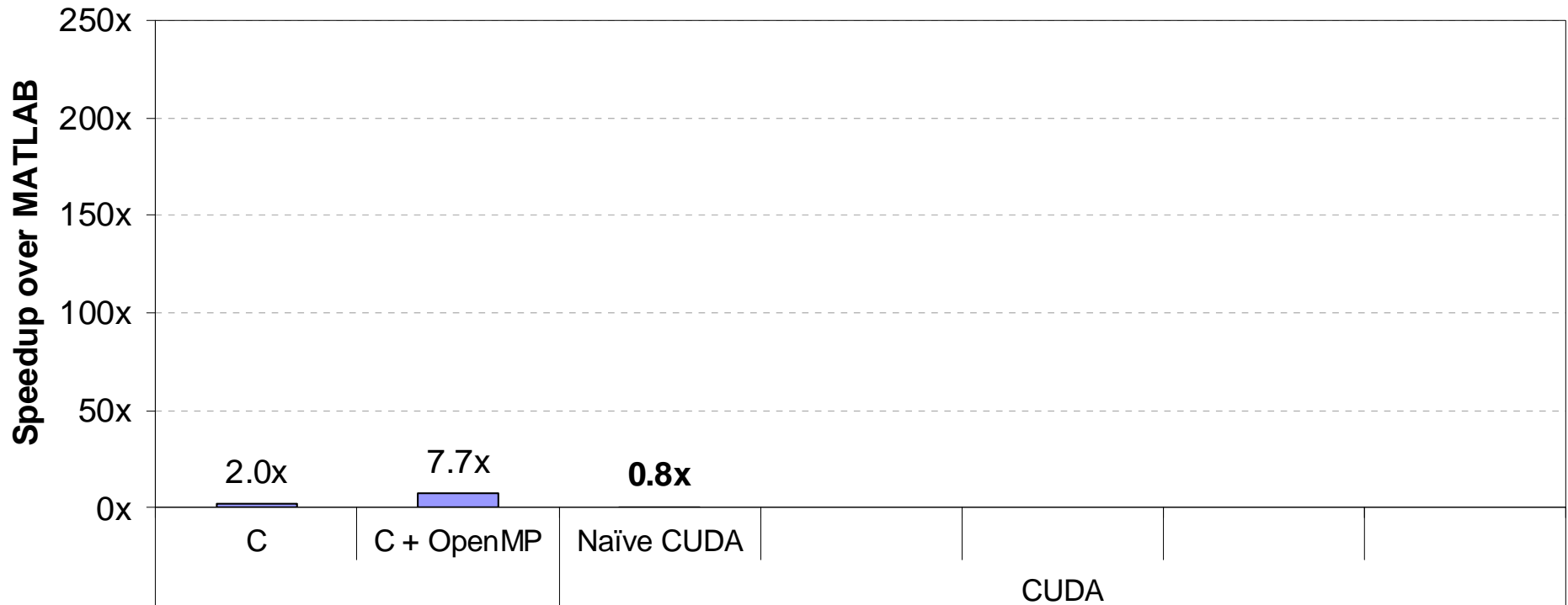
Initial
kernel
body

 Update MGVF matrix

 Compute convergence criterion

} while (not converged)

Naïve CUDA Implementation



- Kernel is called $\sim 50,000$ times per frame
- Amount of work per call is small
- Runtime dominated by CUDA overheads:
 - Memory allocation
 - Memory copying
 - Kernel call overhead

Kernel Overhead

- Kernel calls are not cheap!
 - Overhead of one kernel call: 9 μ s
 - Overhead of one CPU function: 3 ns
- Heaviside kernel:
 - 27% of kernel runtime due to computation
 - 73% of kernel runtime due to kernel overhead

Lesson 1: Reduce Kernel Overhead

- Increase amount of work per kernel call
 - Decrease total number of kernel calls
 - Amortize overhead of each kernel call across more computation

Larger Kernel Implementation

MGVF = normalized sub-image gradient

do {

Compute the difference between each pixel and its eight neighbors

Compute the regularized Heaviside function across each matrix

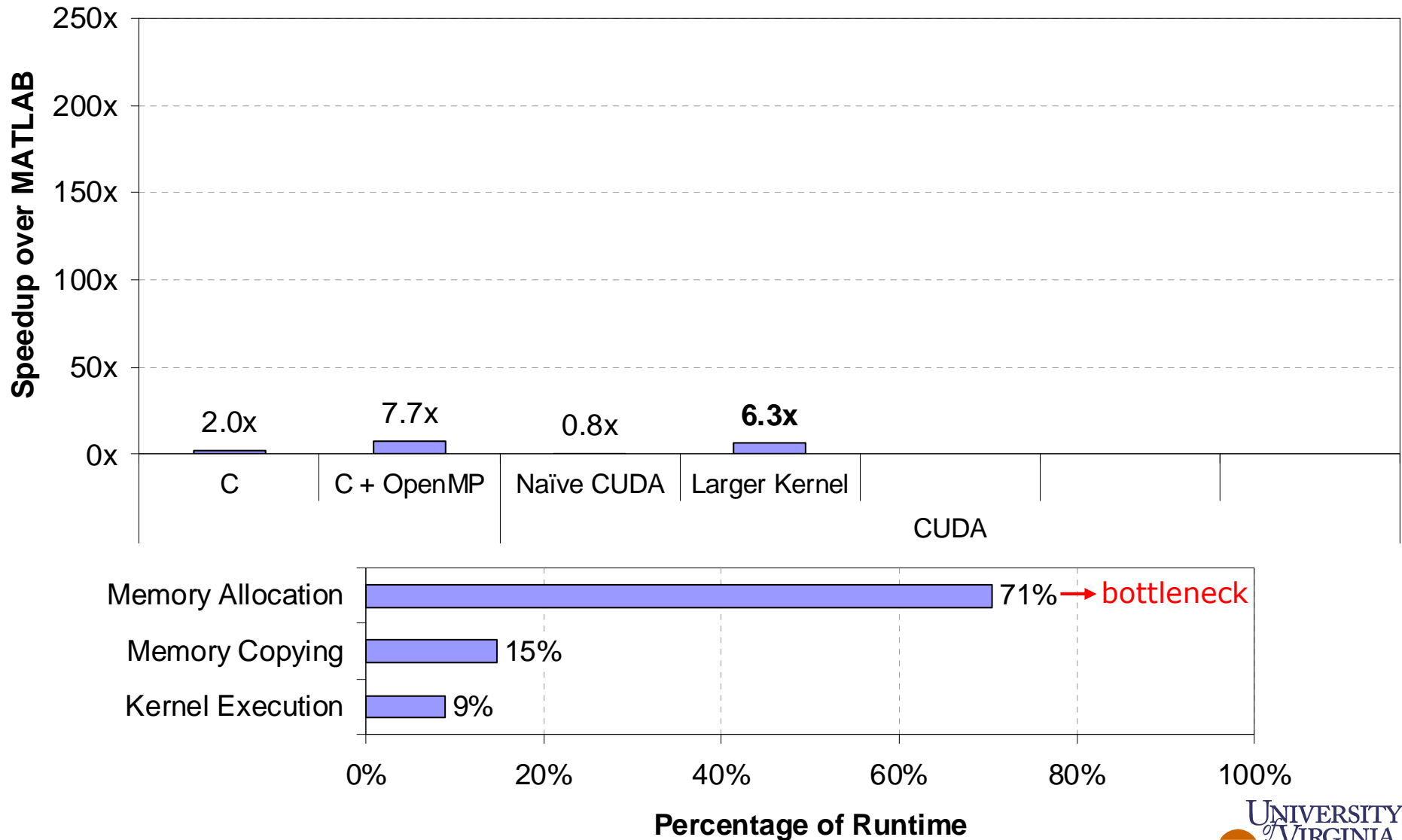
Update MGVF matrix

Compute convergence criterion

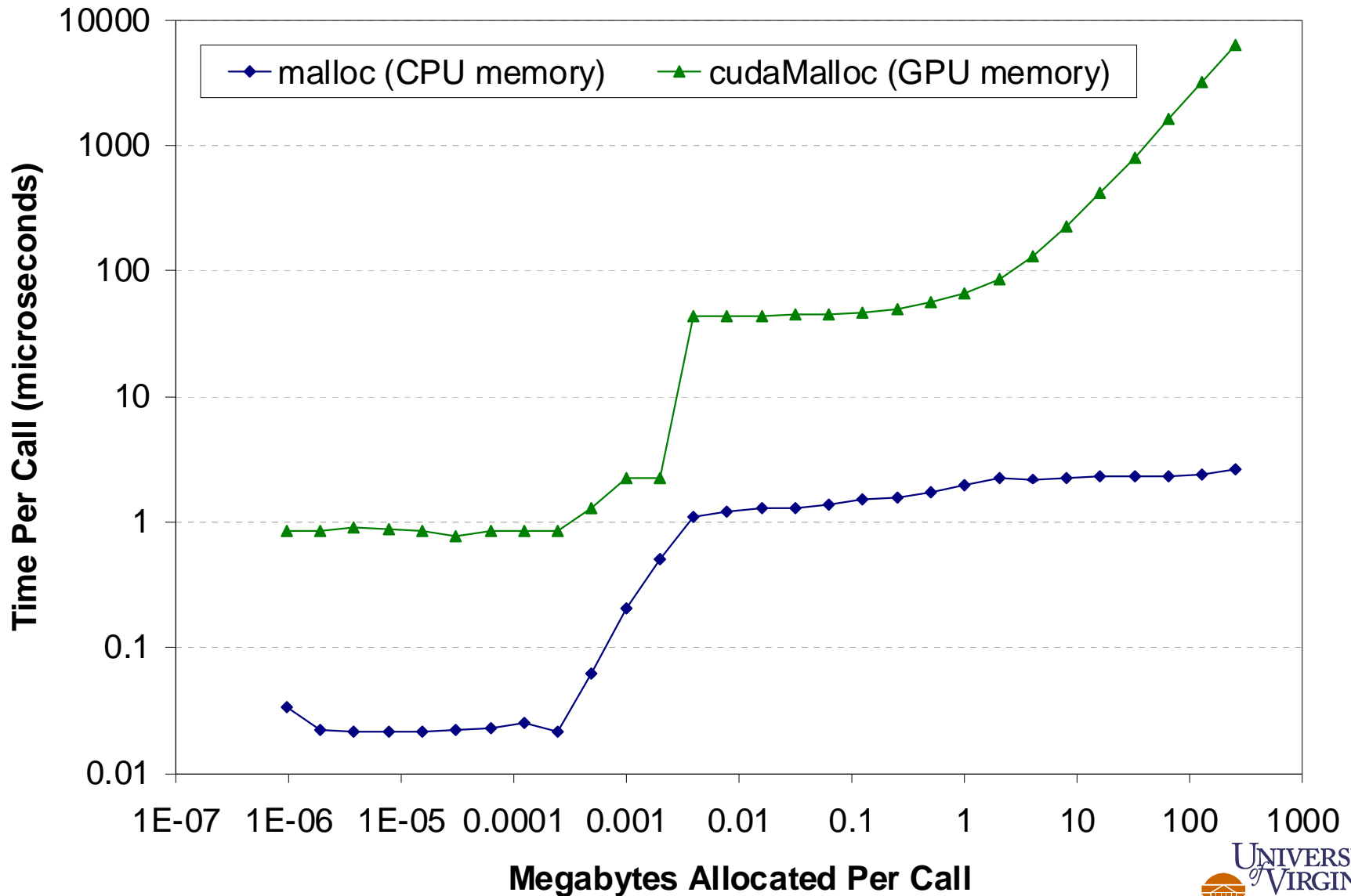
} while (! converged)

Expand
kernel
body

Larger Kernel Implementation



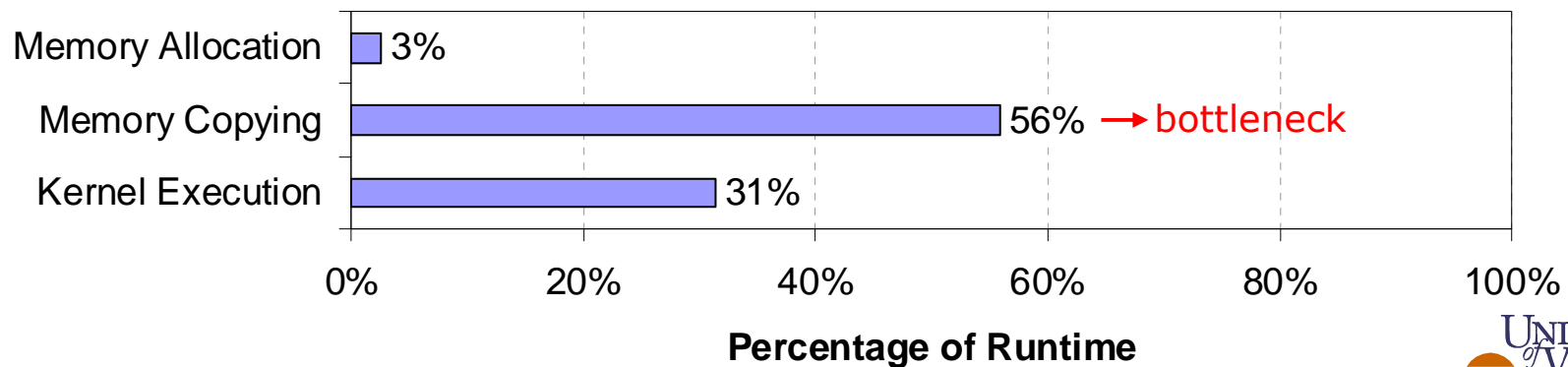
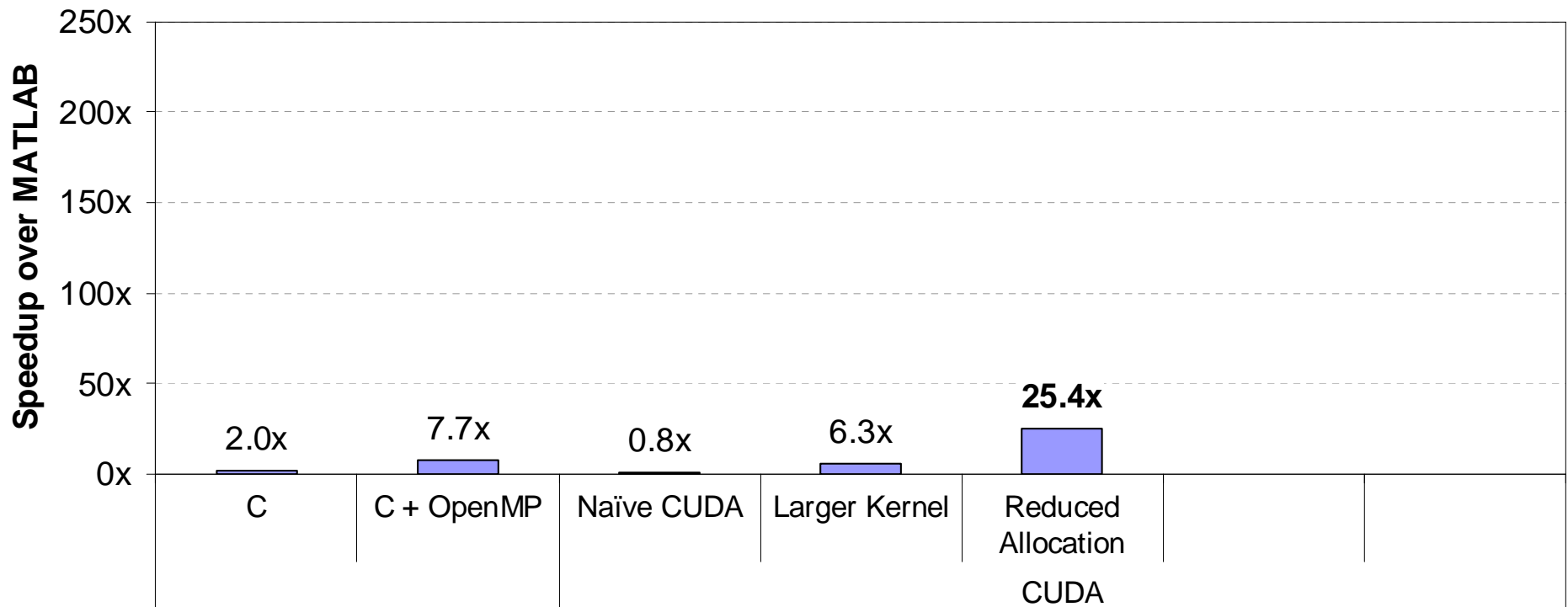
Memory Allocation Overhead



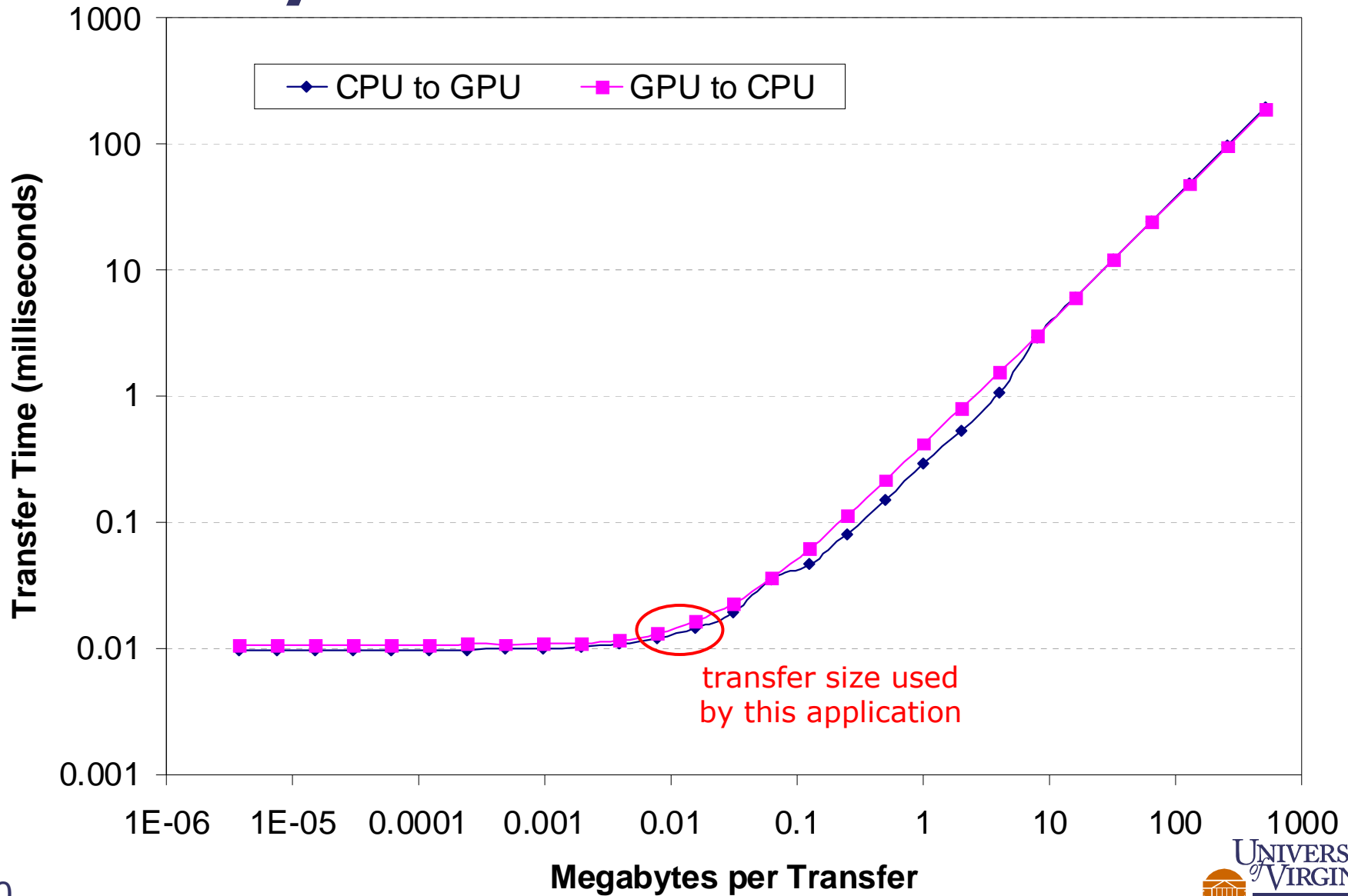
Lesson 2: Reduce Memory Management Overhead

- Reduce the number of memory allocations
 - Allocate memory once and reuse it throughout the application
 - If memory size is not known a priori, estimate and only re-allocate if estimate is too small

Reduced Allocation Implementation

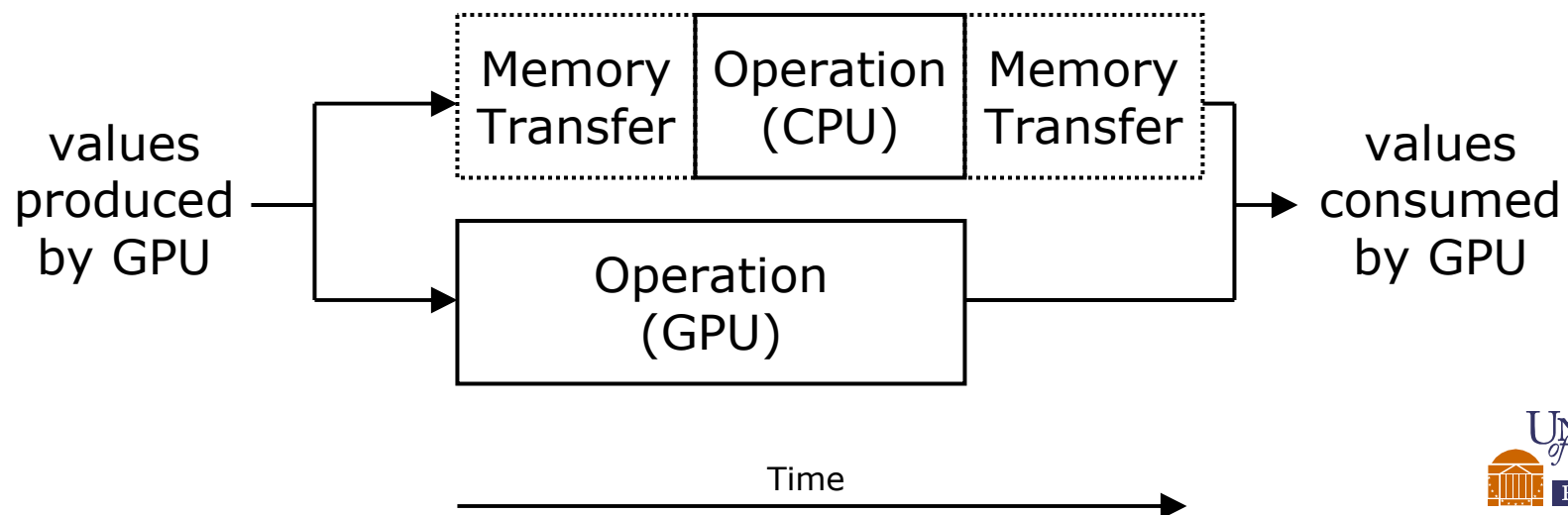


Memory Transfer Overhead



Lesson 3: Reduce Memory Transfer Overhead

- If the CPU operates on values produced by the GPU:
 - Move the operation to the GPU
 - May improve performance even if the operation itself is slower on the GPU



GPU Reduction Implementation

MGVF = normalized sub-image gradient

do {

Compute the difference between each pixel and its eight neighbors

Compute the regularized Heaviside function across each matrix

Update MGVF matrix

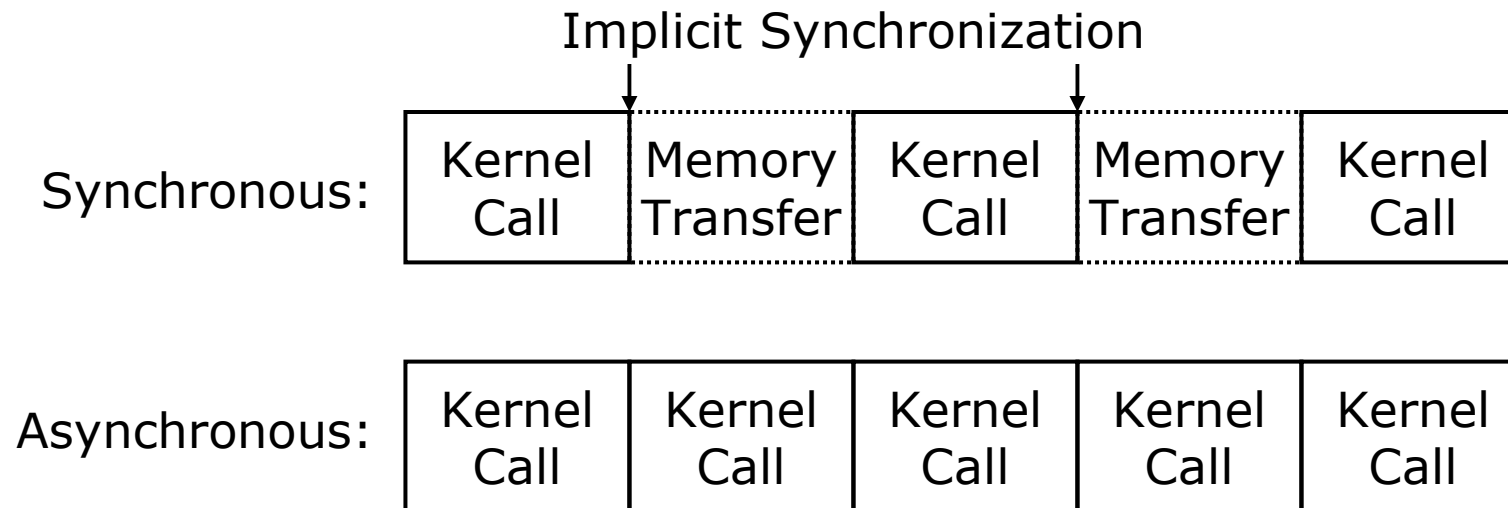
Compute convergence criterion

} while (! converged)

Add
conver-
gence
check
to
kernel
body

Kernel Overhead Revisited

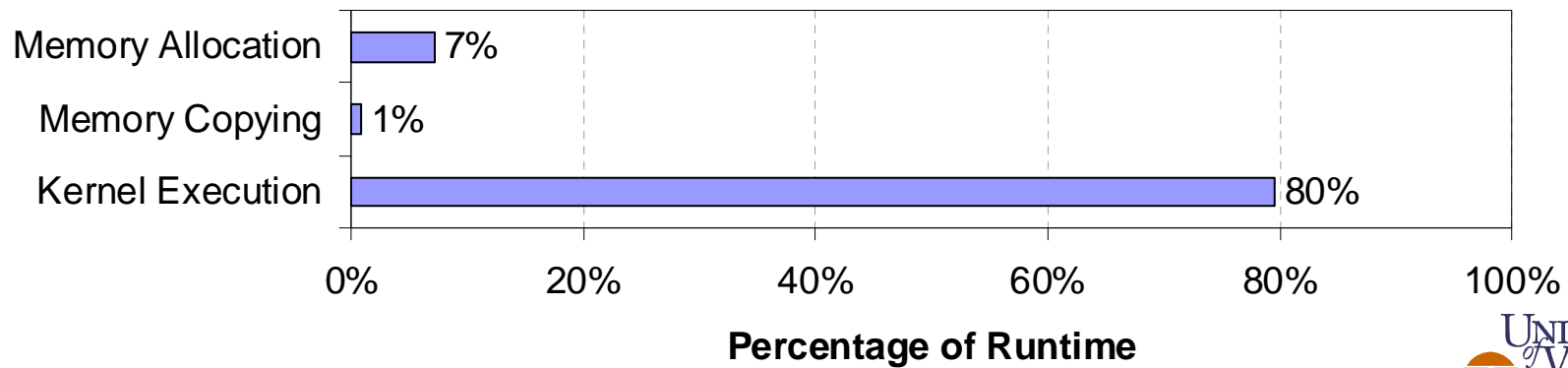
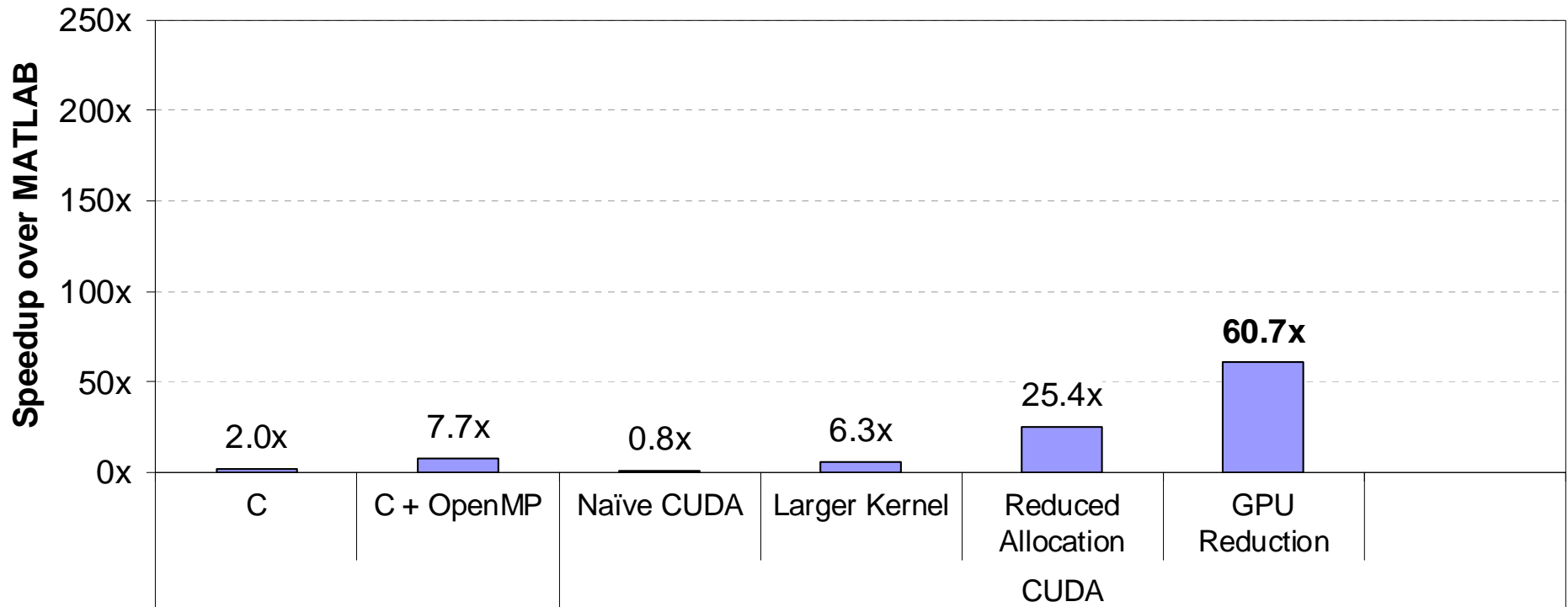
- Overhead depends on calling pattern:
 - One at a time (synchronous): 9 μ s
 - Back-to-back (asynchronous): 3 μ s



Lesson 1 Revisited: Reduce Kernel Overhead

- Increase amount of work per kernel call
 - Decrease total number of kernel calls
 - Amortize overhead of each kernel call across more computation
- Launch kernels back-to-back
 - Kernel calls are asynchronous: avoid explicit or implicit synchronization between kernel calls
 - Overlap kernel execution on the GPU with driver access on the CPU

GPU Reduction Implementation



Persistent Thread Block

MGVF = normalized sub-image gradient

do {

 Compute the difference between each pixel and its eight neighbors

 Compute the regularized Heaviside function across each matrix

 Update MGVF matrix

 Compute convergence criterion

} while (! converged)

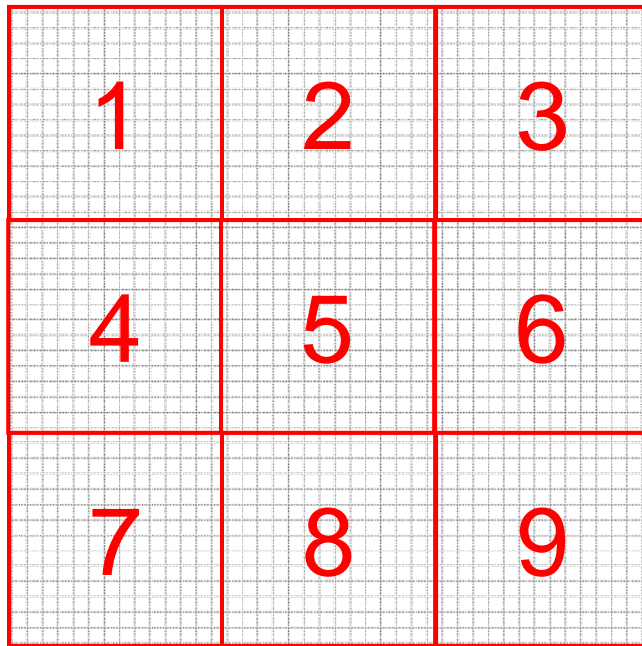
How
can we
offload
the
entire
while
loop
as a
kernel?

Persistent Thread Block

- Problem: need a global memory fence
 - Multiple thread blocks compute the MGVF matrix
 - Thread blocks cannot communicate with each other
 - So each iteration requires a separate kernel call
- Solution: compute entire matrix in one thread block
 - Arbitrary number of iterations can be computed in a single kernel call

Persistent Thread Block: Example

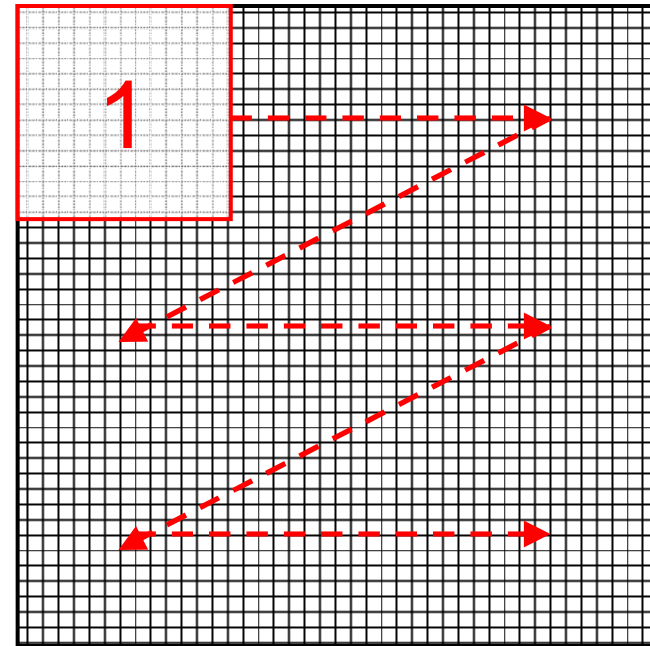
MGVF Matrix



Canonical CUDA Approach

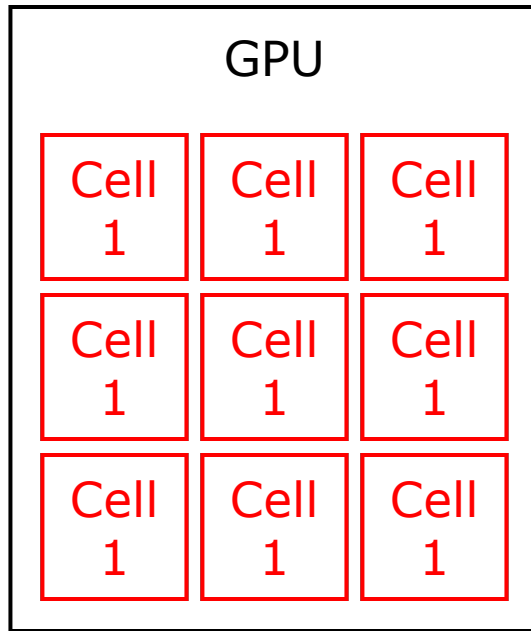
(1-to-1 mapping between threads and data elements)

MGVF Matrix



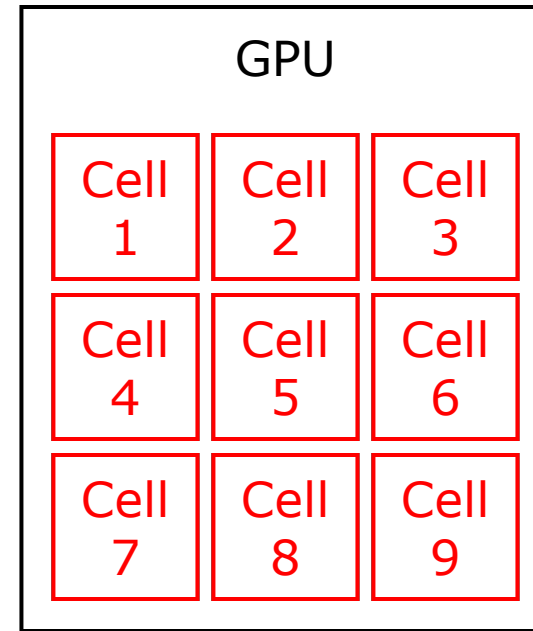
Persistent Thread Block

Persistent Thread Block: Example



Canonical CUDA Approach

(1-to-1 mapping between threads and data elements)



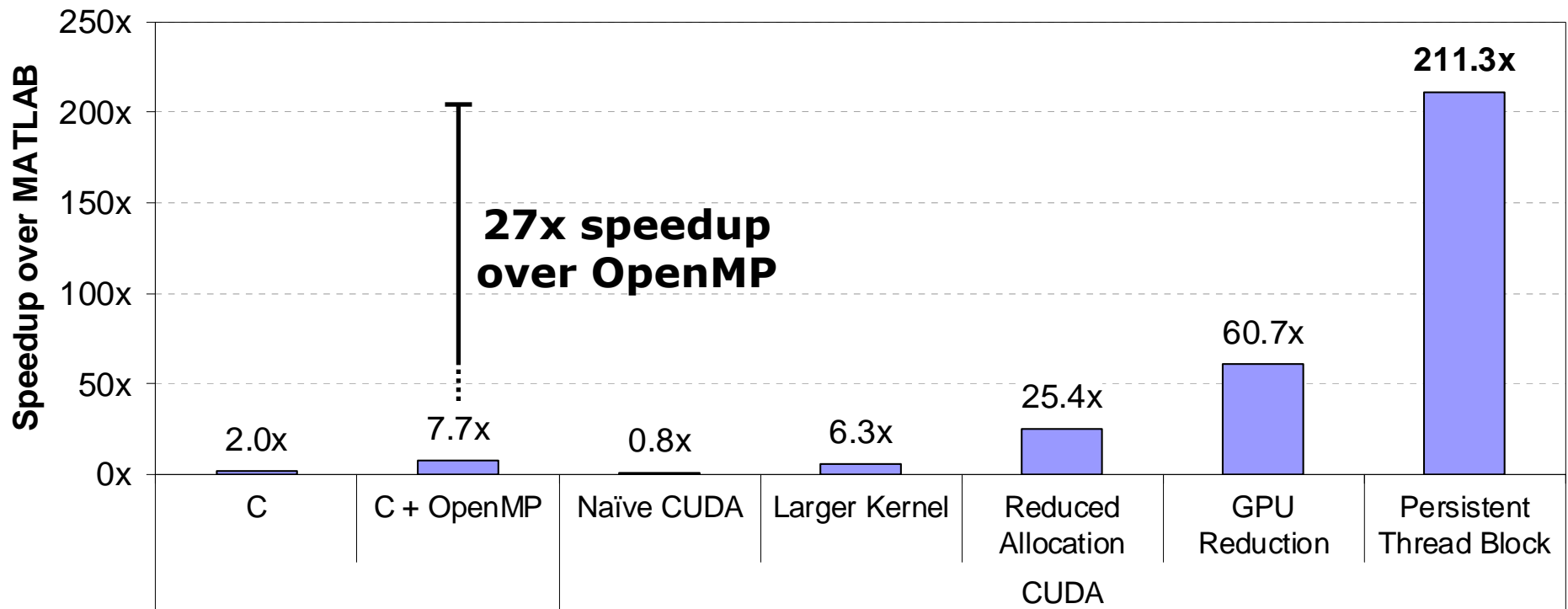
Persistent Thread Block

Lesson 4:

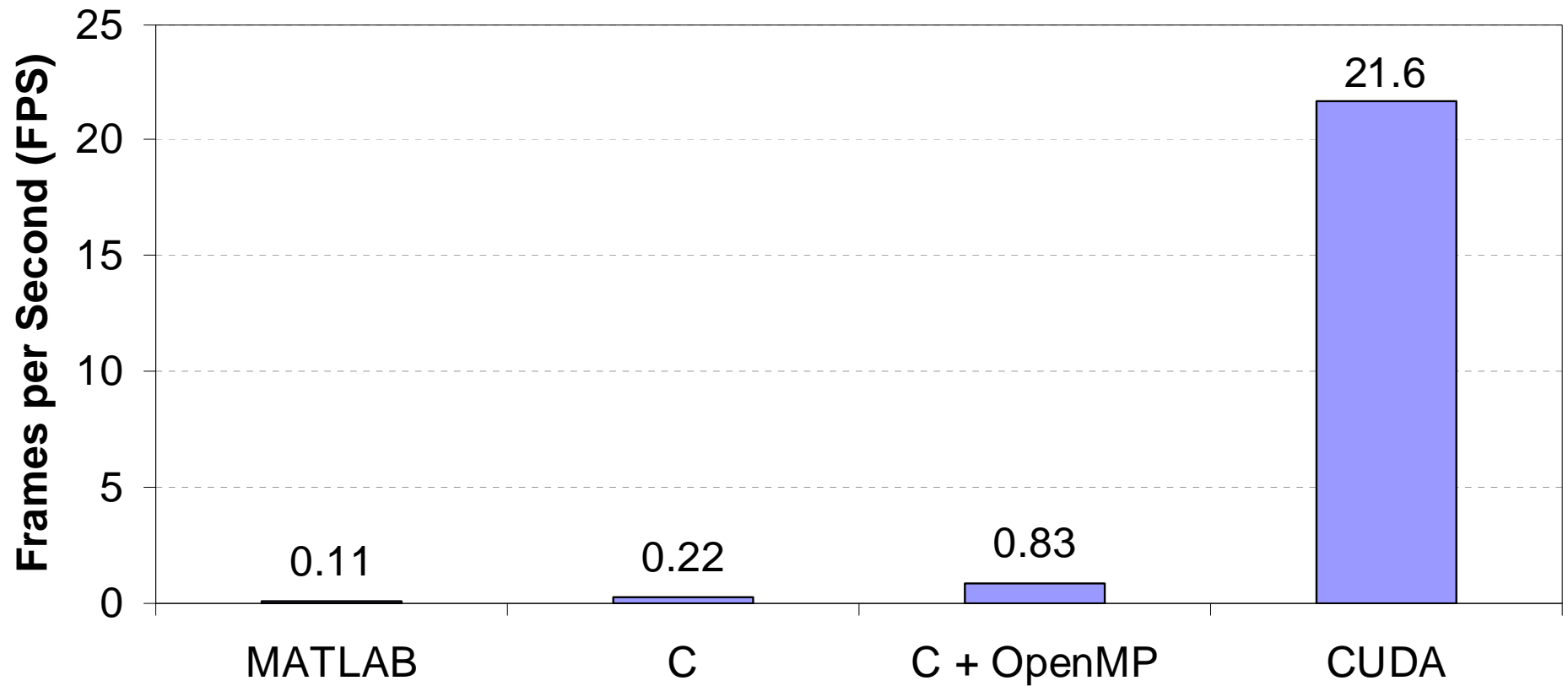
Avoid Global Memory Fences

- Confine dependent computations to a single thread block
 - Execute an iterative algorithm until convergence in a single kernel call
 - Only efficient if there are multiple independent computations

Persistent Thread Block Implementation



Absolute Performance



Conclusions

- CUDA overheads can be significant bottlenecks
- Techniques presented here can help mitigate the impact of these bottlenecks
- CUDA provides enormous performance improvements for leukocyte tracking
 - 200x speedup over MATLAB
 - 27x speedup over OpenMP
- Processing time for a 1 minute video reduced from >4.5 hours to <1.5 minutes
- Real-time leukocyte tracking will be feasible in the near future

Acknowledgements

- Funding provided by:
 - NSF grant IIS-0612049
 - SRC grant 1607.001
 - NVIDIA research grant
 - GRC AMD/Mahboob Kahn Ph.D. fellowship
- Equipment donated by NVIDIA

Software

Source code available at:

<http://lava.cs.virginia.edu/wiki/rodinia>

ImageJ plugin will be available soon