https://doi.org/10.2352/ISSN.2470-1173.2017.12.IQSP-224

Microarchitectural analysis of a GPU implementation of the Most Apparent Distortion image quality assessment algorithm

Vignesh Kannan[1], Joshua Holloway[1], Sohum Sohoni[1], and Damon M. Chandler[2]; [1]Arizona State University, [2]Shizuoka University; [1]Arizona, USA; [2]Shizuoka, JAPAN

Abstract

Due to the massive popularity of digital images and videos over the past several decades, the need for automated quality assessment (QA) is greater than ever. Accordingly, the impetus on QA research has focused on improving prediction accuracy. However, for many application areas, such as consumer electronics, the runtime performance and related computational considerations are equally as important as the accuracy. Most modern QA algorithms exhibit a large computational complexity. However, the large complexity of these algorithms does not necessarily prohibit their ability of achieving low runtimes if hardware resources are used appropriately. GPUs, which offer a large amount of parallelism and a specialized memory hierarchy, should be well-suited for QA algorithm deployment.

In this paper, we analyze a massively parallel GPU implementation of the most apparent distortion (MAD) full-reference image QA algorithm with optimizations guided by a microarchitectural analysis. A shared memory based implementation of the local statistics computation has yielded 25% speedup over its original implementation. We describe the optimizations that produce the best results. We also justify our optimization recommendations with descriptions of the microarchitectural underpinnings. Although our study focuses on a single algorithm, the image-processing primitives used in this algorithm are fundamentally similar to those used in most modern QA algorithms.

Introduction

Image quality simply is how good the image appears to an observer. It is a measure of how accurate the image of a subject represents the subject. Digital images are rapidly becoming part of our daily lives in the form of photos and videos of different resolution[18]. Right from the acquisition of an image, whether it is transmitted over the internet or stored on a disk, image processing is done as part of the standard. It is not always possible to use lossless compression because of its bad compression ratio and lossless compression techniques cannot guarantee compression for all input datasets[19]. Lossy compression schemes introduce blurring and ringing effects, leading to quality degradation [2]. Hence, it is critical to analyze the impact of the effects caused by distortion on image's visual quality bringing in the need for Image quality assessment algorithms.

In applications where the end-users are humans, the default method of quantifying image quality is through evaluation by the subject, which is usually expensive, inconvenient, subject-biased, and time-consuming [8]. This introduces a need for automated quality prediction. In order and to fulfil this requirement, objective image quality assessment was introduced to develop methods that can predict perceived image quality automatically.

Until 2010, research on IQA algorithms were focusing only on prediction fidelity with very less importance to practical constraints such as algorithmic, runtime and microarchitectural complexities [3] [4] [5]. When IQA algorithms march into production scenarios, the runtime performance and related computational considerations become as important as the prediction accuracy. There has been very little research on accelerating IQA algorithms using hardware techniques such as GPU or FPGA. GPU implementations of SSIM[24], MSSIM[23] and MAD[9] yielded 150x, 35x and 24x speedups over their corresponding CPU versions [6][7].

In this paper, we perform microarchitectural analysis of a CUDA [25] implementation of MAD algorithm to identify a CUDA portion of code with the largest bottleneck. Then, we exploit shared memory[11] feature provided by the NVIDIA GPU to resolve the bottleneck and improve performance. By exploiting the microarchitectural features of the GPU, it is possible to achieve a better match between the what the algorithm requires and what the underlying hardware can offer thus utilizing the GPU to its full potential. Most Apparent Distortion IQA algorithm is selected because it is currently the best predictive performance IQA algorithm.

Related Work

It is crucial to have knowledge of the underlying GPU hardware for efficient programming. Programmers can improve the efficiency by tailoring their algorithm specifically for parallel execution. Che *et al.*[12] explored the GPU bottlenecks on different applications in terms of memory overhead, shared memory bank conflict and control flow overhead setting the stage for further research on GPU bottlenecks. Harris[1] discussed different strategies for doing parallel reduction such as interleaved addressing with divergent branches, interleaved addressing with bank conflicts, sequential addressing and optimal method of doing computation while loading the data from global memory. This paper[1] proposed the idea of using a sliding window across the shared memory as well as the need to avoid bank conflicts showing a speeding up 30X over the naive implementation. Our research takes inspiration from the shared memory implementation[1] to resolve the memory bottleneck as described in Experiment 1.

Tuning strategies to improve performance, such as coalescing, prefetching, unrolling, and occupancy maximization are introduced in classical CUDA textbooks[13]. Ryoo, S. *et al.*[14][15] discussed different tuning strategies and also show how optimum usage of hardware resources is critical for occupancy and

performance. However, the entire study focused on a pre-Fermi architecture. An analytical performance model discussed by Hong .S [16] provided details of the number of parallel memory requests by using details about currently running threads and memory bandwidth consumption. Performance analysis via profiling can yield invaluable information in understanding the behavior of GPUs[17], which is the model adopted in this paper for microarchitectural analysis.

It is common to observe irregular memory accesses on the GPU. Wu, B.*et al.*[20] discussed reorganizing data to minimize non-coalesced memory access. Brodtkorb *et al.* [21] gave a detailed picture on profile driven development, stressing the importance of iterative programming and optimization. The authors[21] got into detail about using the NVIDIA profiler to profile the implementation and by using the data, improving a local search. Micikevicius[22] asserted the importance of increasing the memory bandwidth, optimum utilization of compute resources, instruction, and memory latency by discussing about profiler driven analysis and optimization. The author[22] provided a note on the essential profiling parameters to consider and possible conclusions to be drawn from the data, which is the model applied in the experiment section of this paper.

There is no prior research on microarchitectural analysis of image quality assessment algorithms on a GPU and this document provides first of its kind microarchitectural analysis of a GPGPU implementation of an image quality assessment algorithm, specifically the most apparent distortion (MAD) algorithm. While this analysis is specific to a CUDA implementation of MAD, it can provide insight into other related algorithms, which can reuse the concepts discussed in this document.

Experimental Setup (Apparatus and Stimulus)

The GPU version of MAD was developed using NVIDIA's CUDA API and the CPU portion of the code uses C++. GPU Profiling of the implementation is performed by NVIDIA Nsight[10] and Visual Profiler. The apparatus consists of a modern desktop computer with Intel I7 processor and two NVIDIA GPUs. For this experiment, we are using NVIDIA Tesla K40.

Table 1: Details of The Test System

CPU	Intel® Xeon® Processor E5-1620 @ 3.70 GHz Cores: 4 cores (8 threads) Cache: L1: 256 KB, L2: 1 MB, L3: 8 MB
RAM	RAM: 24GB DDR3@1866MHz(dual channel)
OS	Windows 7 64-bit
Compiler	Visual Studio 2013 64-bit;
GPU1	NVIDIA Tesla K40(PCIe3.0)
GPU2	NVIDIA NVS 310 (PCIe3.0)

The application entail two experiments.

1. Microarchitectural analysis of the statistical computations of the CUDA MAD implementation which take the worst running time and least occupancy to identify the bottleneck.
2. By using the microarchitectural information about the underlying GPU - shared memory, the bottleneck identified in Experiment 1 is resolved.

EXPERIMENT 1: Microarchitectural analysis of CUDA-MAD

Using NVIDIA Visual Profiler, local statistics component of the current MAD implementation has been identified as the code with highest bottleneck. Every thread running an instance of the local statistics computation does the following operations.

IN: 512 x 512 image OUT: Three 128x128 arrays corresponding to local statistics (standard deviation, skewness and kurtosis).

1. Declare a 1D array of 256 elements.
2. Gather 16x16 data from global memory [11] and store into its own local memory from step 1.
3. Sum all the elements of its local array through a 1D traversal. Using the sum, calculate the mean.
4. Using the mean value, calculate standard deviation, skewness and kurtosis.
5. Scatter the calculated values across the corresponding memory locations in global memory.

In order to evaluate the development process guided by the profiler [22], in this study, the current MAD implementation is profiled in terms of Memory bandwidth, Compute resources, Instruction and memory latency. Figure 1 identifies Memory bottleneck to be the primary performance limiter. Hence, we will focus on Memory bandwidth. Compute resources and Latency are not covered as part of this paper.

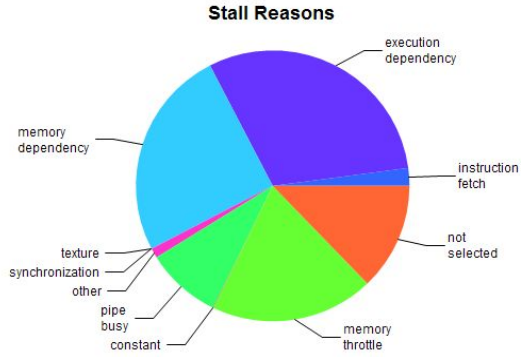

figure 1. *Stall Reasons*

Memory Bandwidth:

Memory Bandwidth is the rate at which data is read or written from the memory. On a GPU, bandwidth depends on efficient usage of memory subsystem, which involves L1/shared memory, L2 cache, Device memory and System memory (via PCIe). Since there are many components in the memory subsystem, separate profiling is done to collect data from the corresponding subsystem. Memory statistics are collected from Global, Local, Shared, Buffer and Caches.

- **Global**: Performs profiling on memory operations to the global memory. Specifically focuses on the communication between streaming multiprocessors (SM) and L2 cache.

Note: Memory statistics from Local, Shared, Buffer and Caches are not taken into consideration in this paper.

Memory Statistics – Global:

Global device memory can be accessed in two different data paths; Data traffic can go either through (L2 and/or L1), read only global memory access can alternatively go through the read-only data cache/texture cache. NVCC compiler has control over the behavior of caches by setting appropriate compilation flag. In this experiment, no explicit setting has been made. In Figure 1, cached loads uses the L1 cache or texture cache as well as L2 whereas uncached loads uses only the L2 cache.

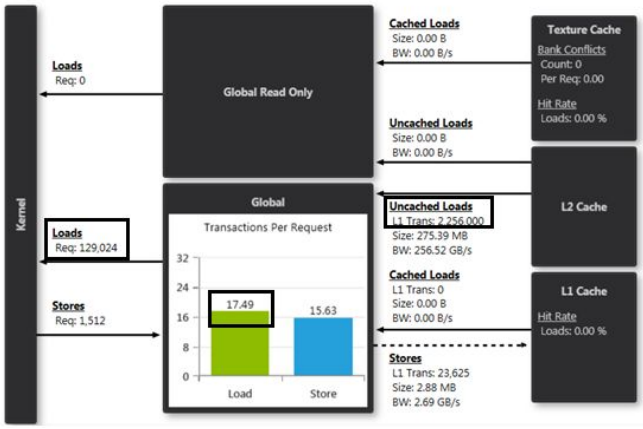

figure 2. *Memory Statistics - Global*

A warp [25] in execution accessing device memory using LD or ST assembly instruction coalesces the memory accesses of all the 32 threads in a warp into one or more of these memory transactions depending on the size of the word accessed by each thread. It can be observed that if all the threads within a warp performs random stride, coalescing gets disturbed resulting in 32 different accesses in a warp. Figure 1 shows the average number of L1 and L2 transactions required per executed global memory instruction, separately for load and store operations. Lower numbers are better; It is better to have 1 transaction for a 4-byte access (32 threads * 4 byte = one 128 byte cache line), 2 transactions for a 8-byte access (32 threads * 8 byte = two 128 byte cache lines) access. The exact lines of code performing global memory access is described in Table 2.

Table 2: Global memory access in original implementation

```
for(x = global_threadIdx_x; x < (x + tile_size) ; ++x)
    {
        for(y = global_threadIdx_y; y < (y+ tile_size) ; ++y)
            {
                local_memory[index] = global_memory[x *
SIZE_OF_IMAGE + y];
                index = index + 1;
            }
    }
```

Note: tile_size = 16, SIZE_OF_IMAGE = 512

A memory "request" is an instruction, which accesses memory, and a "transaction" is the movement of a unit of data between two regions of memory. In this seemingly benign, otherwise correct implementation, 129,024 requests are made resulting in 2,256,000 transactions. Each of the requests are 4-byte requests (float). Hence, there are almost 17 transactions made for each load request. This exorbitant number of transactions and global memory access latency of 200 - 400 cycles indicate the need to reduce access to global memory. Moreover, the current implementation does not use the shared memory effectively.

EXPERIMENT 2: Microarchitectural analysis of CUDA-MAD

Using proposed changes from Experiment 1, the current local statistics computation will be modified by exploiting the shared memory feature provided by the GPU to reduce global memory access and the results will be compared.

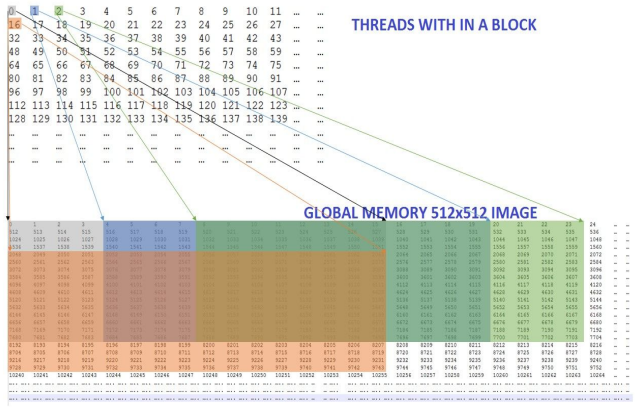

figure 3. *Global Memory Stride Visualization*

From Figure 3, it can be inferred that there is overlap among the elements gathered from global memory by subsequent threads. Also, every thread fetches 128 bytes of data in a single request i.e. when thread 0 requests data, 128-bytes are provided to the thread (coalesced memory access). However, the requester thread utilizes only 4-byte, other threads in the warp utilizing only 60 bytes data thus discarding rest of the fetched in data. Only the mean computation in local statistics calculation takes 1.000512 ms.

Proposed changes in shared memory implementation:

In the original implementation, nested for loops are involved in fetching the data from global to local memory as seen in Table 2. In shared memory implementation, every thread will access a memory location based on its global thread id. Every thread copies the data from global memory referenced using its global thread id

to location in shared memory corresponding to its local thread id. It can be calculated as given in Table 4.

Table 4: Shared memory implementation to collect global data

```
int global_idx = (threadIdx.x + blockIdx.x * blockDim.x);
int global_idy = (threadIdx.y + blockIdx.y * blockDim.y);
shared_memory[threadIdx.x][threadIdx.y]=
global_memory[global_idx* N + global_idy];
```

Note: N = 512

In shared memory implementation, a thread is launched for each individual pixel. Every thread accesses four locations from global memory and stores in the shared memory as seen as colored blocks in Figure 4. As soon as all the threads bring in the data, a 2D sliding window of 16 * 16 size iterates over the shared memory to calculate the mean using nested for loops.

Optimization:

The nested loop for calculating the sum is very inefficient with the implementation taking 1.6ms. Hence, instead of nested loop to calculate the sum of the sliding window, the inner loop is unrolled manually. This is done by exploiting the fact that the window comprises of 16 x 16 elements. The runtime of the kernel has dropped down to 0.9 ms from 1.0ms.

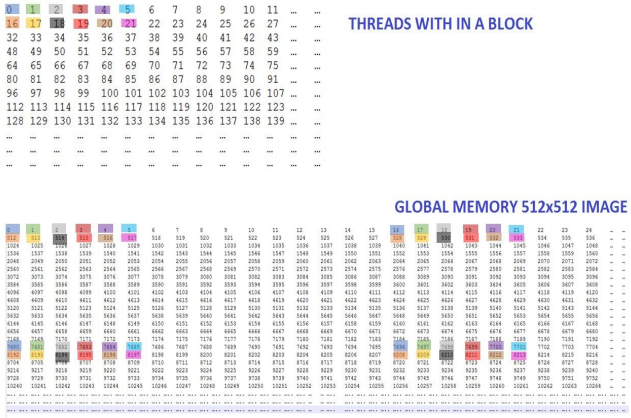

figure 4. *Shared memory implementation*

Bank conflict:

Theoretically, shared memory access is 20-30x faster than global memory. Hence, speed up of only 10% prompts the need for further analysis into the shared memory implementation. The on-chip memory is partitioned into equal sized memory modules called banks which can be accessed concurrently at the same time. However, if multiple threads accesses the same bank, the requests get serialized decreasing the memory bandwidth. There are 32 banks in Tesla K40. The bandwidth of shared memory is 32 bits per clock cycle per bank.

Shared memory bank conflicts in the current implementation can be visualized in Figure 5. Unlike global memory where typical memory access is made by coalescence, shared memory access is request - delivery type i.e. if a 4-byte access to shared memory is made, a 4-byte data chunk should be returned. Ideally 1 shared memory request =1 shared memory transaction.

In Figure 5, 1 shared memory request = 4 shared memory transactions. This indicates conflict in shared memory access. Threads within a warp are numbered in the equivalent of column major order. Hence using shared_memory[threadIdx.x][threadIdx.y] causes threads in a warp reading from the same column which interprets to reading from the same memory bank resulting in bank conflicts. Typically, bank conflict is resolved by using shared_memory[threadIdx.y][threadIdx.x] instead of shared_memory[threadIdx.x][threadIdx.y]. After resolving bank conflicts, the run time is improved and it is reduced to 757.984 us. 25 % improvement over the original implementation.

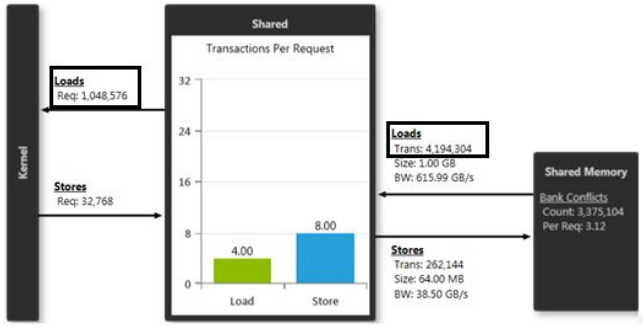

figure 5. *Shared memory implementation with bank conflict*

Experimental Results

The following microarchitectural analysis is for the shared memory implementation with bank conflict resolved.

Memory statistics: Global:

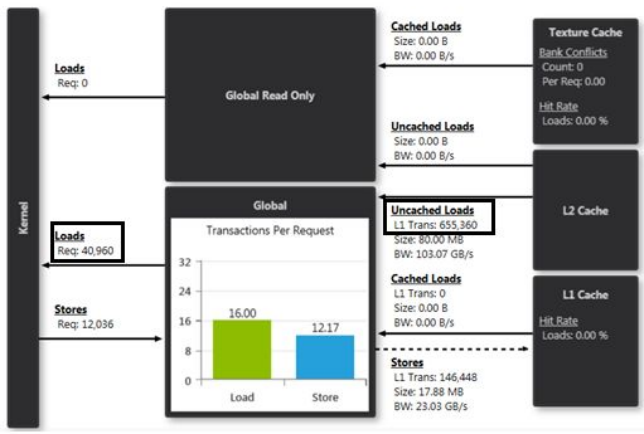

figure 6. Memory Statistics - Global

From Figure 6, It can be observed that the number of requests to global memory has reduced drastically (40,960 vs 129,024) improving the runtime. Figure 7 shows effective use of shared memory showing 1 to 1 correspondence between load and store.

Memory statistics: Shared:

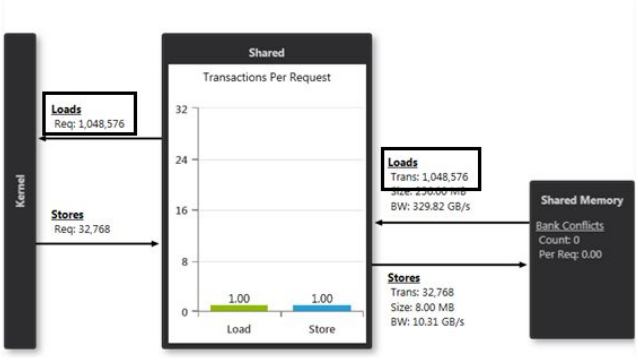

figure 7. Shared Memory Bank Conflict Resolved

Table 5: Results

Original implementation	1.000512ms
Shared memory implementation without nested for loop	0.928031ms
Shared memory implementation without nested for loop, bank conflict resolved.	0.757984ms

Closing and Future Work

General purpose GPU based solution to accelerate the algorithm is a niche area of research and development with respect to IQA algorithms. Still, they do not provide enough speedup to use the algorithms in real-time environment. That is why, it is essential to understand the underlying microarchitecture to map complex algorithms effectively onto the GPU. In this paper, the microarchitectural analysis of an implementation of the most apparent distortion (MAD) image quality assessment (IQA) algorithm is done, a bottleneck is strategically analyzed, and a solution is offered. Microarchitectural profiling of MAD implementation has showed that the local statistics computation is memory bandwidth limited. Hence, in order to improve memory bandwidth, frequent access to the global memory had to be reduced by exploiting the on-chip memory which offers low latency access.

In the original inefficient implementation, every thread runs nested for loops to bring in the data from global memory and store in local memory for local statistics computation. In the shared memory implementation, every thread brings in data from global memory corresponding to global id and stores in shared memory location corresponding to local thread id. The conclusion is, by increasing the amount of data reuse by the threads and by reducing high latency memory access to global memory, performance can be improved. We have demonstrated a promising shared memory implementation of the most problematic kernel with 25% improvement in the runtime. Individual kernel execution showed 1.33x speedup over the original implementation and since the kernel is called 40 times as part of the local statistics computation makes the speedup prominent thus improving the overall algorithmic runtime.

The application that is demonstrated does not involve any communication among the threads. If data must be communicated between the threads, necessary care must be taken to ensure race conditions do not occur. In this paper, only the mean calculation is taken into account and the performance is improved. It can be extended to kurtosis, standard deviation and skewness. It is expected to lead to much higher performance gains. Other performance limiters like compute resources, latency can be explored in detail not only for A5 but also with other kernels in the CUDA MAD implementation.

References

[1] Harris, Mark. "Optimizing parallel reduction in CUDA." NVIDIA Developer Technology 2, no. 4 (2007).

[2] Mohammadi, Pedram, Abbas Ebrahimi-Moghadam, and Shahram Shirani. "Subjective and objective quality assessment of image: A survey." arXiv preprint arXiv:1406.7799 (2014).

[3] Chandler, Damon M. "Seven challenges in image quality assessment: past, present, and future research." ISRN Signal Processing 2013 (2013).3.

[4] Phan, Thien D., Siddharth K. Shah, Damon M. Chandler, and Sohum Sohoni. "Microarchitectural analysis of image quality assessment algorithms." Journal of Electronic Imaging 23, no. 1 (2014): 013030-013030.

[5] Moorthy, Anush Krishna, and Alan Conrad Bovik. "Visual quality assessment algorithms: what does the future hold?." Multimedia Tools and Applications 51, no. 2 (2011): 675-696.

[6] Okarma, Krzysztof, and Przemysław Mazurek. "GPGPU based estimation of the combined video quality metric." In Image Processing and Communications Challenges 3, pp. 285-292. Springer Berlin Heidelberg, 2011.

[7] Holloway, J., Kannan, V., Chandler, D. M., & Sohoni, S. On the computational performance of single-GPU and Multi-GPU CUDA implementations of the MAD IQA Algorithm. Image Media Quality and its Applications 2016

[8] Wang, Zhou, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. "Image quality assessment: from error visibility to structural similarity." IEEE transactions on image processing 13, no. 4 (2004): 600-612..

[9] Larson, Eric C., and Damon M. Chandler. "Most apparent distortion: full-reference image quality assessment and the role of strategy." Journal of Electronic Imaging 19, no. 1 (2010): 011006-011006.

[10] Nsight, N. V. I. D. I. A., and Visual Studio Edition. "3.0 User Guide." NVIDIA Corporation (2013).

[11] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A unified graphics and computing architecture." IEEE micro 28, no. 2 (2008): 39-55.

[12] Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. "A performance study of general-purpose applications on graphics processors using CUDA." Journal of parallel and distributed computing 68, no. 10 (2008): 1370-1380.

[13] Kirk, David B., and W. Hwu Wen-mei. Programming massively parallel processors: a hands-on approach. Newnes, 2012.

[14] Ryoo, Shane, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73-82. ACM, 2008.

[15] Ryoo, Shane, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73-82. ACM, 2008.

[16] Hong, Sunpyo, and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness." In ACM SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 152-163. ACM, 2009.

[17] Rui, Ran, Hao Li, and Yi-Cheng Tu. "Join algorithms on GPUs: A revisit after seven years." In Big Data (Big Data), 2015 IEEE International Conference on, pp. 2541-2550. IEEE, 2015.

[18] Silverstein, D. Amnon, and Joyce E. Farrell. "The relationship between image fidelity and image quality." In Image Processing, 1996. Proceedings., International Conference on, vol. 1, pp. 881-884. IEEE, 1996.

[19] Said, Amir, and William A. Pearlman. "An image multiresolution representation for lossless and lossy compression." IEEE Transactions on image processing 5, no. 9 (1996): 1303-1310.

[20] Wu, Bo, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU." In ACM SIGPLAN Notices, vol. 48, no. 8, pp. 57-68. ACM, 2013.

[21] Brodtkorb, André R., Trond R. Hagen, Christian Schulz, and Geir Hasle. "GPU computing in discrete optimization. Part I: Introduction to the GPU." EURO journal on transportation and logistics 2, no. 1-2 (2013): 129-157.

[22] Micikevicius, Paulius. "Analysis-driven optimization." In GPU technology conference, pp. 1-55. 2010.

[23] Wang, Zhou, Eero P. Simoncelli, and Alan C. Bovik. "Multiscale structural similarity for image quality assessment." In Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on, vol. 2, pp. 1398-1402. Ieee, 2003.

[24] Wang, Zhou, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. "Image quality assessment: from error visibility to structural similarity." IEEE transactions on image processing 13, no. 4 (2004): 600-612.

[25] Documentation, CUDA Toolkit. "v6. 0." Santa Clara (CA, USA): NVIDIA Corporation (2014).

Author Biography

Vignesh Kannan *received the Bachelors in Electronics and Communication engineering from SASTRA University, India (2010). He has 4 years of industrial experience working on 4G technology. Currently, he is a Masters student in Software Engineering at the Arizona State University, Tempe, USA. His interests are in GPGPU programming and Computer Architecture.*

Joshua Holloway *received his B.S. in Electrical Engineering from Oklahoma State University (2013). Joshua is currently a member of the Parallel Systems and Computing Laboratory at Arizona State University where he is pursuing a PhD in Computer Engineering in the School of Computing, Informatics, and Decision Systems Engineering. His research interests include digital signal and image processing, computer vision, heterogeneous parallel programming, and has recently begun working on efficient hardware mapping of complex algorithms.*

Sohum Sohoni *received the B.E. degree in Electrical Engineering from Pune University, India, in 1998 and a PhD in Computer Engineering from the University of Cincinnati, Cincinnati, Ohio, in 2004. He is currently an Assistant Professor in The Polytechnic School in the Ira A. Fulton Schools of Engineering at Arizona State University. Prior to joining ASU, he was an Assistant Professor at Oklahoma State University. His research interests are broadly in the areas of engineering and computer science education, and computer architecture. He has published in the International Journal of Engineering Education, Advances in Engineering Education, and in ACM SIGMETRICS and IEEE Transactions on Computers.*

Damon M. Chandler *received the B.S. in Biomedical Engineering from The Johns Hopkins University (1998); and the M.Eng., M.S., and Ph.D. in Electrical Engineering from Cornell University (2000, 2004, 2005). From 2005-2006, he was a postdoc in the Department of Psychology at Cornell. From 2006-2015, he was on the faculty at Oklahoma State University. He is currently an Associate Professor at Shizuoka University, where his research focuses on modeling properties of human vision. He is as an Associate Editor for the IEEE TIP and the Journal of Electronic Imaging.*