A Hybrid CPU-GPU System for Stitching of Large Scale Optical Microscopy Images

Timothy Blattner
SSD, ITL
National Institute of
Standards & Technology
Gaithersburg, MD 20899-8970
Email: timothy.blattner@nist.gov

Walid Keyrouz*
SSD, ITL
National Institute of
Standards & Technology
Gaithersburg, MD 20899-8970
Email: walid.keyrouz@nist.gov
*Corresponding author

Joe Chalfoun
SSD, ITL
National Institute of
Standards & Technology
Gaithersburg, MD 20899-8970
Email: joe.chalfoun@nist.gov

Bertrand Stivalet
SSD, ITL
National Institute of
Standards & Technology
Gaithersburg, MD 20899-8970
Email: bertrand.stivalet@nist.gov

Mary Brady
SSD, ITL
National Institute of
Standards & Technology
Gaithersburg, MD 20899-8970
Email: mary.brady@nist.gov

Shujia Zhou
Computer Science
University of Maryland Baltimore County
1000 Hilltop Circle
Baltimore, MD 21250
Email: szhou@umbc.edu

Abstract— **Researchers in various fields are using optical microscopy to acquire very large images, 10 000–200 000 of pixels per side. Optical microscopes acquire these images as grids of overlapping partial images (thousands of pixels per side) that are then stitched together via software. Composing such large images is a compute and data intensive task even for modern machines. Researchers compound this difficulty further by obtaining time-series, volumetric, or multiple channel images with the resulting data sets now having or approaching terabyte sizes.**

We present a scalable hybrid CPU-GPU implementation of image stitching that processes large image sets at near interactive rates. Our implementation scales well with both image sizes and the number of CPU and GPU cores in a machine. It processes a grid of 42×59 tiles into a $17\text{k} \times 22\text{k}$ pixels image in $43\,\text{s}$ (end-to-end execution times) when using one NVIDIA Tesla card and two Intel Xeon E-5620 quad-core CPUs, and in $29\,\text{s}$ when using two Tesla C2070 cards and the same two CPUs. It also composes and renders the composite image without saving it in $15\,\text{s}$. In comparison, ImageJ/Fiji takes $> 3.6\,\text{h}$ for the same workload despite being multithreaded and executing the same mathematical operators; it composes and saves the large image in $1.5\,\text{h}$.

This implementation takes advantage of coarse-grain parallelism. It organizes the computation into a pipeline architecture that spans CPU and GPU resources and overlaps computation with data motion. The implementation achieves a nearly 10x performance improvement over our optimized non-pipeline GPU implementation and demonstrates near-linear speedup when increasing CPU thread count and increasing number of GPUs.

Keywords*—*Hybrid systems, Parallel Architectures, Heterogeneous (hybrid) systems, Scheduling and task partitioning

I. Introduction

Image Stitching comes up in Optical Microscopy because of a scale mismatch between the dimensions of a plate being examined and the microscope's Field Of View (FOV). For example, the region of interest in a plate is measured in *cm* (e.g., $2 \times 2\,\text{cm}^2$) whereas the FOV for a 10x objective is one order of magnitude smaller per side ($< 1 \times 1\,\text{mm}^2$).

To image a plate, a microscope scans the plate as it travels under the optical column by distances smaller than the FOV and generates overlapping partial images or tiles; a software tool then assembles the grid of tiles into a plate image. This reconstruction requires computing the (x, y) translations between adjacent images because the image overlap distances vary during an experiment from their preset values due to the stage's mechanical properties, actuator backlashes, and camera angle.

Long-running experiments impose an additional time-based requirement on image stitching as plates are imaged periodically. Image stitching must reconstruct a plate image in a fraction of the imaging period to allow researchers enough time to examine and analyze the acquired images and, if need be, intervene in these long-running experiments. Such a capability is essential to transform experiments from being simply automated to ones that are computationally steerable.

As an example, biologists at the National Institute of Standards and Technology (NIST) are using automated optical microscopes to study cell colony behavior over 5 days. In these experiments, the plate is $2 \times 2\,\text{cm}^2$ and is scanned every $45\,\text{min}$ to produce two tile grids, one per color channel; each grid consists of up to $10\,000$ tiles depending on the microscope's overlap and magnification settings. A particular experiment, which produced one of the smaller dataset, scanned a plate 161 times, produced 2 grids of 18×22 tiles per scan, with each tile having 1392×1040 16-bit grayscale pixels. The resulting raw image data had a size of $344\,\text{GB}$ ($161 \times 2 \times 18 \times 22 \times 1392 \times 1040 \times 2$ bytes).

Achieving scalable image stitching is the main motivation of this work. Realizing this goal faces two major challenges. The first challenge relates to computational time. Scanning a plate takes between 15 and $45\,\text{min}$ depending on a microscope's overlap and magnification settings. Stitching should reconstruct a plate image in a fraction of the time needed to scan a plate in order to give researchers enough time to (1) analyze the resulting image and derive measurements from it via an image segmentation tool for example—yet another potentially time-consuming task—and (2) decide if there is a need to intervene in the running experiment.

The second challenge is algorithmic. Optical microscopy can generate images with few distinguishable features within the overlap region that can be used to guide stitching. This occurs often in the early part of live cell experiments when cell colonies are seeded at low densities and the colonies have not expanded to cover most of the plate. This lack of distinguishable features makes it difficult to determine the relative positioning of two adjacent image tiles and rules out a large class of stitching algorithms with good performance characteristics.

To quantify our approach's results, we will measure its performance when processing an imaging dataset of A10 cell colonies acquired by NIST biologists using an Olympus IX71 microscope with a 10x lens and an infrared camera. The images form a grid of 42×59 tiles. Each tile is a 1040×1392 16-bit grayscale image; they are each $2.76\,\text{MB}$ in size and cover an area of $896.44\,\mu\text{m} \times 669.76\,\mu\text{m}$. The size of the dataset is $6.68\,\text{GB}$. Our evaluation machine has two Intel Xeon E-5620 CPUs (quad-core with hyper-threading), $48\,\text{GB}$ of RAM, and two NVIDIA Tesla C2070 GPU cards. We will also compare our execution times and results with the ImageJ/Fiji stitching plug-in [1]–[3]; this comparison is critical given the widespread adoption of ImageJ/Fiji and its plugins within the bio-imaging informatics community. The stitching plug-in took more than $3.6\,\text{h}$ to compute displacements for the plate image and an additional $1.5\,\text{h}$ to compose and save the stitched image which has $14\,579 \times 12\,290$ pixels.

A. Approach and Contributions

Our work uses a Fourier-based approach to image stitching which we describe in Section II. We present and compare several implementations and detail a hybrid CPU-GPU implementation which achieves end-to-end processing times of $49.7\,\text{s}$ for a 42×59 grid of tiles on a machine with one high-end GPU card. Such execution times transform image stitching into a quasi-interactive task and are more than two orders of magnitude better than those of ImageJ/Fiji which takes nearly $3.6\,\text{h}$ for the same workload. Furthermore, these execution times compare favorably with other published timing results for similar problems using GPUs [4]. Lastly, these implementations lay the foundation to developing a software stack that supports computationally steerable experiments centered around optical microscopy.[1]

The hybrid CPU-GPU implementation takes advantage of coarse grain parallelism and organizes the computation as a pipeline of functional stages (reading, computing, and bookkeeping). Each stage consists of one or more CPU threads, some of which interact with GPUs. The pipeline overlaps various computations that take place on CPU or GPU cores with data transfers between disk, main memory, and graphics memory. This pipeline implementation provides a near 11.2x performance improvement over a basic approach to GPU-based acceleration and can be used to address other problems where coarse-grained parallelism is available.

B. Organization

The rest of the paper is organized as follows: section II discusses alternative approaches underlying image stitching algorithms; section III describes in detail the Fourier-based image stitching algorithm used in this study; sections IV and V present the implementations that were developed and discuss their performance; section VI then concludes and outlines possible extensions.

II. Image Stitching Algorithms & Related Work

Szeliski discusses many algorithms for finding the proper alignment of image tiles [5]. These algorithms fall into two categories: feature-based alignment techniques [6], [7] and direct methods [8], [9].

In our study, we use a direct method, a version of Kuglin and Hines' phase correlation image alignment method [8] that is modified to use normalized correlation coefficients as described by Lewis [10], [11]. This method uses Fast Fourier Transforms (FFTs) to compute Fourier Correlation Coefficients and then uses these correlation coefficients to determine image displacements. Algorithms 2 & 3 give pseudo-code listings of the correlation functions. In our context, the Fourier-based approach is advantageous because it has predictable parallelism and is more robust for optical microscopy; it does not depend on explicitly detecting features in images whose presence can be sparse in live cell microscopy images.

The literature reports on feature-based approaches that work for some microscopy images. For example, the AutoStitch software [12] implements Brown and Lowe's scale-invariant feature transform [13]. Ma et al. report successfully using AutoStitch to process microscopy images [7]. However, they used images that are feature-rich.

Cooper, Huang, and Ujaldon use a feature-based algorithm in an implementation aimed at clusters with CPU and GPU compute nodes [14]. Their implementation uses a combination of FFT-based normalized cross-correlation and feature detection. They report performance numbers using a data set with feature rich images that are much larger than the ones we use (16k $\times$ 16k and 23k $\times$ 62k pixels). Their study demonstrates the feasibility of using a hybrid CPU-GPU implementation running on a cluster to improve image-stitching performance. However, the improvement is less than can be achieved by the hardware because their implementation was not designed to hide data transfer latencies.

Preibisch, Saalfeld, and Tomanak describe an ImageJ/Fiji plugin for image stitching [15]. This plugin uses a direct approach to compute image displacements and is multi-threaded to improve performance. Our implementation uses the same approach for computing relative displacements. However, it is

[1]The implementations are currently available upon request from the authors; they will be posted on the web along with a reference dataset in the near future.

focused on handling the scale issue and on taking advantage of accelerator technologies to improve performance.

III. Computation

Image stitching operates in three phases. The first phase computes relative displacements for adjacent image pairs. These displacements form an over-constrained system that one can represent as a directed graph where vertices are images and edges relate adjacent images. The over-constraint in the system is due to the equivalence between absolute displacements of images and path summations in the graph which must be path invariant. The second phase resolves the over-constraint in the system and computes absolute displacements. It selects a subset of the relative displacements or uses a global optimization approach to adjust them to a path invariant state in the graph. The third phase uses the absolute displacements to compose the stitched image.

This work focuses on the first phase of the algorithm, namely the *relative displacements computation* phase, as it is the more compute-intensive phase. The second phase is much lighter computationally while the third phase can be carried out on demand as part of visualizing the stitched image.

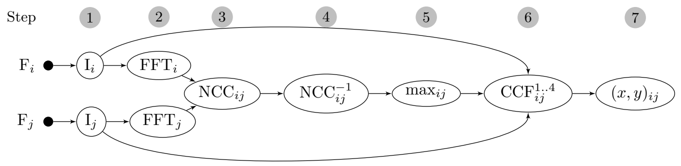

Fig. 1. Computation for Two Adjacent Images

Fig. 1 shows the data flow graph for computing the relative displacement between two adjacent images, i and j (east-west or north-south). The steps are outlined below and use the same step numbers as in the data flow graph.

1) Read image files, F_i and F_j, into objects I_i and I_j.
2) Compute the 2D Fourier transforms, FFT_i and FFT_j.
3) Compute the image pair's Normalized Correlation Coefficient (NCC_{ij}). This is the element-wise normalized conjugate multiplication of two complex vectors.
4) Compute the 2D inverse Fourier transform of the normalized correlation coefficient (NCC_{ij}^{-1}).
5) Reduce the inverse transform to its maximum ($\max_{ij}$) and map its index back to image coordinates (x, y). Fourier transforms are periodic in nature. As such, the overlap distances, x and y, are ambiguous and can be interpreted as x or $(w - x)$ and y or $(h - y)$.
6) Compute the Cross-Correlation Factors ($\text{CCF}_{ij}^{1..4}$). Each cross-correlation factor corresponds to one of the four overlap modes, (x or $w - x$), (y or $h - y$), (x or y), and ($w - x$ or $h - y$).
7) Find $\text{CCF}_{ij}^{\max}$ and map it back to $(x, y)_{ij}$.

Fig. 2 lists the pseudo-code corresponding to steps 2–7; the pseudo-code for function CCF appears in Fig. 3. Fig. 4 shows the computation for the whole grid; it repeats the pair-wise computation for all adjacent image pairs.

The image stitching algorithm is compute-bound and is dominated by Fourier transform computations. Table I shows the count and complexity of operations as well as the sizes of

```
// Function pciam($I_i$, $I_j$)
Input: two image arrays—same size!
Output: tuple—max correlation, $x$-disp & $y$-disp
begin
    $\text{FFT}_i \longleftarrow$ FFT_2d($I_i$)             // Forward FFTs
    $\text{FFT}_j \longleftarrow$ FFT_2d($I_j$)
                 // Normalized Correlation Coeff.
    fc $\longleftarrow \text{FFT}_i\ .\times \overline{\text{FFT}_j}$              // elt wise op
    $\text{NCC}_{ij} \longleftarrow$ fc ./ abs($fc$)           // normalize
                               // max in Inverse FFT
    $\text{NCC}_{ij}^{-1} \longleftarrow$ iFFT_2d($\text{NCC}_{ij}$)
    [max, y, x]$_{ij} \longleftarrow$ max(abs($\text{NCC}_{ij}^{-1}$))
     // Consider four combinations
    c1 $\longleftarrow$ ccf($I_i$[y:H, x:W], $I_j$[0:H-y, 0:W-x])
    c2 $\longleftarrow$ ccf($I_i$[y:H, W-x:W], $I_j$[0:H-y, 0:x])
    c3 $\longleftarrow$ ccf($I_i$[H-y:H, x:W], $I_j$[0:y, 0:W-x])
    c4 $\longleftarrow$ ccf($I_i$[H-y:H, W-x:W], $I_j$[0:y, 0:x])
    return max([c1, x, y], [c2, W-x, y], [c3, x,
    H-y], [c4, W-x, H-y])
end
```

Fig. 2. Algorithm: Relative Displacement of Adjacent Images

```
// Function ccf($I_1$, $I_2$)
Input: two image arrays—same size!
Output: cross correlation factor (double)
begin
    $I_1 \longleftarrow I_1 -$ mean($I_1$)    // Center both vectors
    $I_2 \longleftarrow I_2 -$ mean($I_2$)
    $N \longleftarrow I_1.I_2$                        // dot product
    $D \longleftarrow |I_1|.|I_2|$                // product of norms
    return $N/D$
end
```

Fig. 3. Algorithm: Fourier Cross Correlation

the operands in these operations; in the table, n and m denote the grid size while h and w give the size of the partial images. Processing an $n \times m$ grid performs $(3nm - n - m)$ forward and backward 2-D Fourier transforms on double complex numbers. The cost of each transform is $O(hw \log(hw))$ when h and w have a special form, a power of small prime numbers (e.g., 2, 3, 5, & 7) or a product of such powers, and the FFT library uses a divide and conquer approach to take full advantage of the recursive formulation of FFT. For optical microscopy, there is no guarantee that the partial images will have such *nice* dimensions and the cost of these transforms may be substantially higher. The image stitching computation also includes a large number of vector multiplications and reductions; these operations can become comparatively expensive unless they are implemented using hardware vector instructions.

For the class of problems under consideration (grids with thousands of tiles), the relative displacement computation exhibits a high degree of coarse-grain parallelism: computing the forward transforms of all images (FFTs), computing the normalized correlation coefficients of all adjacent image pairs (NCCs), computing the inverse transforms of all NCCs, etc. However, these are not embarrassingly parallel because of data dependencies and memory size limits.

```
Input: Grid of image tiles
Output: 2 arrays of tuples ([correlation, $x$, $y$])
begin
    foreach $I \in$ Grid of Tiles do
        translations-west[$I$] $\leftarrow$ pciam($I$, $I$#west)
        translations-north[$I$] $\leftarrow$ pciam($I$#north, $I$)
    end
end
```

Fig. 4. Algorithm: Grid Relative Displacements

Operation	Operation Count	Operation Cost	Operand Size (B)
Read	$n \times m$	$h \times w$	$2hw$
FFT-2D	$n \times m$	$hw \log(hw)$	$16hw$
$\otimes$	$2nm - n - m$	$h \times w$	$16hw$
FFT-2D^{-1}	$2nm - n - m$	$hw \log(hw)$	$16hw$
/max	$2nm - n - m$	$h \times w$	$16hw$
CCF$^{1..4}$	$2nm - n - m$	$h \times w$	$4hw$

TABLE I. Operation Counts & Complexities

There are two sets of computed entities with multiple dependencies, NCC$_{ij}$ and CCF$^{1..4}_{ij}$. A parallel implementation must explicitly handle these data dependencies across CPU and GPU threads.

A scalable parallel implementation must manage memory because the problem does not fit into main memory, let alone GPU memory. Each transform takes up nearly 22 MB in RAM. This results in a total of 53.5 GB just for the forward transforms of the grid. Such a size is well beyond the capacity of most machines. It will have a highly negative effect on performance when the program's working set exceeds physical memory limits and the virtual memory subsystem starts paging to disk. This constraint is substantially more severe with GPUs where even high end GPUs are often limited to 6 GB.

Fig. 5 illustrates this point further. It plots the speedup of a simple multi-threaded application on the same evaluation machine but with 24 GB of RAM only; this application reads tiles and computes their transforms without releasing any memory. The figure clearly shows the speedup falling off a cliff, across all thread counts, when the tile count changes from 832 to 864; a similar plot of execution times exhibits the same behavior.

The challenges in developing parallel implementations lie in exploiting the available coarse grain parallelism by scheduling computations on the available computing resources (CPU and GPU cores) as early as possible without violating any of the data dependency constraints and memory size limits.

IV. Implementations

This section describes the reference sequential implementations that we have developed as well as the parallel ones that take advantage of the available hardware.

Our evaluation machine was described in Section I. It has the following hardware specifications: two Intel Xeon E-5620 CPUs, 46 GB of RAM, and two NVIDIA Tesla C2070 cards with 6 GB of GDDR5 memory. The machine is running Ubuntu Linux 12.04/x86_64, kernel v. 3.2.0, and Libc6 v. 2.15. The rest of the software stack consists of libstd++6 v. 4.6, BOOST v. 1.48 [16], GCC v 4.6.3 with `-O3` optimization, and NVIDIA CUDA and cuFFT v. 5.5.

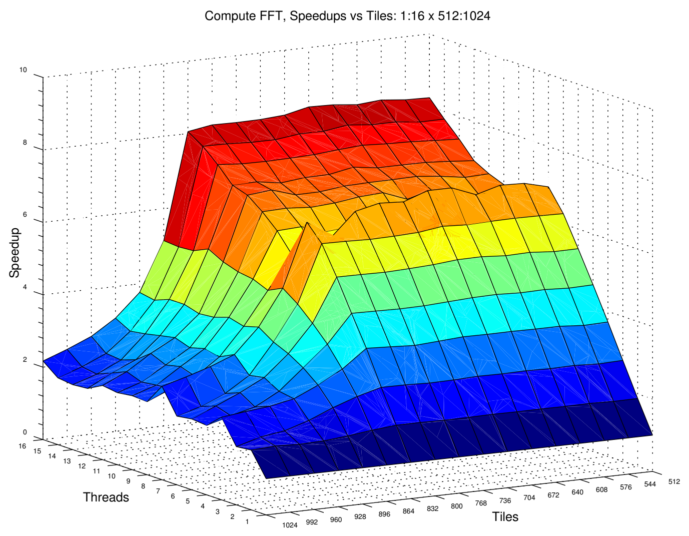

Fig. 5. Virtual Memory Performance Cliff

A. Reference Implementations

We developed two reference implementations: (1) a sequential CPU-only version and (2) a simple GPU version that is almost a direct port of the sequential CPU version. We label these two implementations, *Simple-CPU* and *Simple-GPU*.

The reference CPU-only sequential implementation reads images using libTIFF4 v. 3.9.5 [17] and uses FFTW3 v. 3.3 [18], [19] to compute Fourier transforms. We explicitly coded the functions for the element-wise vector multiplication and the max reduction with SSE intrinsics because the compiler being used (GCC v. 4.6.3 with `-O3` optimization and `-msse2`) was not generating such code.

FFTW is an auto-tuning library; it operates in two modes, *planning* and *execution*. It first generates a plan, based on problem and machine characteristics, and then executes it. FFTW planning can be expensive. However, this cost is amortized by saving a plan and reusing it. We experimented with and decided to use FFTW's *patient* planning mode as it demonstrated similar performance to the *measure* and *exhaustive* planning modes, while only taking 4 min 20 s to generate (*measure* and *exhaustive* planning took 4 min 20 s and 7 min 1 s respectively). Using *patient* planning mode yielded a 2x improvement in computing FFTs compared to *estimate* planning mode for 1392×1040 image sizes.

This implementation used a strategy of freeing memory as early as possible: freeing an image's transform memory as soon as the relative displacements of its eastern, southern, western, and northern neighbors were computed. For this purpose, this implementation supported multiple traversal orders of the grid (row, column, diagonal, and their chained counterparts). The chained-diagonal traversal order gave the best performance because it allowed memory to be freed earlier than the other

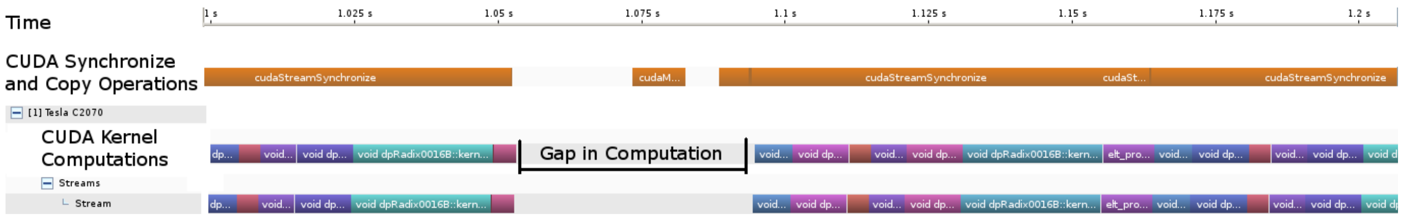

Fig. 7. Profile of a $0.2\,\mathrm{s}$ Interval of *Simple-GPU* Execution (8×8 grid)

traversal orders. Consequently, the chained-diagonal traversal order became the default.

The *Simple-CPU* implementation computes the relative displacements for the 42×59 grid on the evaluation machine in $10.6\,\mathrm{min}$ with $80\,\%$ of this time spent on Fourier transforms.

We used the *Simple-CPU* implementation to develop a simple multi-threaded implementation *MT CPU*. This implementation uses spatial domain decomposition and a thread-variant of the SPMD (Single Program Multiple Data) approach to handle coarse-grained parallelism. The best execution time was $96\,\mathrm{s}$ with 16 threads, which is a 6.6x speedup over the *Simple-CPU* implementation.

The first GPU implementation, *Simple-GPU*, is almost a direct port of the CPU sequential version. It maintains the sequential architecture of the *Simple-CPU* implementation, but invokes operations on the GPU which takes advantage of the GPU's massively parallel architecture. Fig. 6 illustrates the data flow underlying this implementation; in this figure, entities that reside in GPU memory are shaded in gray. FFT operations are done using NVIDIA's cuFFT library v 5.5 [20]. The NCC_{ij} and $\max_{ij}$ computations are performed by custom CUDA kernels that implements common CUDA optimizations (decomposing operations to maximize active blocks, shared memory, etc.).

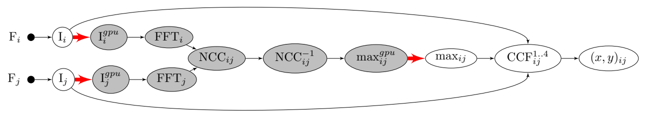

Fig. 6. Data Flow in Sequential GPU Implementation

The reference GPU implementation is single threaded on the CPU, executes CUDA memory copies synchronously, and invokes all kernels on the default stream. This implementation stitches the 42×59 grid in $9\,\mathrm{min}\,16\,\mathrm{s}$, a mere 1.14x speedup over the reference CPU-only sequential implementation. This implementation was not expected to be much faster than its CPU counterpart. Nevertheless, it included several performance improvement measures:

- It uses the NVIDIA cuFFT library to compute FFTs. The tiles in our data set are 1392×1040; these sizes are not powers of small primes and, as such, do not play well with the divide-and-conquer approach implemented by most FFT libraries. Nevertheless, a CPU-GPU comparison reveals that cuFFT is ≈ 1.5 times faster than FFTW running in patient planning mode.
- It computes forward transforms once per tile and keeps them in GPU memory ready to be reused. It frees a transform's GPU memory when the NCCs of a tile's four neighbors have been computed.
- It uses a custom CUDA kernel to compute the normalized cross correlations; this kernel uses shared memory and maximizes the graphics card occupancy. It is ≈ 2.3x faster than the corresponding CPU function.
- It uses a custom CUDA kernel to perform the max reduction and determine its index. This kernel borrows optimization ideas from Mark Harris's optimized reduction kernel [21] and is ≈ 1.5 times faster than the corresponding CPU function which uses SSE intrinsics.
- It allocates a pool of buffers in GPU memory for FFT transforms and keeps track of the buffers to help manage the limited memory available on the GPU.
- It minimizes transfers from device to host memory by only copying the result of the parallel reduction.

To better understand why the *Simple-GPU* implementation did not yield better performance, we used NVIDIA's visual profiler [22]. Fig. 7 shows a $0.2\,\mathrm{s}$ interval of a stitching computation sample run (8×8 grid) as visualized by the profiler (out of a $15.9\,\mathrm{s}$ total). The profile illustrates that there is only one kernel executing on the GPU at a time; this correlates with the single CUDA stream. It also reveals gaps between kernel invocations; these gaps account for CCF computations on the CPU and memory copies between the CPU and GPU.

The major issues with the *Simple-GPU* implementation are synchronously invoking kernels, waiting for CPU reads and CCF computations, and copying between CPU and GPU memories. Each of these uses valuable cycles and keeps the GPU unoccupied. To overcome these problems, we decided to restructure the code with the goals of (1) overlapping data transfers with GPU computations and (2) overlapping CPU tasks such as reading and computing the CCFs.

B. *Pipelined GPU Implementation*

The pipelined GPU implementation, *Pipelined-GPU*, organizes the image stitching computation into a pipeline of six producer-consumer stages with the option of having multiple threads per stage.

It establishes one execution pipeline per GPU and, as such, readily takes advantage of multiple GPUs. It decomposes the image grid spatially and allocates one partition per GPU.

Fig. 8 shows the stages of the pipeline. Each stage has an input and an output queue; the threads of a stage consume from its input queue and add items to its output queue. These queues have monitor implementations to prevent race conditions.

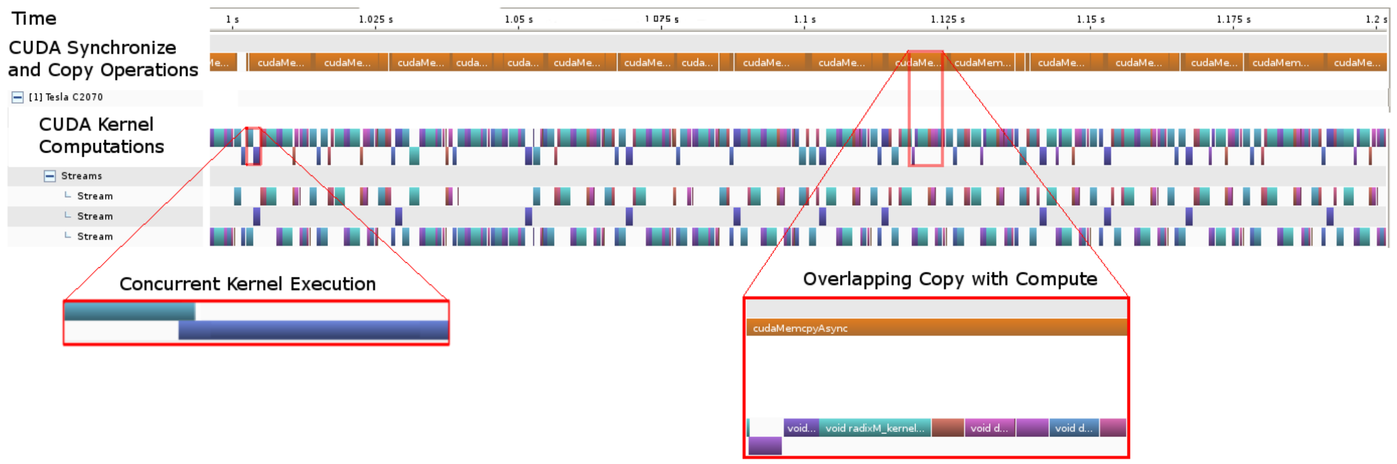

Fig. 9. Profile of a 0.2 s Interval of *Pipelined-GPU* Execution (8 × 8 grid)

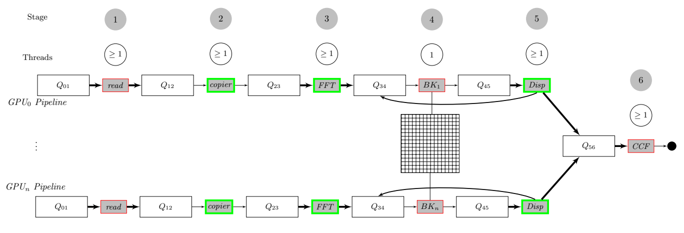

Fig. 8. Multi-GPU Pipeline Architecture

Each execution pipeline processes images as follows:

1) One CPU thread reads image tiles.
2) One CPU thread copies tile data from CPU to GPU memory.
3) One CPU thread initiates FFT computations on the GPU. NVIDIA's cuFFT implementation (v. 5.5) allocates a large number of registers on the NVIDIA Fermi architecture. This prevents the GPU from executing cuFFT kernels concurrently; the architecture pipeline takes this into account by launching one such computation at a time.
4) One CPU thread manages the state of the computation. It resolves dependencies and advances pairs of adjacent tiles that are *ready* (i.e., their FFTs are available) to the next stage.
5) One CPU thread invokes the relative displacement computation (NCC, FFT^{-1}, and max reduction) on pairs of adjacent tiles (north-south or east-west) on the GPU. This stage copies the index of the maximum, a single scalar, from GPU to CPU memory. It also adds an entry to the queue between stages 3 and 4 to handle memory management.
6) Multiple CPU threads, based on the number of available CPU cores, carry out CCF computations. Each thread maps the index of the max, found in the previous stage, to image coordinates and computes the four $\text{CCF}_{ij}^{1..4}$ values. This yields the final x and y relative displacements for the image pair.

The system has special measures for memory management. It allocates a memory pool on the GPU for each pipeline as part of initialization. The system allocates GPU memory only once to avoid any further allocations which would force a global synchronization on all kernels and memory transfers. The pool consists of a fixed number of buffers, one per transform (forward or backward). The size of the pool effectively limits the number of images in flight. Furthermore, every tile has a reference count that is decremented when the tile is used to compute a relative displacement. The system recycles the GPU buffer associated with a tile when its reference count reaches zero; this guarantees that the system does not run out memory. The minimum pool size must exceed the smallest dimension of the image grid; using the *chained diagonal* grid traversal ensures that the system starts recycling GPU buffers as early as possible.

The pipeline architecture uses one CUDA stream per stage to enable the overlapping of asynchronous memory transfers and kernel executions on the GPU. The profile shown in Fig. 7 demonstrated that this functionality is essential to improving performance.

Fig. 9 shows a 0.2 s interval of the *Pipelined-GPU*'s visual profile (out of a total of 1.6 s) which shows a much higher kernel execution density (see "*CUDA Kernel Computations*" row in Fig. 9). This profile does not have the gaps observed in Fig. 7. This is due to (1) having dedicated CPU threads for executing the *CCF* computations, (2) executing memory copies asynchronously along GPU kernel computations, and (3) having 3 CUDA execution streams which allow concurrent GPU kernel executions.

CPU threads are responsible for *CCF* computations. This design has the following two consequences: (1) the system minimizes device to host memory transfers as the input to the *CCF* stage is a scalar, the result of the *max* reduction; (2) the system frees GPU transform memory as early as practical, thereby initiating the computation of additional transforms.

To better compare CPU and GPU performance, we implemented a *Pipelined-CPU* version which includes all the memory mechanisms in its GPU counterpart. The CPU pipeline consists of three stages: *reader*, *displacement/fft*, and *bookkeeping*. In the future, we will modify this implementation to create one execution pipeline per CPU socket. The next section analyzes the run-times of the various implementations and the Fiji image stitching plugin.

V. Results

For each implementation, we ran the image stitching workload (a 42×59 tile grid) ten times and used the average end-to-end run time. Table II summarizes these timing results.

		Speedup		CPU	
Implementation	Time	S/CPU	ImageJ	Threads	GPUs
ImageJ/Fiji	3.6 h		–	5–6	
Simple-CPU	10.6 min	–	20.3	1	–
MT-CPU	1.6 min	6.6	135	16	–
Pipelined-CPU	1.4 min	7.5	154	16	–
Simple-GPU	9.3 min	1.14	23.2	1	1
Pipelined-GPU	49.7 s	12.8	261	16	1
Pipelined-GPU	26.6 s	23.9	487	16	2

TABLE II. Run times and speedups for a 42×59 image grid.

The *Pipelined-GPU* implementation using one GPU completed the relative displacement computation in 49.7 s; this is a 11.2x performance improvement over *Simple-GPU*, corresponding to a 12.8x speedup with respect to *Simple-CPU*. Adding a second GPU to the *Pipelined-GPU* execution improves the run time by 1.87x and processes the same grid in 26.6 s. The improved run time is attributed to instantiating one execution pipeline per GPU as seen in Fig. 8. The system schedules tasks efficiently to fully occupy all CPUs and GPUs, while staying within memory limits.

The *Pipelined-GPU* implementation improves on the runtimes of the CPU-only versions. It reduces the execution times of the *Simple-CPU*, *MT-CPU*, and *Pipelined-CPU* implementations by factors of 23.9x, 3.61x, and 3.16x respectively.

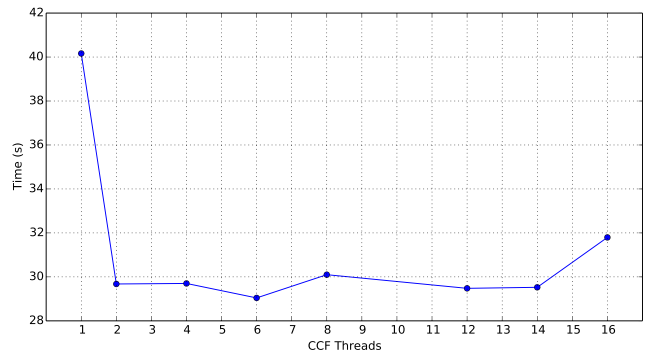

Fig. 10. *Pipelined-GPU* run times with varying CCF Threads

Fig. 10 illustrates the behavior of the *Pipelined-GPU* implementation as the number of *CCF* CPU-threads increases when processing the 42×59 grid with two GPU cards. It shows that increasing the number of *CCF* threads running on the CPU beyond 2 has a minimal impact on execution times; this indicates that the performance bottleneck lies in the GPU computations.

Fig. 11 illustrates the near-linear strong scaling of the *Pipelined-CPU* version as the number of CPU threads is increased when processing the 42×59 grid. The plot shows that the speedup is almost linear as the thread count increases up to 8, the number of physical cores; the speedup curve changes to another linear slope between 9 and 16, the number of logical cores. Furthermore, this behavior is consistent across varying grid sizes (128 to 1024) as shown in Fig. 12.

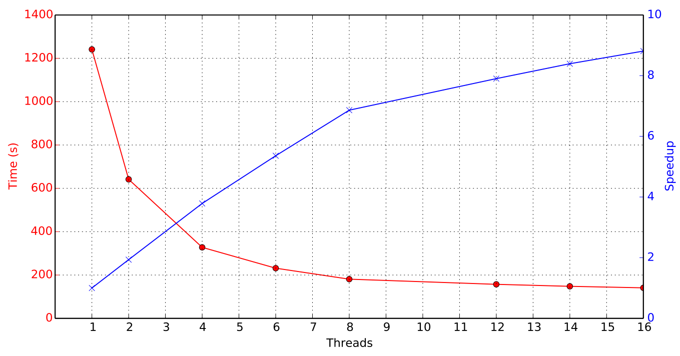

Fig. 11. Scalability of Pipelined-CPU Implementation

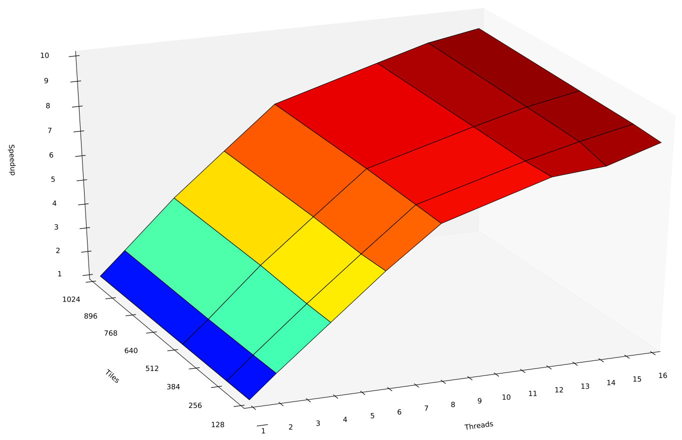

Fig. 12. Speedup Surface of *Pipelined-CPU*

The performance of the *Pipelined-GPU* implementation compares very favorably with the ImageJ/Fiji stitching plugin. This plugin is implemented in Java and is "fully multi-threaded taking advantage of multi-core CPUs" [15]. It computes the relative displacements of the grid in 3.6 h. In contrast, the *Pipelined-GPU* implementation completes the same workload in 49.7 s when using one GPU and in 26.6 s when using two GPUs. These times result in 261x and 487x relative speedups, which are more than two orders of magnitude. Such large reductions in execution times are bound to have a transformative impact on the users of such a tool. At the very least, image stitching for large data-sets becomes a quasi-interactive task and enables computationally steerable experiments centered around optical microscopy.

VI. Conclusions

We started our effort with the goal of supporting long running experiments that repeatedly image microscopy plates at set intervals. These experiments require that the image stitching tool used be fast enough to allow researchers to examine a plate's image, run image analysis tools, and intervene in the experiment before a plate is imaged again. We believe that we have achieved this goal with our pipelined architecture and implementations that carefully manage concurrency and

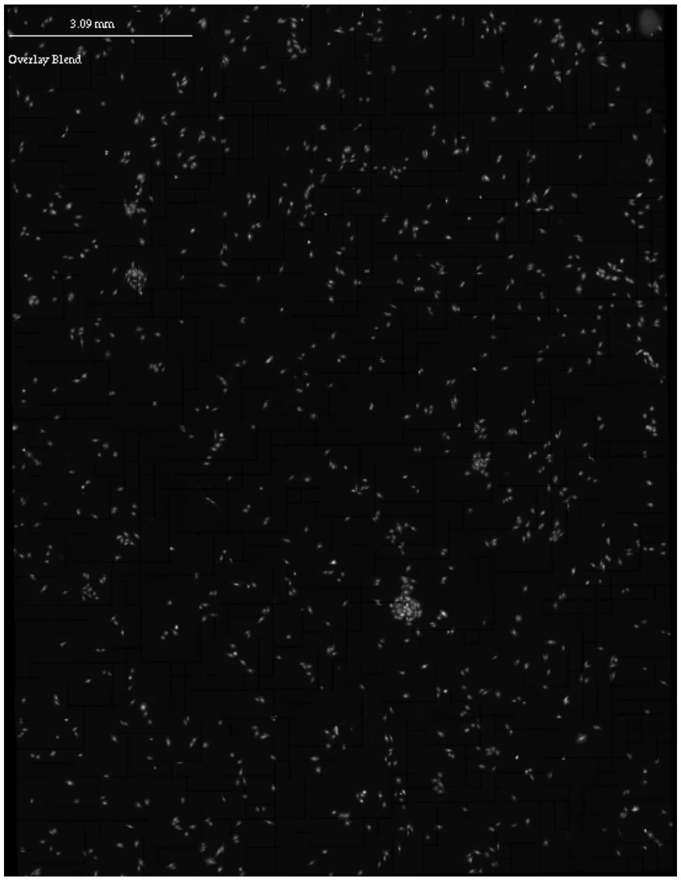

Fig. 13. Stitched 42×59 image grid (17K $\times$ 22K pixels, $\approx 1\text{cm} \times 1.4\text{cm}$), composed using an overlay blend.

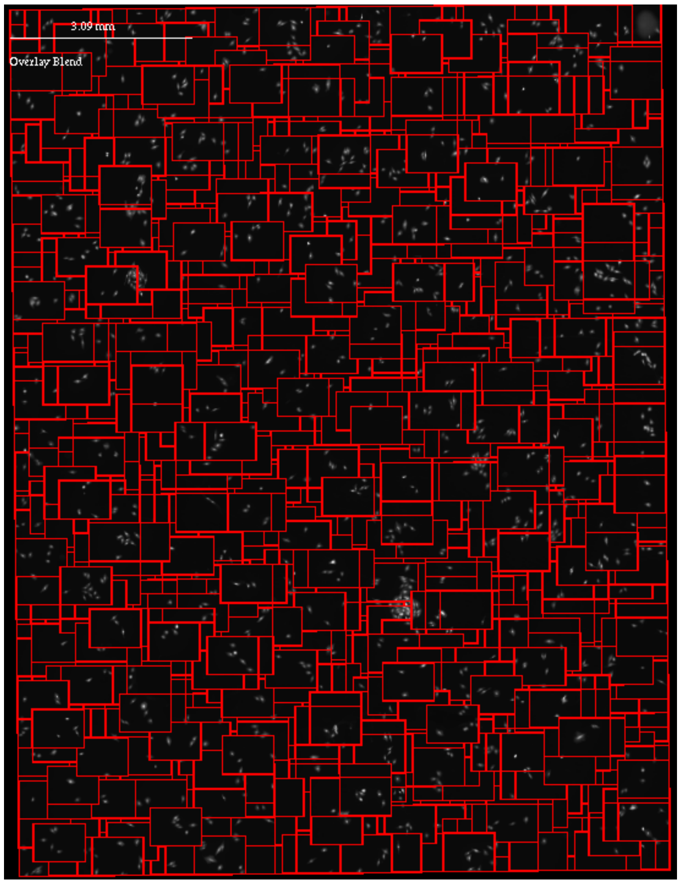

Fig. 14. Stitched 42×59 grid with highlighted tiles

memory. Our *Pipelined-GPU* implementation stitches a large workload (a 42×59 grid) in under $1\,\text{min}$ and can take advantage of multiple CPUs and GPUs; our *Pipelined-CPU* implementation also achieves a respectable $1.4\,\text{min}$ execution time. We further validated our architecture by running it on a 3 year-old laptop with an Intel i7-950 (quad-core), 12 GB of RAM, and an NVIDIA GTX 560M card. The laptop processed the 42×59 grid in $130\,\text{s}$ with *Pipelined-GPU* and in $146\,\text{s}$ with *Pipelined-CPU*.

A. Future Work

One of ImageJ/Fiji's main advantages is its cross-platform nature. We are finalizing the development of a new stitching plugin for ImageJ/Fiji based on our pipeline architecture with the goal of releasing it along with the standalone C++ and CUDA versions. The plugin is written in Java and will run in two modes: one that is pure Java and another that uses native bindings for the CUDA and FFTW libraries if they are installed locally. Preliminary benchmarks reveal that the plugin, running in either mode, will have a performance within a factor of 2–3 of the C++ and CUDA implementations. We are also prototyping a visualization tool to be packaged with the plugin that will generate image pyramids for all the tiles in a grid and render a stitched image at varying resolutions. Figs. 13 and 14 show the composed image using our visualization prototype.

We expect that our algorithm can deliver further performance improvements with NVIDIA's Tesla Kepler GK110 GPUs as this new architecture has additional concurrency capabilities in the hardware. The GK110 architecture features a hardware-based GPU scheduler (Hyper-Q) that allows multiple CPU threads to issue work simultaneously to a GPU [23]. The *Pipelined-GPU* implementation is currently setup so only one CPU thread per stage issues GPU kernel invocations. This can be changed easily to take advantage of Hyper-Q with multiple CPU threads invoking GPU kernels.

Two performance optimizations that we plan to investigate are padding image tiles and using real to complex transforms. Padding image tiles (or trimming them) to have lower prime factors (e.g., 1536×1536) is known to enhance the performance of FFTW and cuFFT because the implementations use divide and conquer techniques. By padding our images, we expect to see performance benefits when computing the forward and inverse FFTs. The other performance optimization (real to complex FFTs) will further lower the computation's memory footprint.

The results of the pipeline implementation demonstrate a highly effective mechanism for structuring the image stitching problem. We plan to evaluate its scalability on a machine with more than 2 GPUs. We also plan to extract a general purpose API for the pipeline, so it can be applied to other problems that can benefit from the GPU. The tool will provide developers a method to overlap disk and PCI express I/O with computation.

Acknowledgments

Dr. Kiran Bhadriraju from University of Maryland at College Park acquired the A10 cell colony images at the NIST cell biology microscopy facility. We would like to acknowledge Dr. Derek Juba of NIST for his many useful suggestions to improve this document.

References

[1] "ImageJ," last access: 2014-02-06. [Online]. Available: http://imagej.nih.gov/ij

[2] "Fiji Is Just ImageJ," last access: 2014-02-06. [Online]. Available: http://fiji.sc

[3] J. Schindelin, I. Arganda-Carreras, E. Frise, V. Kaynig, M. Longair, T. Pietzsch, S. Preibisch, C. Rueden, S. Saalfeld, B. Schmid, J.-Y. Y. Tinevez, D. J. J. White, V. Hartenstein, K. Eliceiri, P. Tomancak, and A. Cardona, "Fiji: an open-source platform for biological-image analysis." *Nature methods*, vol. 9, no. 7, pp. 676–682, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1038/nmeth.2019

[4] K. Venkataraju, M. Kim, D. Gerszewski, J. Anderson, and M. Hall, "Assembling large mosaics of electron microscope images using GPU," 2009, last access: 2012-07-17. [Online]. Available: http://www.cs.utah.edu/sci/publications/kannanuv09/Venkataraju_SAAHPC09.pdf

[5] R. Szeliski, "Image alignment and stitching: a tutorial," *Found. Trends. Comput. Graph. Vis.*, vol. 2, no. 1, pp. 1–104, Jan. 2006.

[6] D. I. Barnea and H. F. Silverman, "A class of algorithms for fast digital image registration," *Computers, IEEE Transactions on*, vol. C-21, no. 2, pp. 179–186, February 1972.

[7] B. Ma, T. Zimmermann, M. Rohde, S. Winkelbach, F. He, W. Lindenmaier, and K. E. Dittmar, "Use of AutoStitch for automatic stitching of microscope images," *Micron*, vol. 38, no. 5, pp. 492–499, 2007.

[8] C. D. Kuglin and D. C. Hines, "The phase correlation image alignment method," in *Proceedings of the 1975 IEEE International Conference on Cybernetics and Society*, 1975, pp. 163–165.

[9] Z. Jing, W. Chang-shun, and L. Wu-ling, "An image mosaics algorithm based on improved phase correlation," in *Proceedings of 2009 International Conference on Environmental Science and Information Application Technology*. IEEE, 2009, pp. 383–386.

[10] J. Lewis, "Fast template matching," *Vision Interface*, vol. 10, pp. 120–123, 1995.

[11] ——, "Fast normalized cross-correlation," 1995, last access: 2014-02-07. [Online]. Available: http://www.scribblethink.org/Work/nvisionInterface/nip.pdf

[12] AutoStitch, "AutoStitch," last access: 2012-12-19.

[13] M. Brown and D. G. Lowe, "Automatic panoramic image stitching using invariant features," *Int. J. Comput. Vision*, vol. 74, no. 1, pp. 59–73, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1007/s11263-006-0002-3

[14] L. Cooper, K. Huang, and M. Ujaldon, "Parallel automatic registration of large scale microscopic images on multiprocessor CPUs and GPUs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. IEE, May 2011, pp. 1367–1376.

[15] S. Preibisch, S. Saalfeld, and P. Tomancak, "Globally optimal stitching of tiled 3d microscopic image acquisitions," *Bioinformatics*, vol. 25, no. 11, pp. 1463–1465, June 2009. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/25/11/1463.abstract

[16] "BOOST C++ library," last access: 2012-07-12. [Online]. Available: http://boost.org

[17] "libTIFF," last access: 2012-07-11. [Online]. Available: http://libtiff.org

[18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[19] "FFTW3," last access: 2012-07-12. [Online]. Available: http://fftw.org

[20] NVIDIA Corp., "CUFFT Library," last access: 2012-04-10. [Online]. Available: http://developer.nvidia.com/cufft

[21] M. Harris, "Optimizing parallel reduction in CUDA," last access: 2014-02-07. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[22] NVIDIA Corp., "Visual profiler," last access: 2012-12-19. [Online]. Available: http://developer.nvidia.com/nvidia-visual-profiler

[23] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," last access: 2013-11-12. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf