CATEGORY: **COMPUTATIONAL PHOTOGRAPHY**

POSTER **CP01** | CONTACT NAME Walid Keyrouz: walid.keyrouz@nist.gov

GPU TECHNOLOGY CONFERENCE

A Hybrid CPU-GPU Approach to Fourier-Based Image Stitching of Optical Microscopy Images

Walid Keyrouz, Timothy Blattner, Bertrand Stivalet, Joe Chalfoun, and Mary Brady

Software and Systems Division, Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899–8970
Contact: walid.keyrouz@nist.gov

Shujia Zhou

Department of Computer Science
University of Maryland Baltimore County
100 Hilltop Circle
Baltimore, MD 21250

Abstract: We present a hybrid CPU-GPU approach for the Fourier-based stitching of optical microscopy images. This system achieves sub-minute stitching rates with large grids; it stitches a grid of 59x42 tiles in 26 seconds on a two-CPU (8 physical cores) & two-GPU machine. This is a speedup factor of more than 24x; the optimized sequential implementation takes more than 10 minutes to perform the same task. The system scales to take advantage of additional CPU cores or GPU cards. For the sake of comparison, ImageJ/Fiji, which uses a similar algorithm, exceeds 3.6 hours on the same workload.

Project

Goal: Use computing to enable and accelerate biological measurements

Objectives:
- Image stitching of optical microscopy images at interactive rates
- General purpose library

Motivation:
- Scientists are now frequently acquiring tiled images
- Stitched images essential for measurements based on acquired images

Contributions

Separate execution pipeline per GPU
- Scalable across multiple GPUs

Host-side pipelined threading organization
- Overlaps compute and data transfers
- Uses all available system resources

Memory Management
- Avoids virtual and physical memory limitations (CPU & GPU)
- Copy data to the GPU only once

Evaluation Platform

Hardware
- Dual Intel® Xeon® E-5620 CPUs
  Quad-core, 2.4 GHz, hyper-threading
- 48 GB RAM
- Dual NVIDIA® Tesla® C2070 cards

Reference Implementation
- ImageJ/Fiji™ Stitching plugin, >3.6 hours
- Goal: < 1 minutes

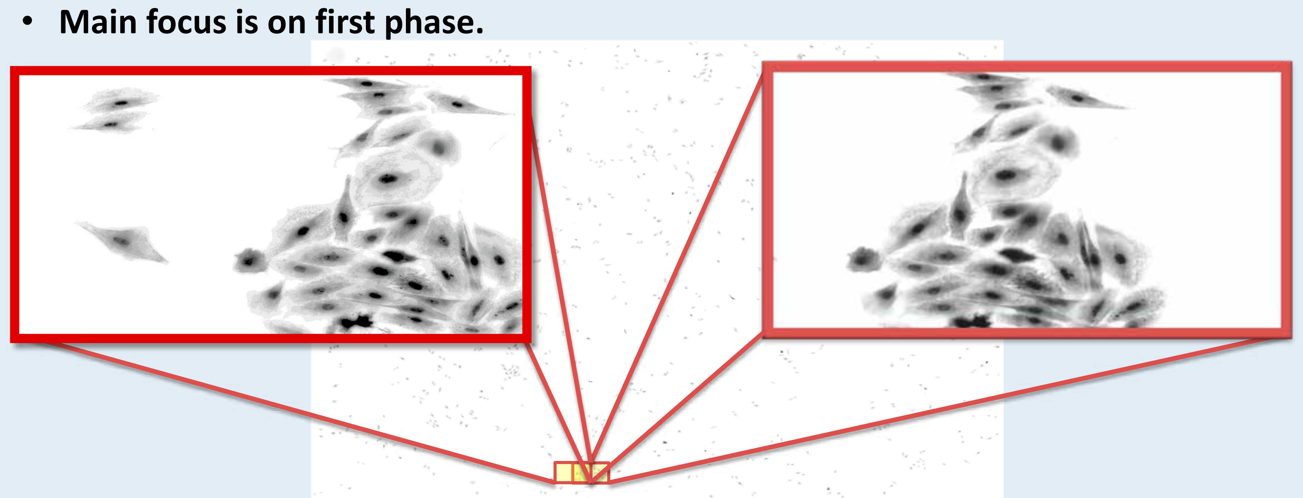
Software
- Ubuntu 12.04/x64, kernel 3.2.0
- Libc6 2.15, libstd++6 4.6
- BOOST 1.48, FFTW 3.3, libTIFF4
- NVIDIA CUDA & CUFFT 5.0

Data Set:
- Grid of 59x42 images (2478)
- 1040x1392 16-bit grayscale images (2.8 MB per image)
- Total: ~ 6.7 GB

Image Stitching Problem

Three phases of image stitching:
1. Compute the X & Y translations for all image tiles
2. Eliminate over-constraint through global optimization
3. Apply the computed translations & compose into one image

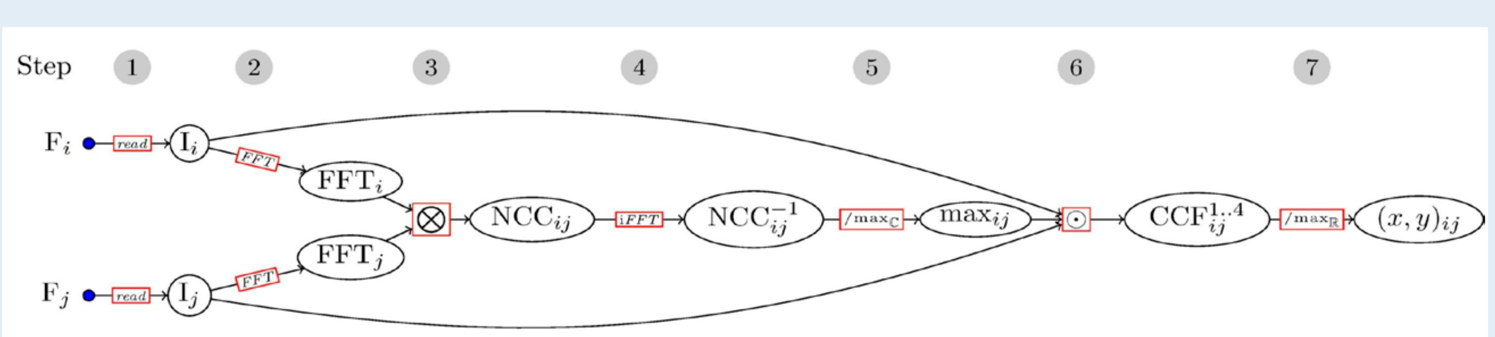
- **Main focus is on first phase.**



Algorithm

Loop over all images:
1. Read an image
2. Compute its Forward 2D Fourier Transform (FFT-2D)
3. Compute correlation coefficients with west and north neighbors
   - Depends on FFT-2D
   - NCC^{-1} & index of max give (x,y) disp.

Major compute portion:
- FFT-2D of each tile (FFT)
- Normalize phase correlation (NCC)
- Inverse FFT-2D (NCC^{-1})
- Max Reduction (max)
- 4 Correlation Coefficients ($CCF^{1\text{-}4}$)



Fourier Transforms

FFTW (fftw.org)

Auto-tuning FFT-2D plan
- First creates plan to compute FFT based on CPU properties and FFT dimensions
- Planning mode specifies effort to find "best" FFT algorithm

Amortized planning cost
- Save plan to use later
- Run prior to stitching computation

FFTW Planning Mode	Planning Time	Execution Time
Estimate	0.02 s	137.7 ms
Measure	4 min 23 s	66.1 ms
Patient	4 min 23 s	66.1 ms
Exhaustive	7 min 1 s	66.1 ms

Implementations

Reference: Sequential Implementation	
Simple Multi-Threaded	Simple GPU
Pipelined Multi-Threaded	Pipelined GPU

Results

	Time	Speedup	Threads	GPUs
Sequential*	10 min 37 s	-	1	-
Simple Multi-Threaded*	1 min 35 s	6.7x	16	-
Pipelined Multi-Threaded*	1 min 22 s	7.7x	19	-
Simple GPU	9 min 47 s	1.08x	1	1
Pipelined GPU Single GPU	43.6 s	14.6x	11	1
Pipelined GPU Dual GPU	26 s	24.5x	15	2

* FFT computations on the CPU use FFTW *exhaustive* planning

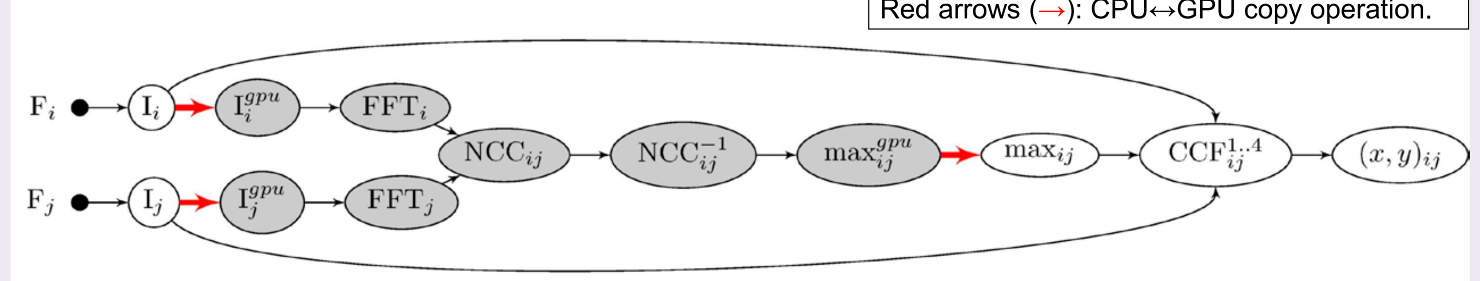
Sequential Implementations: CPU & GPU

CPU
- All computation in double precision; complex to complex 2-D Fourier transforms
- Explicitly manages memory for image transforms to avoid memory limits
- Functions hardcoded with SSE intrinsics:
  - Normalized Cross Correlation factors (step 3)
  - Max reduction (step 5)
- > 80% of computation in forward and backward Fourier transforms

GPU
- Direct port to GPU of sequential CPU implementation
- Offloads most computational tasks to GPU
  - CUFFT function calls for forward & backward transforms
  - Custom CUDA kernels for NCC & Max Reduction computations
    - NCC uses shared memory, one thread per data element
    - Max reduction extracts the max's index
      - Modified version of NVIDIA's SDK reduction sum example
- Copies image data to GPU memory, copies one scalar per image pair back to CPU memory
- CCF computed on CPU
  - Enables minimal data transfer back to CPU memory; computation not very expensive

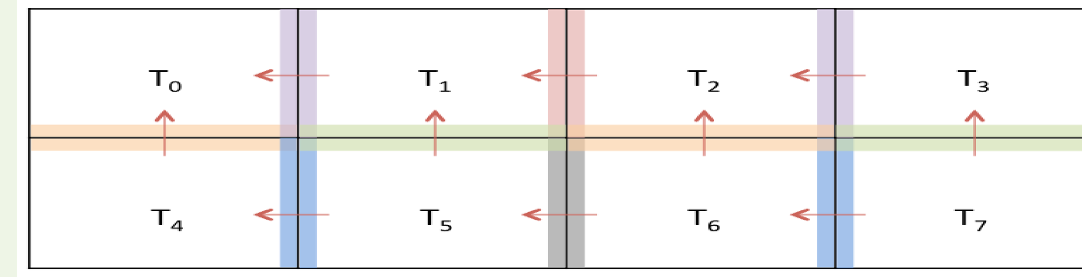
Operation	w/o Intrinsics	SSE Intrinsics		GPU	
	Avg. Time	Avg. Time	Speedup	Avg. Time	Speedup
NCC	55.8 ms	21.2 ms	2.63x	9 ms	6.2x
Max Reduction	27 ms	5.9 ms	4.57x	4.9 ms	5.51x

Quantities computed on GPU appear in gray.
Red arrows (→): CPU↔GPU copy operation.



Simple Multi-Threaded Implementation

- Spatial domain decomposition, one thread per partition
- Explicit handling of inter-partition dependencies (red arrows in figure below)



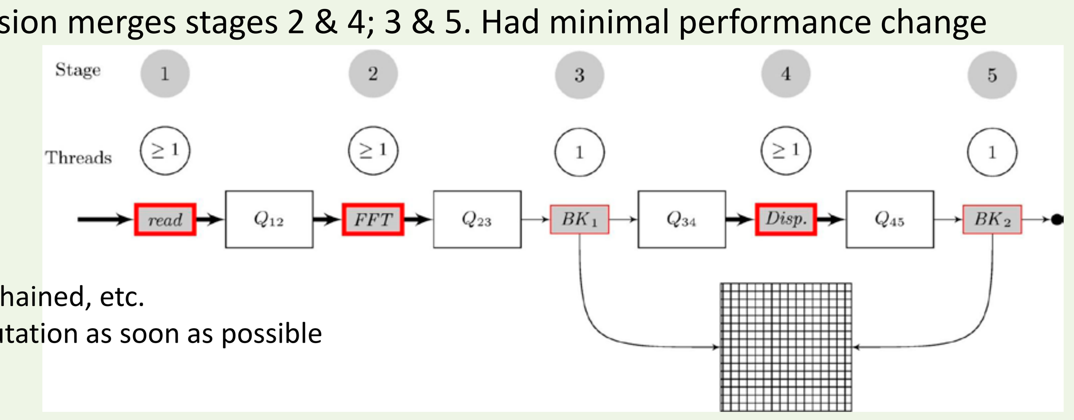
8-way spatial domain decomposition of grid. Shaded areas and arrows indicate dependencies

Three phases for all threads separated by barriers:
1. Compute FFT of own images
   Compute relative displacements of tiles with no inter-partition dependencies
   Release memory of transforms w/o dependents
   Barrier
2. Compute relative displacements for *remaining* tiles (on partition boundaries)
   Barrier
3. Release memory of remaining transforms

Pipelined Multi-Threaded Implementation

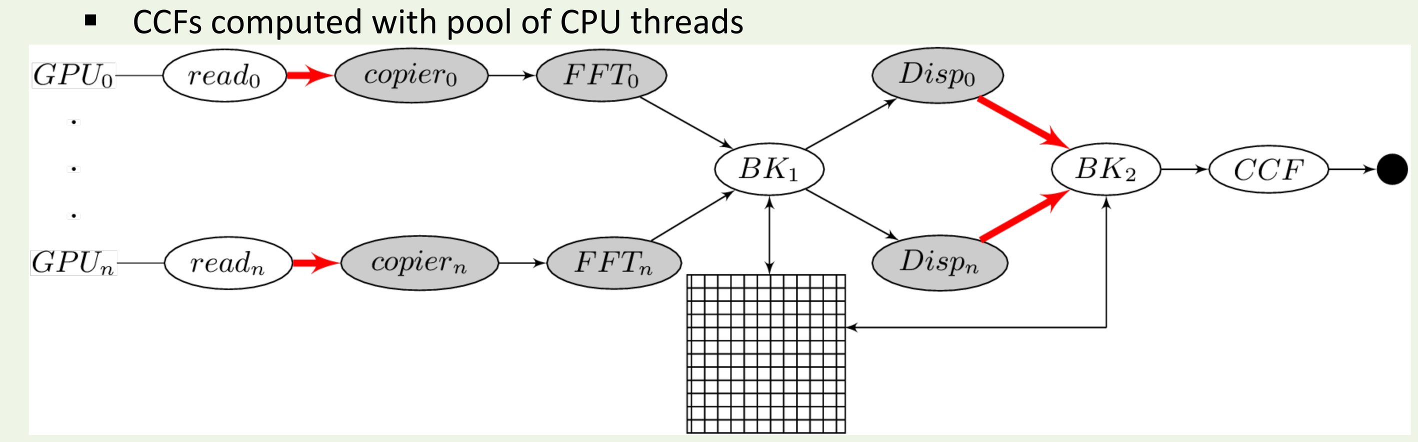
5-stage pipeline
- Stages communicate via queues with synchronization using mutexes
- # threads ≥ 1 for stages 1, 2, & 4; 1 book-keeping thread for each of stages 3 & 5
- 3-stage pipeline version merges stages 2 & 4; 3 & 5. Had minimal performance change

Grid traversals
- Diagonal, diagonal chained, etc.
- Enables disp. computation as soon as possible



Pipelined GPU Implementation

- Adapts pipelined multi-threaded implementation to GPU
  - 1 execution pipeline per GPU
  - Distribute image grid evenly between GPUs
- Seven stage pipeline
  - 1 queue and CPU thread per stage per GPU for read, copy, FFT, and Disp.
    - Overlaps copies and compute on GPUs
  - BK_1 gathers FFTs and distributes pairs of FFTs to GPUs using tile grid
  - BK_2 frees GPU memory using tile grid
    - BK_1 & BK_2 use a spinlock on tile grid to prevent race conditions
  - CCFs computed with pool of CPU threads



Quantities computed on GPU appear in gray.
Red arrows (→): CPU↔GPU copy operation.

Remarks

- No benefit from direct port of sequential version to GPU
- Simple multi-threaded suffers from load imbalance
  - Load imbalance handled by pipelined implementation
- FFTW & CUFFT sensitive to vector sizes (should be powers of 2, 3, or 5)

Future Work
- Explore padding & single precision complex FFTs
- Use different data sets & compare with other benchmarks
- Experiment with alternative GPU architectures & accelerator cards