COMMUNICATION-MINIMIZING 2D CONVOLUTION IN GPU REGISTERS

Forrest N. Iandola, David Sheffield, Michael Anderson,
Phitchaya Mangpo Phothilimthana, and Kurt Keutzer

Parallel Computing Laboratory (ParLab)
University of California, Berkeley, CA, USA
{forresti, dsheffie, mjanders, mangpo, keutzer}@eecs.berkeley.edu

ABSTRACT

2D image convolution is ubiquitous in image processing and computer vision problems such as feature extraction. Exploiting parallelism is a common strategy for accelerating convolution. Parallel processors keep getting faster, but algorithms such as image convolution remain memory bounded on parallel processors such as GPUs. Therefore, reducing memory communication is fundamental to accelerating image convolution. To reduce memory communication, we reorganize the convolution algorithm to prefetch image regions to register, and we do more work per thread with fewer threads. To enable portability to future architectures, we implement a convolution autotuner that sweeps the design space of memory layouts and loop unrolling configurations. We focus on convolution with small filters (2x2–7x7), but our techniques can be extended to larger filter sizes. Depending on filter size, our speedups on two NVIDIA architectures range from 1.2x to 4.5x over state-of-the-art GPU libraries.

Index Terms— Convolution, parallel, GPU, autotuning

1. INTRODUCTION

Convolution is a key component in most algorithms for feature extraction, image segmentation, object tracking, and object recognition. In a recent "periodic table" of the fifteen most recurring computational patterns in image processing and computer vision literature, *convolution ranked as the most ubiquitous*, followed by histogram accumulation, vector distance, and quadratic optimization [1]. Our work focuses on image convolution with small nonseperable filters (2x2 to 7x7), which are extremely common for edge detection, feature extraction [2], and difference of gaussians [3].

The computer architecture community has developed many-threaded processors that offer tremendous boosts in peak FLOP/s over traditional single-core CPUs. However, improvements to memory bandwidth and latency have lagged behind the improvements to the processors themselves. As a result, the performance of convolution and other algorithms with low computational complexity tend to be limited by the memory bandwidth, much like trying to drink a thick milkshake through a narrow straw.

To recap, the predicament is that parallel processors keep getting faster, but algorithms like convolution remain memory-bounded on these architectures. The solution to this is to redesign algorithms with the goal of minimizing communication among off-chip memory, on-chip shared memory, and registers. On a variety of parallel architectures, reducing and optimizing memory- and interprocess communication has accelerated memory-bounded problems in linear algebra [4] and graph traversal [5] by as much as an order of magnitude. In this paper, we accelerate 2D convolution by reducing communication with off-chip memory, while also avoiding long strides in the access pattern. For 3x3 – 7x7 convolution kernels, we produce a speedups of 1.2x-3.4x on the NVIDIA Fermi architecture, and 1.3x-4.5x on NVIDIA Kepler. These speedups are *not* with respect to CPU implementations; instead *these are speedups over GPU implementations* in the ArrayFire [6] GPU library, which are faster than the NVIDIA-provided convolution routines. Further, for 2x2 and 3x3 filters, our communication-optimized convolution algorithm achieves peak memory bandwidth on NVIDIA Kepler GPUs, and we achieve within a factor of 2 of peak bandwidth on NVIDIA Fermi GPUs.

A common objection to performing algorithms like convolution on the GPU is that copying data from the CPU to GPU can be quite expensive. However, our work is targeted toward image processing pipelines for applications like feature extraction. These pipelines perform a long sequence of image transformations on the GPU, and this more than offsets the CPU-GPU copy time.

Recent work on domain-specific languages such as PetaBricks [7] and Halide [8] has sought to automate, simplify, or autotune image processing algorithms for various parallel architectures. Our work also employs autotuning, but our foremost goal is to produce the fastest possible image convolution implementation for small filter sizes on modern GPUs. Toward this goal, we explore several performance strategies that these DSLs do not explore, such as prefetching image regions to register and varying the amount of work performed by each thread. Ultimately, our work will inform designers of DSLs and libraries about design parameters that can improve convolution performance.

The rest of this paper is organized as follows. In Section 2, we review the relevant aspects of the NVIDIA Fermi and Kepler architectures, and we benchmark these architectures' memory hierarchies. Section 3 describes how we redesign convolution to reduce the time spent waiting for memory accesses. In Section 4, we implement an autotuner that explores the 2D convolution design space to minimize memory communication time on two GPU architectures. We benchmark our work against NVIDIA-provided image processing libraries and other related literature in Section 5, and we conclude in Section 6.

2. ARCHITECTURE

2.1. GPU Architecture Overview

NVIDIA Fermi and Kepler GPUs are comprised of eight to fifteen *streaming multiprocessors (SMs)*, which each execute up to $\sim$1000

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung. The first author is funded by the US DoD NDSEG Fellowship. Research also funded by DARPA Award Number HR0011-12-2-0016. Approved for public release; distribution is unlimited. The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

concurrent threads.[1] Users' GPU-side code is typically implemented in the CUDA or OpenCL language extensions to C/C++. As in a typical Intel CPU, the NVIDIA GPUs have off-chip DRAM called *global memory* which is amplified by system-managed L1 and L2 caches. Also like a typical CPU, NVIDIA GPUs have on-chip *registers*, and each thread has its own register address space that is not accessible to other threads. For the C2050 (Fermi GF100) and GTX680 (Kepler GK104) GPUs that we use in this paper, each thread is allocated a maximum of 63 registers (Table 1), and the registers offer sufficient bandwidth to saturate the arithmetic or floating-point units. NVIDIA's recent K20 (Kepler GK110) can allocate up to 255 registers per thread. Each SM also has a small read-only *constant memory*, which is as fast as the registers. Unlike most CPUs, the NVIDIA GPUs also have a user-managed fast read-only pipeline to global memory called the *texture cache* (texcache), as well as a user-managed on-chip cache called *shared memory* (shmem). Shared memory address spaces are common within each thread block, but not globally across the GPU.

In Table 1, notice that the on-chip memory is quite limited. If an implementation uses a large number of registers per thread, then fewer threads can run concurrently. In other words, when each thread uses a larger portion of the register space, occupancy goes down.

Table 1. NVIDIA Memory Space Per Streaming Multiprocessor (SM) [9] for C2050 (Fermi) and GTX680 (Kepler).

	Max 4KB Registers Per Thread	Registers Per SM	Shmem Per SM	Texcache Per SM
C2050	63	128KB	48KB	12KB
GTX680	63	256KB	48KB	48KB
K20	255	256KB	48KB	48KB

Data is persistent in the off-chip memory (global memory and the texture cache), but the on-chip memory (register and shared memory) contents are not guaranteed to persist beyond the lifetime of a thread. As a result, it's necessary to store data (e.g. images) in the off-chip memory, and to load this data to registers when performing computations such as convolution. We spend the remainder of this section discussing and benchmarking the NVIDIA memory hierarchies, with the goal of informing convolution algorithm design decisions later in the paper.

2.2. Benchmarking the Memory Hierarchy

We now turn to benchmarking the key properties of the NVIDIA Fermi and Kepler memory hierarchy. In Table 2, we empirically benchmark the bandwidth of the global memory and shared memory, again using benchmarks described in [10].[2] Our global memory bandwidth results are for memory accesses with unit stride–adjacent threads access adjacent global memory addresses. Longer strides reduce the usable memory bandwidth, because the hardware coalescers are optimized for unit stride. We direct the interested reader to [11] for a discussion of strides and coalescing in an earlier generation of NVIDIA hardware.

Notice in Table 2 that the newer GTX680 has slightly less shared memory bandwidth than the C2050. This is in part due to the fact that the GTX680 has fewer streaming multiprocessors than the C2050. We also attempted to benchmark the texture cache, but our microbenchmarking experiments did not come close to attaining the theoretical texture cache fill rate. So, Table 2 reports the theoretical texture cache fill rate reported by NVIDIA [12]. A key point about the texture cache is that Kepler's fill rate is 2.6x greater than Fermi's.

[1]512 threads per SM on Fermi, and 1536 threads per SM on Kepler.

[2]Given this paper's space limitations, memory bandwidth is our main benchmark. Memory latency is also a useful parameter. We suggest [10] for data and benchmark implementations for NVIDIA memory latency.

Table 2. NVIDIA Memory Bandwidth – Global Memory, Texture Cache, Shared Memory on C2050 (Fermi) and GTX680 (Kepler).

	Measured Global Bandwidth	Theoretical Texcache Fill-rate [12]	Measured Shmem Bandwidth (full GPU)
C2050	90.4 GB/s	49.4 Gtexels/s	931 GB/s
GTX680	123GB/s	129 Gtexels/s	893 GB/s

3. COMMUNICATION-MINIMIZING CONVOLUTION IMPLEMENTATION

In the early days of CUDA, NVIDIA advocated storing data (e.g. images) in global memory, then loading this data to shared memory (much like loading to cache on a CPU) for computation. However, Volkov and Demmel showed that higher performance can be obtained by unrolling loops and prefetching data up to registers instead of working out of shared memory. As a result, register prefetching and loop unrolling has become a common practice for linear algebra problems like matrix-matrix multiplication [13]. The key intuition is that, since global and and even shared memory communication is expensive, prefetching and unrolling can increase in-register data reuse. We now discuss our strategy for implementing convolution, where *our goal is to minimize the time spent communicating with the caches and off-chip memory*.

Our algorithm works as follows. Each thread begins by prefetching a region of the image from off-chip memory into its registers. Prefetching gives the compiler more flexibility to do instruction-level parallelism (ILP) and to overlap communication with computation. For example, when using a 3x3 convolution filter, we might prefetch a 4x4 region of the image into registers. Then, using a user-provided convolution filter that we store in the constant memory, we compute a region of output pixels. Finally, we write the output pixels back to off-chip memory. For the example with a 3x3 filter and 4x4 region in registers, each thread would produce four output pixels (right side of Figure 1). We use the term *loop unrolling* to describe implementations that produce more than one output pixel per thread. Loop unrolling reduces the total number of requests for data in off-chip memory, and it can further increase ILP.

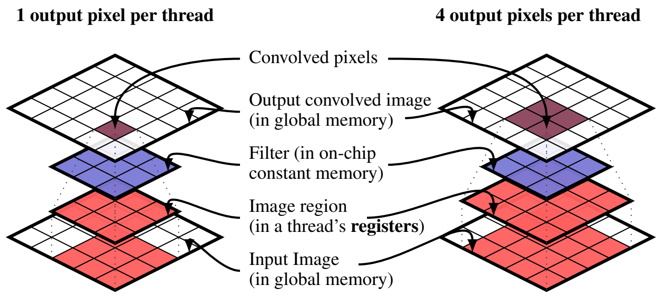
1 output pixel per thread
4 output pixels per thread
Convolved pixels
Output convolved image (in global memory)
Filter (in on-chip constant memory)
Image region (in a thread's registers)
Input Image (in global memory)

Fig. 1. Loop unrolling: More work per thread. *Left:* Typical approach, one output pixel per thread. *Right:* Our optimal implementations produce multiple output pixels per thread.

A slight modification to this approach is to first copy a large image region to each thread block's shared memory, then copy from shared memory to registers. Using shared memory allows for cooperative loading: adjacent threads can load adjacent pixels from off-chip to shared memory, which maximizes coalescing even in implementations with unrolled loops. A trade-off is that we pay bandwidth and latency penalties for one set of loads from off-chip memory, plus two sets of accesses to store and load in shared memory.

4. AUTOTUNING

We now turn to exploring the design space of places to store the image in off-chip memory (texture cache or global memory), the amount of work to do per thread (loop unrolling), and whether or not to prefetch data to registers. We also evaluate the impact of first fetching an image region to shared memory, then distributing the pixels from shared memory to registers. Finally, hard-coded loop bounds allow the compiler more freedom to add further performance improvements, so our autotuner produces a broad range of hard-coded implementations in the convolution design space.

We show our autotuner's findings for 3x3 filters on NVIDIA Fermi in Figure 2 and on Kepler in Figure 3. Targeting high-resolution surveillance cameras such as the 9216x9216 CCD595 from Fairchild Imaging [14], we use 9216x9216 1-channel floating-point images in Figures 2–5. Bear in mind that convolution's computation time scales linearly with the number of pixels. Therefore, the performance stratifications in these figures generalize to images that are sufficiently large to saturate the GPU–approximately 640x480 or larger for our unrolled implementations.

Observe in Figures 2 and 3 that the unrolled (4 or more outputs per thread) global$\rightarrow$register implementations produce similar performance regardless of the amount by which we unroll the loop. Since more unrolling should lead to more ILP and fewer memory accesses, we might expect performance to continue to improve as we increase the amount of work per thread. However, more unrolling leads to longer strides in memory accesses which, as discussed in Section 2.2, reduces coalescing and thus reduces usable bandwidth. Also, while more unrolling reduces the number of off-chip memory requests, the L1 and L2 caches enable data reuse and reduce the penalty for the redundant memory accesses in overlapping window algorithms like convolution. Further, more unrolling (e.g. 25 outputs per thread) increases the registers allocated per thread, thus reducing occupancy. In short, we find that unrolling is important for improving performance, but factors like register pressure, ILP, occupancy, and strided accesses balance out so that global$\rightarrow$register implementations are not particularly sensitive to the amount by which the loop is unrolled.

Also notice in Figures 2 and 3 that the strategy of loading to shared memory, then to register actually diminishes performance slightly. While the shared memory step can reduce the number of memory accesses and increase coalescing, it also adds a thread synchronization and an extra set of load and store penalties from the shared memory's bandwidth and latency. In contrast, loading directly from global memory to registers exploits the L1 cache and requires no synchronization. As we discussed in the previous paragraph, the L1 cache is amenable to the overlapping windows used in convolution. Tables 3 and 4 show that using shared memory doesn't improve performance for convolution with 2x2 – 7x7 filters.

For brevity, we limit the autotuning visualization (Figures 2 and 3) to 3x3 filters, but we summarize the autotuner's optimal implementations in Tables 3 and 4. Tables 3 and 4 also show a comparison between the optimal autotuned results and our basic "global memory only" that uses hard-coded loop bounds but does not prefetch to register. Out of the existing 2D convolution implementations that we will benchmark in Section 5, ArrayFire [6] is the provides the fastest implementation (out of the implementations that provide the full 2x2 – 7x7 range). With this in mind, we also provide speedup numbers with respect to ArrayFire in Tables 3 and 4.

In our experiments, we define the *bandwidth bound* as the amount

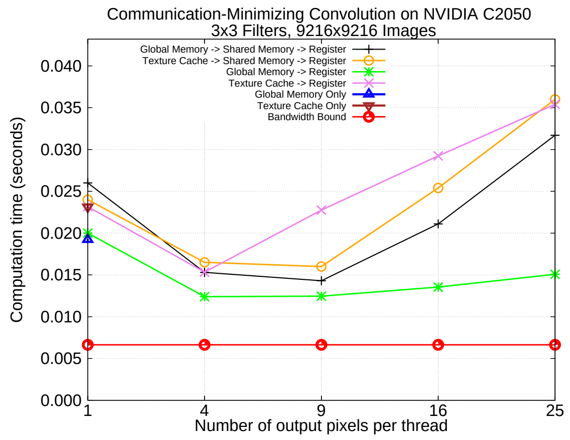

Fig. 2. Convolution performance with several memory paths and loop unrolling configurations on C2050 (Fermi GF100). 3x3 convolution filters.

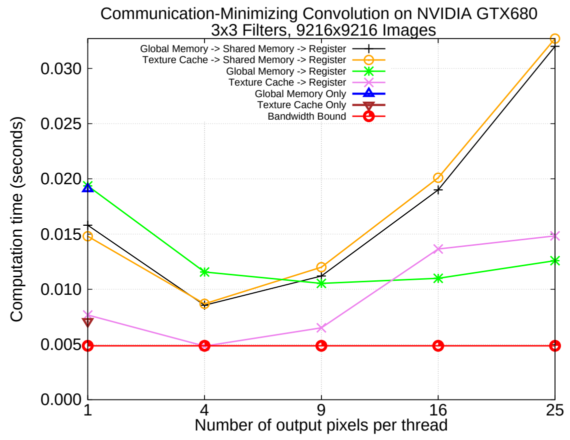

Fig. 3. Convolution performance with several memory paths and loop unrolling configurations on GTX680 (Kepler GK104). 3x3 convolution filters.

of time to transfer one 9216x9216 image from global memory to registers and back, *not* including the overlapping memory accesses that we use in convolution. In Tables 3 and 4, the "% of BW Bound" column is calculated as $\frac{convolutionTime}{bandwidthBoundTime}$. On Kepler, our convolution implementations with small filters can exceed the global memory bandwidth bound by using the texture cache. Notice that our results are within 2x of the global bandwidth bound for 2x2 and 3x3 filters on Fermi (Table 3), and within 2x of the global bandwidth bound for 2x2–5x5 filters on Kepler (Table 4). The GPU allocates a maximum of 63 registers per thread, and each input and output pixel uses one register in our prefetching implementations, so loop unrolling is quite limited for the 6x6 and 7x7 filter sizes. However, our prefetching and unrolling strategy could be extended to efficiently handle 7x7 and larger filters. Specifically, we would handle 7x7 and larger filters by having each thread load the pixels it needs in small blocks, alternating between loading input pixels and computing a convolved output pixel. In addition, NVIDIA's new "Big Kepler" GK110 architecture allocates up to 255 registers per thread, so we anticipate that this architecture will be more amenable to unrolling with larger filter sizes.[3]

[3]NVIDIA K20 (Kepler GK110) was released in late 2012. We were not able to obtain a K20 in time for press.

Table 3. Optimal Convolution Implementations – C2050 (Fermi GF100). Speedups with respect to our simple "global memory only" implementation and ArrayFire [6].

Filter Size	Optimal Layout	Output Pixels Per Thread	Speedup vs. ArrayFire	Speedup vs. Global Only	% of BW Bound
2x2	Global→Register	1	1.2x	1.1x	55%
3x3	Global→Register	4	2.0x	1.5x	53%
4x4	Global→Register	16	2.6x	1.6x	40%
5x5	Global→Register	9	3.1x	1.9x	33%
6x6	Global→Register	4	3.4x	2.1x	25%
7x7	Global→Register	1	1.8x	1.0x	9.4%

Table 4. Optimal Convolution Implementations – GTX680 (Kepler GK104). Speedups with respect to our simple "global memory only" implementation and ArrayFire [6].

Filter Size	Optimal Layout	Output Pixels Per Thread	Speedup vs. Array-Fire	Speedup vs. Global Only	% of BW Bound
2x2	Texcache→Register	4	1.7x	1.9x	107%
3x3	Texcache→Register	4	2.2x	3.9x	101%
4x4	Texcache→Register	9	2.4x	6.1x	87%
5x5	Texcache→Register	9	4.5x	8.8x	83%
6x6	Texcache→Register	4	3.5x	7.5x	49%
7x7	Texcache→Register	1	1.3x	2.9x	14%

5. COMPARISON WITH RELATED WORK

Several libraries such as OpenCV [15], NVIDIA Performance Primitives [16], ArrayFire [6], PetaBricks [7], and CUVILib [17][4] provide **GPU** implementations of 2D image convolution. Analysis of the OpenCV source code reveals that OpenCV works directly out of the texture cache, and it does not use hard-coded loop bounds, although it does store the convolution filter in on-chip constant memory. NVIDIA Performance Primitives (NPP) [16] is a closed-source codebase, but decompilation of NPP binaries using `cuobjdump` suggests that NPP does not employ register prefetching or loop unrolling in its 2D convolution. PetaBricks is a domain-specific language that allows algorithms to have several different implementations, and it incorporates autotuning and code generation [7]. PetaBricks was recently extended to generate and optimize OpenCL code for GPUs and CPUs, and it is capable of generating 2D convolution code [18]. Convolution can also be phrased as FFT: `ifft(fft(image) * fft(filter))`. In addition to testing ArrayFire's direct 2D convolution, we use ArrayFire's wrapper around NVIDIA cuFFT [19] as a baseline for comparing our direct convolution implementations with FFT-based convolution.

We now compare our autotuned results with the 2D nonseperable convolution implementations in the aforementioned image processing libraries. The "Our Communication-Minimizing Result" line in Figures 4 and 5 represents the best implementation produced by our autotuner, as enumerated in Tables 3 and 4. First, notice that FFT is one to two orders of magnitude slower than any of our implementations on both architectures. On Fermi, the NVIDIA Performance Primitives convolution is 2-3x slower than our *naive* implementations without blocking or prefetching, and even slower compared to our autotuned results. On Kepler, the NPP performance is almost identical to our naive implementations. PetaBricks is the only related work that we tested that uses hard-coded loop bounds, and as a result it is quite fast. Note that PetaBricks only supports odd-sized convolution filters. Recall that Tables 3 and 4 show our speedups with respect to ArrayFire. Our speedups are most significant for small kernels, which offer a lot of flexibility to unroll loops despite the constrained register file size.

[4]The CUVILib GPU library [17] primarily supports Windows; we were unable to get CUVILib's 2D convolution running on our Linux system.

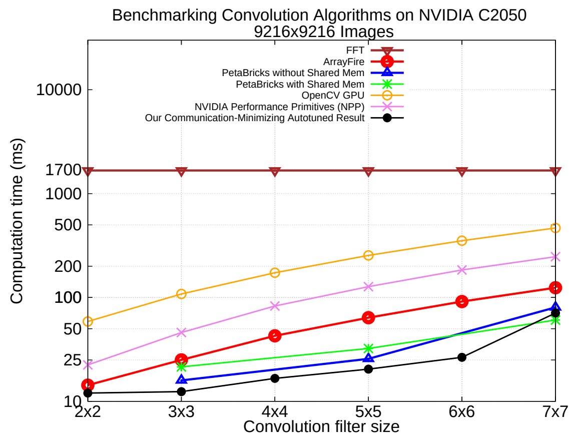

Fig. 4. Comparison of our convolution performance with related work on C2050 (Fermi GF100).

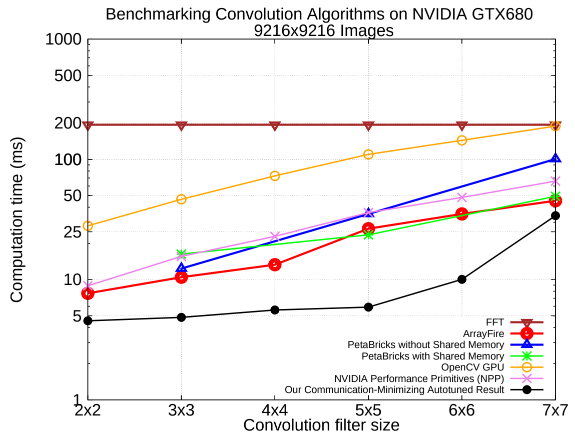

Fig. 5. Comparison of our convolution performance with related work on GTX680 (Kepler GK104).

6. CONCLUSIONS

Convolution with small filter sizes is widely used in edge detection, and it underpins numerous algorithms for feature extraction. Toward accelerating all of these problems, we accelerate nonseperable 2D convolution on NVIDIA GPUs. Convolution is bandwidth bound on GPUs, so we focus on reducing the time spent performing memory accesses. We achieve bandwidth bound for 2x2 and 3x3 filters on NVIDIA Kepler by performing more work per thread and prefetching to registers. For portable performance in future architectures, we have implemented an autotuner that explores the design space of 2D convolution with small filters.

Our approach in this paper has been to optimize memory communication using strategies that do not appear to be implemented in today's domain-specific languages (DSLs) and libraries. We plan to incorporate this paper's performance optimizations into productivity-oriented DSLs like Halide or PetaBricks. Further, our study of optimal register blocking and data movement for 2D convolution will inform the design of composable, in-register image processing pipelines with minimal memory communication.

ACKNOWLEDGEMENTS
The authors would like to thank James Demmel, Leonid Oliker, Pavan Yalamanchili, Yun-Ta Tsai, and Kari Pulli.

7. REFERENCES

[1] Bor-Yiing Su, *Parallel Application Library for Object Recognition, Chapter 3*, Ph.D. thesis, University of California, Berkeley, 2012.

[2] Navneet Dalal and Bill Triggs, “Histograms of oriented gradients for human detection,” in *CVPR*, 2005.

[3] David G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, 2004.

[4] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick, “Avoiding communication in sparse matrix computations,” in *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2008.

[5] Scott Beamer, Krste Asanovic, and David Patterson, “Direction-optimizing breadth-first search,” in *Supercomputing*, 2012.

[6] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos, “ArrayFire: a GPU acceleration platform,” *SPIE Modeling and Simulation for Defense Systems and Applications*, 2012.

[7] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe, “Petabricks: A language and compiler for algorithmic choice,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[8] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Fredo Durand, “Decoupling algorithms from schedules for easy optimization of image processing pipelines,” in *SIGGRAPH*, 2012.

[9] “NVIDIA’s next generation CUDA compute architecture: kepler GK110,” NVIDIA Whitepaper, 2012.

[10] Michael J. Anderson, David Sheffield, and Kurt Keutzer, “A predictive model for solving small linear algebra problems in GPU registers,” in *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

[11] Nathan Bell and Michael Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Supercomputing*, 2009.

[12] “NVIDIA geforce gtx 680: The fastest, most efficient GPU ever built,” NVIDIA Whitepaper, 2012.

[13] Vasily Volkov and James W. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *Supercomputing*, 2008.

[14] “Fairchild Imaging CCD595,” `fairchildimaging.com/products/fpa/ccd/area/ccd_595.htm`.

[15] “OpenCV,” `opencv.willowgarage.com`.

[16] “NVIDIA Performance Primitives (NPP),” `developer.nvidia.com/npp`.

[17] Salman Ul Haq, “Cuvilib: GPU accelerated vision & imaging library,” in *GPU Technology Conference (GTC)*, 2010.

[18] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe, “Portable performance on heterogeneous architectures,” in *Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2013.

[19] “cuFFT,” `developer.nvidia.com/cufft`.