An Analysis of the Memory Bottleneck and Cache Performance of Most Apparent Distortion Image Quality Assessment Algorithm on GPU

by

Vignesh Kannan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2016 by the
Graduate Supervisory Committee:

Sohum Sohoni, Chair
Fengbo Ren
Mohamed Sayeed

ARIZONA STATE UNIVERSITY

December 2016

ABSTRACT

As digital images are transmitted over the network or stored on a disk, image processing is done as part of the standard for efficient storage and bandwidth. This causes some amount of distortion or artifacts in the image which demands the need for quality assessment. Subjective image quality assessment is expensive, time consuming and influenced by the subject's perception. Hence, there is a need for developing mathematical models that are capable of predicting the quality evaluation. With the advent of the information era and an exponential growth in image/video generation and consumption, the requirement for automated quality assessment has become mandatory to assess the degradation. The last few decades have seen research on automated image quality assessment (IQA) algorithms gaining prominence. However, the focus has been on achieving better predication accuracy, and not on improving computational performance. As a result, existing serial implementations require a lot of time in processing a single frame. In the last 5 years, research on general-purpose graphic processing unit (GPGPU) based image quality assessment (IQA) algorithm implementation has shown promising results for single images. Still, the implementations are not efficient enough for deployment in real world applications, especially for live videos at high resolution. Hence, in this thesis, it is proposed that microarchitecture-conscious coding on a graphics processing unit (GPU) combined with detailed understanding of the image quality assessment (IQA) algorithm can result in non-trivial speedups without compromising quality prediction accuracy. This document focusses on the microarchitectural analysis of the most apparent distortion (MAD) algorithm. The results are analyzed in-depth and one of the major bottlenecks is identified. With the knowledge of underlying microarchitecture, the implementation is restructured thereby resolving the bottleneck and improving the performance.

DEDICATION

I would like to dedicate this thesis to the beginning – Mom. You deserve a Nobel Prize for bringing me up. Thank you dad for letting me pursue my dreams and my sister Aishu for being a good sparring partner. Akkshaya, I cannot thank you enough for generously offering me free food. I am grateful to many persons who shared their memories and experiences.

ACKNOWLEDGMENTS

I would like to thank the amazingly talented professors I have worked with on the project that resulted in the work presented in this thesis. First and foremost, I would like to thank Dr Sohum Sohoni for giving me this opportunity to work on this project. He taught me a very important lesson - feeling of appreciation. I need not launch a rocket to feel accomplished, even a small skill can be reassuring. His technical advice was essential to the completion of this research and has taught me valuable lessons and insights on the workings of academic research in general. My thanks and appreciation to Josh for persevering with me as my mentor throughout the time it took me to complete this research and write the dissertation. I will never forget the all-nighter before my defense. The members of my dissertation committee, Dr Fengo Ren and Dr Mohamed Sayeed, have generously given their expertise to better my work. I thank them for their contribution and their good-natured support. I must acknowledge all my colleagues, students, and teachers who assisted, advised, and supported my research and writing efforts over the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Image quality is a subjective measure of how precise an image of a subject represents that subject. It is usually inferred by the preference of one image over another (Silverstein, D. A., & Farrell, J. E., 2006). Digital images are rapidly becoming part of our daily lives in the form of photos and videos of different resolution (Mohammadi, P., Ebrahimi-Moghadam, A., & Shirani, S., 2014). These images are often subjected to several processing stages such as acquisition, compression, and transmission before they reach their end-users. The images can suffer from different types of distortions through each of the above-mentioned stages, which degrade their quality (Pitas, I, 2014).

In image compression stage, it is not always possible to use lossless compression, as it cannot guarantee compression for all input datasets (Said, A., & Pearlman, W. A., 1996), lossy compression schemes introduce blurring and ringing effects, leading in quality degradation (Mohammadi, P et al., 2014). In order to maintain, control, and enhance the quality of images, it is essential for image acquisition and processing systems to assess the quality of images at each stage. Unless lossless compression is used, compressing and decompressing the image results in data loss, which affects the image quality factors such as sharpness, color accuracy, and contrast (Goldmark, P. C., & Dyer, J. N., 1940). Hence, it is critical to analyze the impact of the effects caused by distortion on image's visual quality.

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 (Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P., 2004). Hence, there is a need for automated quality prediction. In order to fulfil this requirement, objective image quality assessment (IQA) was introduced to develop methods that can predict perceived image quality automatically. As highlighted in this survey (Chandler, D. M., 2013), a better human visual system (HVS) modelling can lead to development of IQA algorithms with higher quality prediction accuracy and greater robustness for changing visual signals.

Historically, IQA algorithms have been used in areas such as bio-medical (Barrett, H. H., 1990), remote sensing (Buiten, H. J., & Van Putten, B., 1997), and social media (Bauer, M. W., & Gaskell, G., 2000). In case of distortions like noise, it is assumed that there is a direct relationship between the noise detectability and the perceived image quality. This assumption makes it possible to apply human contrast detection models to perform predictions about image quality (Silverstein, D. A., & Farrell, J. E., 1996).

With the advent of information era in the late 1970s, image quality assessment algorithms have recently found a place in calculating the visual quality index, ranging from applications such as standard image compression (Zhang, L., Zhang, L., Mou, X., & Zhang, D., 2011) to areas like Computer vision (Brosnan, T., & Sun, D. W., 2004), Visual psychophysics (Wang, Z., Lu, L., & Bovik, A. C., 2004), and Machine learning (Suresh, S., Babu, R. V., & Kim, H. J., 2009).

There are 2 classes of objective quality algorithms (Pappas, T. N., Safranek, R. J., & Chen, J., 2000; Kunt, M., & van den Branden Lambrecht, C., 1998); one involves mathematically defined measures such as signal-to-noise ratio (SNR), mean absolute error (MAE), root mean squared error (RMSE), and mean squared error (MSE) whereas the other method takes human visual system (HVS) properties into consideration to integrate perceptual quality measures. Initial research on IQA algorithms focused only on prediction fidelity with less importance to algorithmic, run-time and microarchitectural complexity (Chandler, D. M.,2013; Phan, T. D., Shah, S. K., Chandler, D. M., & Sohoni, S., 2014; Moorthy, A. K., &

Bovik, A. C., 2011). When IQA algorithms march into production scenarios, the runtime performance and related computational considerations become as important as the prediction accuracy.

Section 1: **Background on Image/Video Quality Assessment Algorithms**

Image quality assessment algorithms evaluate the visual quality of an image subject to be viewed by humans. Employing human observers to evaluate the image quality is time consuming and less economic compared to automatic evaluation using IQA algorithms. In addition, this is a subjective topic, as different individuals can perceive the same image differently. Hence automatic image quality assessment algorithms has gained prominence. Image quality assessment algorithms are classified into three categories based on the availability of the ideal reference image (Lundström, C., 2006): Full-reference, No-reference and Reduced-reference.

IQA algorithms normally include a two stage structure. The first stage involves local quality measurement through frequency based decomposition of the images (George, A., & Livingston, S. J., 2013) (i.e.) performing a color space transformation to obtain de-correlated color coordinates, decomposing these new coordinates into perceptual channels (Charrier, C., Knoblauch, K., Moorthy, A. K., Bovik, A. C., & Maloney, L. T., 2010). The second stage involves calculating the quality value through statistical computations. (i.e.) an error is estimated for each of the channels from step 1 and final quality scores are obtained by pooling these errors in the spatial and/or frequency domain (George, A., & Livingston, S. J., 2013; Charrier, C et al., 2010).

Full-reference image quality assessment technique compares an undistorted reference and a distorted/test image whose quality needs to be determined, and predicts the quality of the test image as a scalar value. The quality is usually measured as the impairment from the ideal (Lundström, C., 2006). The reference image usually requires more resources than the distorted image and hence FR-IQA is used to design algorithms for in-lab testing. Typical application areas include image compression (Charrier, C et al., 2010), watermarking (Huang, J., & Shi, Y. Q., 1998), image acquisition (infrared) (Lee, Y. H., Khalil-Hani, M., Bakhteri, R., & Nambiar, V. P., 2016), and others such as television (Lundström, C., 2006).

In case of photography, there is no reference image (Lundström, C., 2006) and so, a no-reference IQA algorithm is needed where blind image quality prediction will be carried out (Kamble, V., & Bhurchandi, K. M., 2015). No-reference quality assessment models that can operate without knowledge of distortion types, reference image, and human opinion scores are of great interest recently (Mittal, A., Soundararajan, R., Muralidhar, G. S., Bovik, A. C., & Ghosh, J., 2013).

A third alternative is one in which only a part of the reference image is available in the form of a set of extracted features (Lundström, C., 2006). This is widely used in satellite (Nickolls, J., 2007) and remote sensing (Buiten, H. J., & Van Putten, B., 1997).

Runtime Performance of IQA Algorithms:

The capabilities of handheld devices have increased from simple telephony to smartphones or tablets capable of capturing, storing, sending and displaying photos. In addition to that, smartphones can support video streaming services like Netflix and YouTube. With the surge in image consumption, there is a need to evaluate and improve the runtime performance of IQA algorithms to be used in real-time. As an example, an IQA algorithm with high predictive performance, requires an execution time of order of seconds for a single image (Phan, T., Sohoni, S., Chandler, D. M., & Larson, E. C., 2012).

In order to improve the runtime performance, understanding where the bottlenecks are through performance analysis (Jain, R. K., 1990; Zhao, L., Iyer, R., Makineni, S., & Bhuyan, L., 2005) is important. Phan et al (Phan, T. D et al., 2014) performed microarchitectural hotspot analysis of image quality assessment algorithms on CPU, which showed most of the algorithms, have bottleneck related to memory hierarchy and execution/computation and so, the authors proposed microarchitecture conscious coding techniques for optimization.

In general, two methods are common for improving the computational complexity: Algorithm based techniques and underlying hardware based techniques.

Software/algorithmic techniques such as, accelerating discrete cosine transform (DCT), which forms the essence of most of the IQA algorithms by using variations of Fast Fourier transform (FFT) is is one example (Chen, W. H., Smith, C. H., & Fralick, S. C., 1977). Hardware based acceleration techniques based on (graphic processing units) GPU (Okarma, K., & Mazurek, P., 2011) and field programmable gate array (FPGA) implementations (Alam, S. R et al., 2007) have also yielded notable speedups as described below. GPUs can accelerate quality assessment algorithms because their algorithmic models attempt to imitate a collection of visual neurons, which operate in a massively independent and parallel fashion yielding data-level parallelism.

A naive implementation would map each pixel of the image to a thread running on the GPU (Holloway, J., Kannan, V., Chandler, D. M., & Sohoni, 2016). General purpose GPU (GPGPU) techniques for speeding up structural similarity (SSIM) (Wang, Z. et al., 2004), multiscale-structural similarity (MS-SSIM) (Wang, Z., Simoncelli, E. P., & Bovik, A. C., 2003), and combined video quality metric (CVQM) are described in (Okarma, K., & Mazurek, P., 2011). The authors used NVIDIA's CUDA programming language, which accelerates the computation by distributing the workload over its massive GPU cores. Their corresponding GPU implementation yielded 150X and 35X speedups for SSIM and MS-SIM respectively.

Holloway et al discussed (Holloway, J et al., 2016) a GPU based acceleration for most apparent distortion (MAD) (Larson, E. C., & Chandler, D. M., 2010) exploiting massive parallelism of the GPU. Each pixel is mapped to a single GPU core. GPU implementation showed 24x speedup and multi-GPU implementation (a single instance of the algorithm executing on multiple GPUs at the same time) yielded 33x speedup over the baseline CPU implementation.

Thus, a new research area where the microarchitectural analysis of image quality assessment algorithms on a GPU is introduced in this thesis. This analysis will ensure that IQA algorithm designers keep parallelism in mind while creating new IQA algorithms. They are not limited by serial instruction execution anymore.

Section 2: **General Purpose GPU Computing Overview**

Historically, increasing the frequency, because of transistor shrinking was prominent in increasing the performance of processors (Moore, G. E., 2006). However, this trend ended a decade ago due to memory wall. Also, power wall (the chip's overall temperature and power consumption), forced the semiconductor industry to stop pushing clock frequencies much further (Hennessy, J. L., & Patterson, D. A., 2011). As Moore's law prevailed, but frequency scaling reached its physical limits, there was a major shift in the microprocessor industry towards multicore processors and parallel computing (Gepner, P., & Kowalik, M. F., 2006). Graphic processors are a special case of multi-core processors. In reality, a GPU is an army of cores used for graphics rendering and shading, as these functions are commonly used, very specific and compute-intensive, and too expensive for the (central processing unit). Furthermore, the tasks of rendering and shading are extremely data-parallel, which the CPU was not designed to exploit well.

GPUs excel at fine-grained data-parallel tasks comprising of thousands of independent threads executing graphic operations like vertex, geometry and pixel shader programs continuously (Nickolls, J., Buck, I., Garland, M., & Skadron, K., 2008). NVIDIA introduced the GeForce 8800 GPU that replaced the traditional dedicated hardware per processing stage (Vertex, Triangle, Pixel, Raster Operation and Memory) with a unified shader processor (Nickolls, J., 2007). However, with the massive parallelism that GPUs provide, researchers have moved from 3D graphics towards general-purpose computation (GPGPU) such as encryption/decryption (Manavski, S. A., 2007; Garland, M., 2008). Though using GPUs for non-graphical computations yielded impressive results (Nickolls, J et al., 2008), the many limitations of doing GPGPU computation through graphics APIs are well-known (T. Dokken, T.R. Hagen, and J.M. Hjelmervik., 2005). Before the introduction of general-purpose languages for GPU computing, OpenGL and DirectX were used to program the GPU. The syntax and the need to program in terms of graphics API's made them difficult for programmers (Hill, F., & Kelley, S., 2007).

NVIDIA introduced CUDA (Nvidia, C. U. D. A., 2011) which allowed programmers to perform general-purpose computation on GPUs. Although GPUs provides massive parallelism with its hardware resources, the onus is on the programmer to map the code effectively for optimal performance. Since its inception, GPGPU has found a prominent spot in applications such as DNA sequencing (Trapnell, C., & Schatz, M. C., 2009), weather prediction (Michalakes, J., & Vachharajani, M., 2008), cryptography (Manavski, S. A., 2007), and databases (Rui, R., Li, H., & Tu, Y. C., 2015).

a) Overview of CUDA Programming structure:

This subsection gives a brief overview of NVIDIA's CUDA programming language and discusses the key components (Nvidia, C. U. D. A., 2014). A CUDA program comprises of host CPU code and device GPU code. Device GPU code is launched as kernels by CPU. Host code locally runs on the CPU. The functions that execute on the GPU are called kernels and a kernel is launched with a configuration of grid blocks (number of blocks in the grid) and thread blocks (number of threads in a block). When a kernel function is encountered during program execution, CPU sends appropriate commands to invoke a kernel on the GPU. GPU kernels can execute independently of CPU execution. The GPU executes one grid at a time, which results in execution of many threads. The host can continue its execution without waiting for the completion of the device kernel.

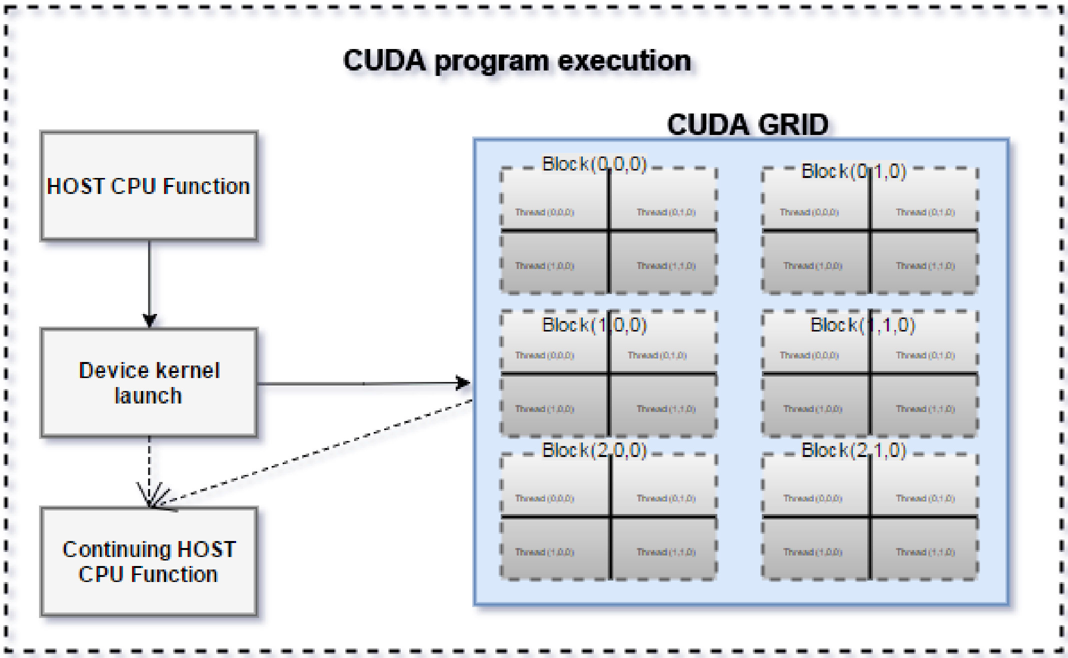


Figure 1. Overview of CUDA Program Execution

Kernel:

- A kernel code is executable only on a device
- It can be called either by device (dynamic parallelism (Jones, S., 2012)or host)
- It is defined by __global__ qualifier and does not return any value.

Host:

Host code (Nvidia, C. U. D. A., 2011) does the following operations (pertaining to this research)

- Select a device if there are many GPUs connected to the system.
- Initialize the GPU
- Allocate memory on the GPU (cudaMalloc)
- Transfer data from host to device (cudaMemcpy)
- Start profiler

- Invoke a GPU kernel
- Stop profiler
- Transfer data from device to host as needed
- Deallocate the GPU memory
- Reset the device.

When a kernel is launched, every thread within the grid configuration executes an instance of the kernel. This inherently provides the scalability needed for the application. CUDA does not guarantee an order of execution of the launched kernels (Nickolls, J et al., 2008). However, CUDA guarantees that all the threads within a thread block are executed at once on the same streaming multiprocessor (SM). For instance, a kernel launched as below guarantees that, a block of (16x16) threads will execute on the same SM, but the order of blocks within 32x32 grid is random.

Kernel <<< (32, 32, 1), (16, 16, 1) >>> (argument);

Multiple thread blocks can reside in an SM, however a single thread block cannot be shared across SMs. Thread blocks can

- Share memory
- Synchronize
- Communicate (co-operate)

CUDA provides block level and thread level data parallelism (different blocks/threads can act on different data). Threads within a block are launched in-group of 32 threads called warps, which the multiprocessor uses for scheduling (Nickolls, J et al., 2008). Threads in the same warp share the program counter (Fung, W. W., & Aamodt, T. M., 2011). This programming model is different from (SIMD) manner because, every thread (not data) is mapped to the processor and executed in SIMD style and hence single instruction multiple thread (SIMT). Every thread executes same instruction, and possibly on different data. Within a thread block, all threads have the same life cycle. Warps are initiated, dispatched, swapped out/in from/to an SM at the same time. Context of all the running threads are stored in warp pools at the respective SMs. At every cycle, the hardware warp scheduler selects a warp that does not stall due to factors such as cache misses, global memory request or pipeline hazard from the pool for execution. Whenever a warp stalls, GPUs are quickly able to context-switch in order to hide the execution latency with minimal penalty.

On a kernel launch, the driver notifies the GPU's work distributor of the kernels' starting program counter and its grid configuration. As soon as an SM has sufficient resources, the scheduler randomly assigns a new thread block and the SM's controller initializes the state for all threads in that thread block.

Loops and conditional statements are allowed in kernel code, but if different threads in the same warp follow different branches (warp divergence), then the SM will automatically serialize or stall execution until the threads resynchronize thus reducing effective parallelism. Each multiprocessor has a fixed number of registers for each core so the number of threads running simultaneously depends on the number of registers. In addition to that, processor occupancy (Nvidia, C. U. D. A., 2014), maximum number of concurrent warps, maximum number of concurrent thread blocks all depend on this count.

b) Overview of Kepler GPU Architecture:

This subsection gives a brief overview of NVIDIA TESLA K40 GPU accelerator (Nvidia, C., 2012), which is the main component in this study. The K in K40 stands for Kepler which is the codename for a GPU microarchitecture developed by Nvidia.

The GPU is connected to the host through a PCI-Express bus in current high performance systems. Data is transferred from the GPU to CPU and vice versa either through DMA or by unified-memory programming which is available with restrictions (NVidia, C. U. D. A., 2014). Data is transferred across the PCI-Express bus at the rate of 32GB/s. The K40X GPU (NVidia, C., 2012) consists of a GK110b processor equipped with 12 GB of GDDR5 memory, 15 streaming multiprocessors (SM) (Wittenbrink, C. M., Kilgariff, E., & Prabhu, A., 2011), each of the SM's consists of 192 CUDA cores clocked at 745 MHz, achieving 5.12 TFLOPS in single-precision peak performance.

Figure 2. NVIDIA Kepler GK110 Internal. Adapted from NVIDIA-Kepler-GK110-Architecture-Whitepaper (NVidia, C., 2012)

Every SM has an on-chip memory area of 64 KB that can be configured as shared memory or as L1 cache. It also has 65536 32-bit registers per SM (Wittenbrink, C. M et al., 2011). In addition, the GK110b processor is equipped with a read-only cache of 48 KB per SM, which can double as texture cache. To sample or filter image data, the GPU's Texture units are ideal. Every SM has 16 Tex units. Furthermore, all 15SMs share the 1.5 MB L2 cache. On Kepler device, the configuration of on-chip memory can be 16KB L1/48KB shared or 32KB L1/32KB shared or 48KB L1/16KB shared memory.

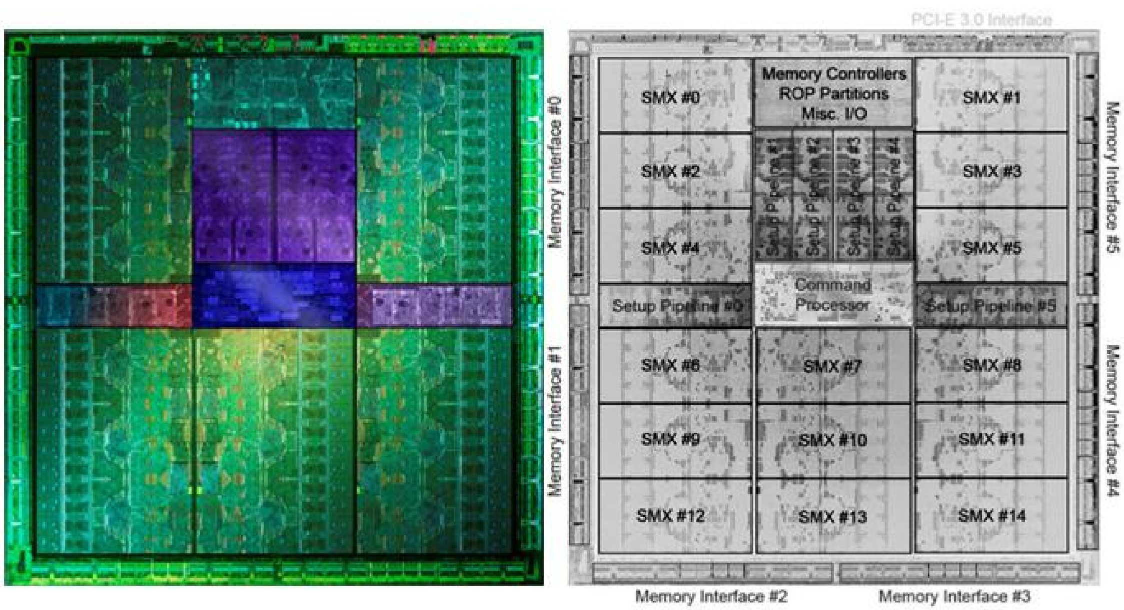


Figure 3. GPU Die-Diagram of Tesla K40. Adapted from ("http://www.guru3d.com/articles-pages/geforce-gtx-780-ti-review,3.html", n.d.).

Shared memory is faster than global and local memory because it is on-chip. Shared memory latency is roughly 100x lower than uncached global memory latency (Harris, M., 2007). Usually, Shared memory is allocated per thread block (it is common to all the threads executing in an SM), so all threads in the block have access to the same-shared memory. This can be viewed as interaction among the threads in the same block as threads can access data in the configured shared memory loaded by other threads from global memory within the same thread block. The programmer must provide necessary synchronization before exploiting this functionality; otherwise, it may lead to race conditions. This user-managed memory can be used in high-performance cooperative parallel algorithms such as mean reduction, and to enable global memory coalescing (Hong, S., & Kim, H., 2009) in cases where it would otherwise be prohibitive.

Tesla K40 (GK 110b) is capable of routing read-only data through the same cache used by texture pipeline. If the incoming data from global memory is read-only, this cache is initialized automatically and used (Nvidia, C. U. D. A., 2014). Data loaded through the read-only cache can be accessed in a non-uniform pattern as well. Programmatically, *const* and __*restrict*__ qualifiers ensures the data is read-only when used while declaring a variable in CUDA program. __ldg() intrinsic can also be employed to ensure this operation if more explicit control is desired.

Constant memory can be used if the data is to be broadcasted over to all the threads in a warp. It can be accessed through 8KB cache on each SM backed by 64KB partition of the global memory. If all the threads in the warp request the same value, that value is broadcasted to all threads in a single cycle otherwise, if the threads in a warp request M different values, the requests are serialized and take M clock cycles.

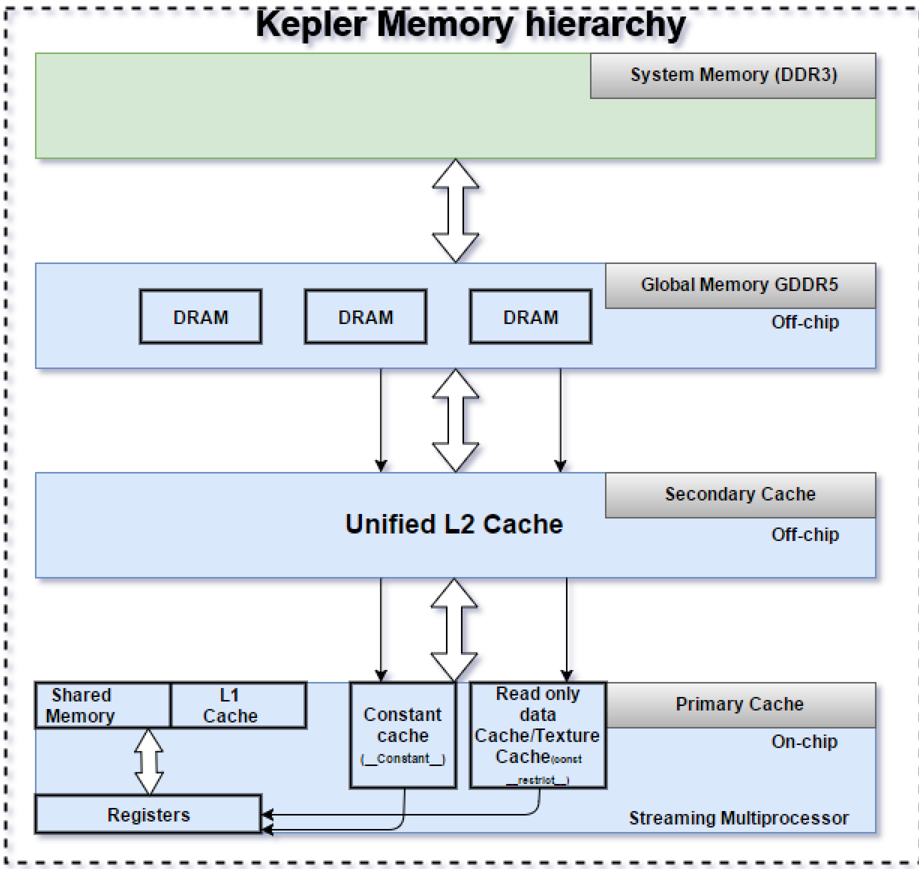


Figure 4. Kepler Memory Hierarchy

The off-chip GDDR5 memory handles each memory request to CUDA global memory. Kepler (Datta, K., Murphy et al., 2008) follows the same coalescing rule (Davidson, J. W., & Jinturkar, S., 1994) as Fermi. However, a significant change from Fermi is that, L1 cache in Kepler is used for stack data and register spilling only and so, global memory loads and stores are not cached in the L1 cache by default (Nsight, N. V. I. D. I. A., & Edition, V. S., 2013; Nvidia, C. U. D. A., 2007), whereas on Fermi load accesses are cached by default (Nvidia, C. U. D. A., 2014). This can be attributed to the fact that, Kepler GPUs are most suited for general purpose scientific computing and the GPU designers expect the programmers to hack into the GPU – the programmers should be capable of exploiting/utilizing the microarchitectural features provided effectively. On the other hand, Fermi architecture is most suited for graphics processing where the programmers are not expected to know the underlying microarchitectural features and thus the GPU should be capable of utilizing its resources optimally. Access to global memory in Kepler has very high latency (200-400 cycles), but GPUs hide this latency by switching between warps as they stall (Nvidia, C. U. D. A., 2011).

Applications that do not automatically employ shared memory benefit from the L1 cache, improving the performance with minimum effort (Glaskowsky, P. N., 2009). On Kepler however, the same implementation, would not perform as efficiently since global memory loads are not cached. Both shared and read-only cache can be utilized on Kepler only after explicit code modifications.

Figure 5. Streaming Multiprocessor Internal in GK110. Adapted from ("http://www.guru3d.com/articles-pages/geforce-gtx-780-ti-review,3.html," n.d.).

Each streaming multiprocessor contains four warp schedulers with two instruction dispatch units each, allowing concurrent operation/execution of four warps. These four warp schedulers select two independent instructions per warp to dispatch each cycle. A warp making a global memory access is stalled and GPUs are able to hide the stall latency by context switching (Jog, A. et al., 2013) to execute instructions from another warp for better resource utilization. Every SM also contains (special functional units) SFU units for fast approximate transcendental operations such as sine, cos, exp. The Cores, Load/Store units and Special Function Units (SFU) are pipelined units. They maintain - in various stages of completion, the results of many computations/operations at the same time. Hence, in one cycle they can accept a new operation and yield the results of another operation that was initiated several cycles ago. Latency is the number of clock cycles a warp takes to be ready to execute its next instruction, and warp schedulers makes sure they have some instruction to issue at every clock cycle to hide the latency (Nvidia, C. U. D. A., 2011).

c) Overview of Program Compilation:

Our project consists of two source files kernel.cu and main.cpp. When compilation is triggered, the header files (#include) are expanded. Primarily, the .cu file gets processed using cudafe and nvopencc (open source compiler provided by NVIDIA based on open64) (Bakhoda, A et al., 2009; Developers, O., 2001) into intermediate .ptx pseudo-assembly (Nvidia, C. U. D. A., 2014). The ptx assembler (ptxas) assembles the ptx file into native CUDA binary (cubin.bin).

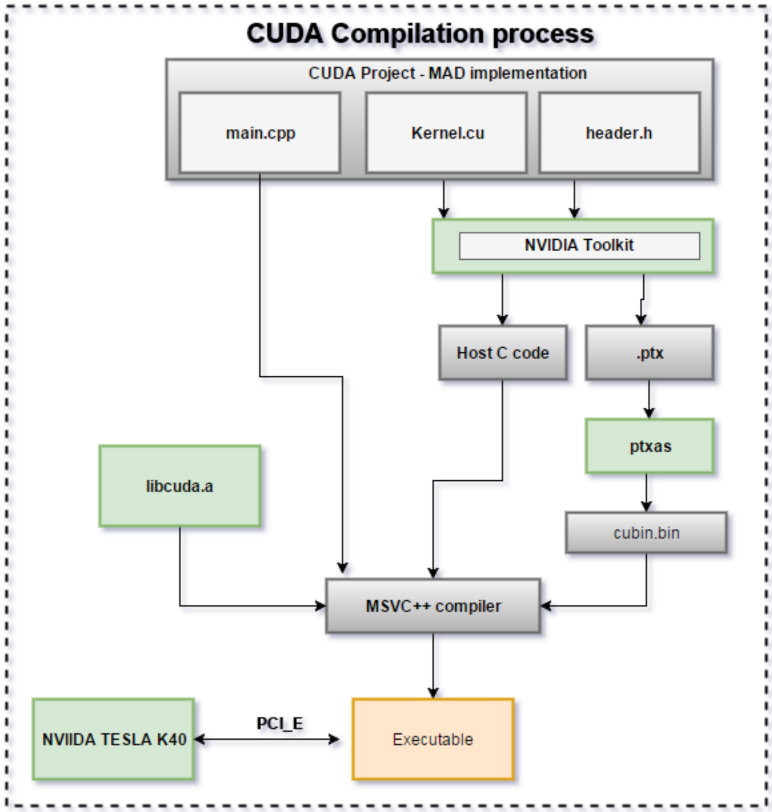


Figure 6. CUDA Compilation Process

The cubin binary is then merged with the host C++ code and compiled into a single executable file to be linked with the CUDA Runtime (application programming interface) API library (libcuda.a). Finally, the executable then calls the CUDA Runtime API in order to initialize and invoke compute kernels onto the GPU through NVIDIA CUDA driver.

Section 3: Related Work

In this section, we talk about the achievements made in recent years to exploit the underlying microarchitecture of the GPU. Shared memory computation on a GPU has been one of the extensively studied types of computations because of shared memory reuse and bandwidth. 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. (Che, S et al., 2008) 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 GPUs bottleneck. The authors brought about the need to find efficient mappings of their applications' data structure to CUDA's domain based model for better efficiency which forms motivation for this thesis.

Harris (Harris, M., 2007) has exhibited the efficiency in using shared memory for computation. The paper discusses 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 proposes the idea of using a sliding window across the shared memory as well as the need to avoid bank conflicts. Overall, the paper displays a speedup of 30X over the naïve implementation. This thesis takes inspiration from the shared memory implementation (Harris, M., 2007) to resolve the memory bottleneck as described in Chapter 3.

Tuning strategies to improve performance, such as coalescing, prefetching, unrolling, and occupancy maximization are introduced in classical CUDA textbooks (Kirk, D. B., & Wen-mei, W. H., 2012). In (Ryoo, S. et al., 2008) the authors not only discuss the different tuning strategies, but also show how optimum usage of hardware resources is critical for occupancy and performance. However, the entire study has been focused on a pre-Fermi architecture. An analytical performance model (Hong, S., & Kim, H., 2009) provides 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 (Rui, R., et al., 2015), which is the model adopted in this thesis.

It is common to observe irregular memory accesses on the GPU. Wu, B.et al (Wu, B.et al., 2013) discuss reorganizing data to minimize non-coalesced memory access. Brodtkorb et al (Brodtkorb, A. R., Hagen, T. R., Schulz, C., & Hasle, G., 2013) give a detailed picture on profile driven development, stressing the importance on iterative programming and optimization. The authors go into detail about using the NVIDIA profiler to profile the implementation and by using the data, improving a local search. Micikevicius, (Micikevicius, P., 2010) has discussed profiler driven analysis and optimization. The author has asserted the importance of Memory bandwidth, optimum utilization of compute resources, instruction, and memory latency and provides a note on the essential profiling parameters to consider and possible conclusions to be drawn from the data. The microarchitectural analysis performed in this thesis as mentioned in Chapter 2 profiles the most problematic kernel with respect to all the parameters mentioned above.

In a paper (Xu, C., Kirk, S. R., & Jenkins, S., 2009) by Xu, Chang et al, both thread level and block level tiling are noted and goes in detail about using tiles of specific size. There are many other research papers about microarchitectural analysis of parallel implementations in areas like

a) Cryptography (Manavski, S. A., 2007) where how optimizing the number of thread blocks, constant memory, and shared memory accelerates the application.

b) Matrix multiplication (Ryoo, S et al., 2008) by increasing the number of warps, redistributing work across threads and thread blocks and inter thread parallelism.

However, 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.

CHAPTER 2

ALGORITHM AND ANALYSIS

Section 1: Methodology

Application domain:

The current CUDA MAD implementation reads both reference and distorted test images from file, and processes the images on the GPU. To avoid frequent data transfer across the PCI bus, both the images are copied from CPU to the GPU before launching the kernel so that subsequent kernel operations process the images from the global memory. For analysis purpose, the kernel with most runtime and less achieved occupancy will be selected and microarchitectural analysis will be performed. At the end of the analysis, a major bottleneck that is resulting in the poor performance of the kernel will be dealt with thus improving the runtime and providing insight into the behavior of the GPU.

The GPU version of MAD was developed using NVIDIA's CUDA API and the CPU portion of the code uses C++. A GPU Profiling of the implementation is performed using NVIDIA Nsight profiler, NVIDIA Visual Profiler. Given the same input dataset, times are measured right after initial setup (e.g., after file I/O) and includes the time required to transfer data between the disjoint CPU and GPU memory spaces.

The experiment:

The experiment involves two phases. Phase 1 performs microarchitectural analysis of the current MAD implementation and phase 2 resolves the bottleneck observed in phase1.

1) NVIDIA NVVP ranks the kernels based on their execution time and achieved occupancy; Occupancy is the ratio of available active warps per cycle to the maximum number of warps that can be executed on a processor. Therefore, the first

section performs microarchitectural analysis of the kernel, which lists the bottlenecks hindering the performance.

2) From the microarchitectural analysis of the kernel listed in the above section, details on the bottlenecks are derived. From that analysis data, the lines of code that are causing the issue are identified and the possible workarounds specified in the literature are applied. The impact of these modifications is measured to test the effectiveness of the changes in the context of MAD.

Experimental Setup:

The details of the overall test system are shown in Table 1.

Table 1

Test System Configuration

	Test system
CPU	Intel® Xeon® Processor E5-1620 @ 3.70 GHz Cores: 4 cores (8 threads)
RAM	RAM: 24GB DDR3@1866 MHz(dual channel)
OS	Windows 7 64-bit
Compiler	Visual Studio 2013 64-bit;
GPU1	NVIDIA Tesla K40(PCIe 3.0)
GPU2	NVIDIA NVS 310 (PCIe 3.0)

The experiment was conducted on a system setup with an Intel CPU and an NVIDIA GPU: a single-socket machine with 24GB of main memory and a hyper threaded Intel Xeon quad-core processor, running at 3.70 GHz with cache configuration in Table 2.

Table 2

Cache Description on Xeon E5-1620

Cache description	**Size**	**Description**
L1 D-Cache	32KB x 4	8-way set associative, 64-byte line size
L1 I-Cache	32KB x 4	8-way set associative, 64-byte line size
L2 Cache	256KB x 4	8-way set associative, 64-byte line size
L3 Cache	10 MB	20-way set associative, 64-byte line size

An NVIDIA Tesla K40 GPU with NVIDIA driver version 10.18.13.5390 and CUDA version 7.5 is used for the experiment. Table 3 gives information on the GPU.

Table 3

Tesla K40 GPU Description

Total amount of global memory	11520 MBytes (12079398912 bytes)
(15) Streaming Multiprocessors, (192) CUDA Cores/MP	2880 CUDA Cores
GPU Max Clock rate	745 MHz (0.75 GHz)
Memory Clock rate	3004 MHz
Memory Bus Width	384-bit (6 x 64 bit memory controller)
L2 Cache Size	1572864 bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32

Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Base Core Clock-Rate	889 MHz
Computational Throughput	5121 GFLOPS Single Precision 1707 GFLOPS Double Precision

Section 2: Most Apparent Distortion Algorithm

Most apparent distortion IQA algorithm is selected because it is currently the best predictive performance IQA algorithm. However, MAD employs relatively extensive perceptual modeling, which imposes a large runtime that prohibits its widespread adoption into real-time applications. MAD takes as input a distorted image and a reference version of the same image. There are two different stages in the algorithm, called the detection stage and the appearance stage, which are independent of each other until the final calculation of the quality score which is done on the CPU after both stages are complete. The detection stage analyzes high quality images with near-threshold distortions; the appearance stage analyzes low quality images with supra-threshold distortions.

The detection stage, represented in the top portion of Figure 7, first performs a series of preprocessing steps on both images. The images are converted to perceived luminance and then each filtered with a contrast sensitivity function (CSF) filter kernel. After filtering the images, the rms contrast images are fed through other stages to extract local statistics and compared to create a visibility difference map between the two processed images. In the appearance stage, represented in the lower portion of the Figure 7, each image is first spectrally decomposed into 20 log-Gabor sub bands (5 scales and 4 orientations) via a filter bank. The local statistics (standard deviation, skewness and kurtosis) are then extracted from each individual sub band. The detection and appearance difference maps are each collapsed into a scalar quantity with a Euclidean 2-norm. The two resulting scalar values are then combined into a final quality score via a weighted geometric mean.

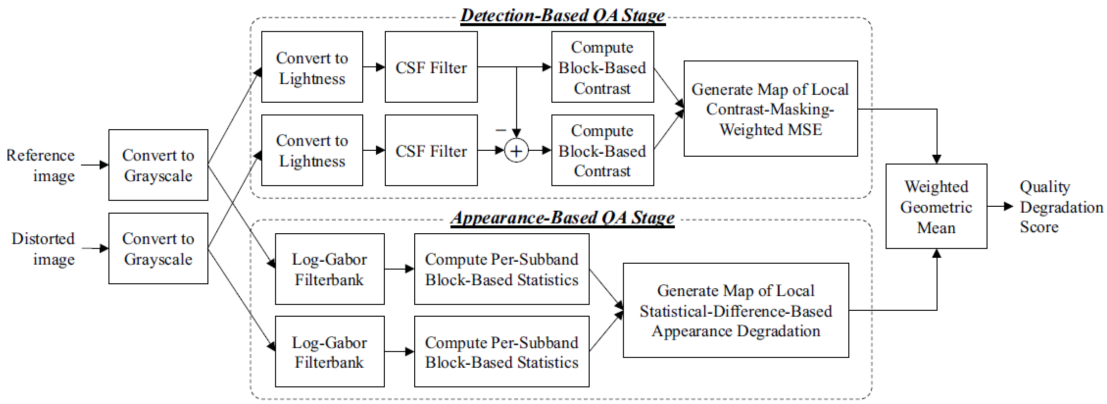


Figure 7. MAD Overview. Adapted from (Holloway, J et al., 2016).

A GPU implementation of the MAD algorithm as discussed on the paper (Holloway, J et al., 2016) is shown in Figure 8. The appearance stage as below.

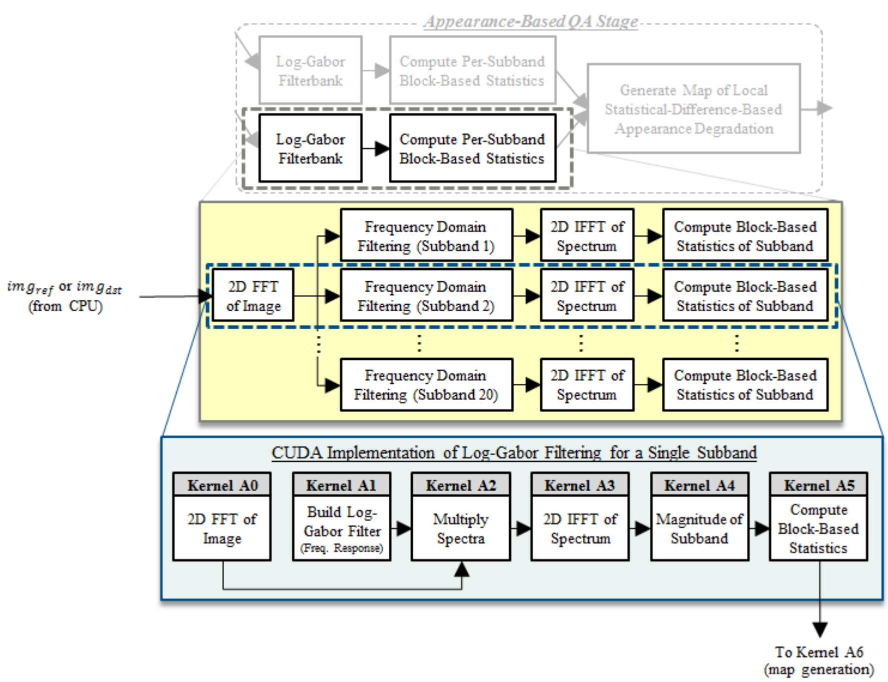


Figure 8. CUDA MAD Appearance stage. Adapted from (Holloway, J et al., 2016).

- Kernel A1 builds the frequency response of the log-Gabor filter.

- Kernel A2 shifts the filter to accommodate for the DC component lying on the edge of each quadrant.
- Kernel A3 pointwise multiplies the filter's and image's spectra
- Kernel A4 performs an inverse FFT on the filtered image.
- Kernel A5 takes the magnitude of each complex valued entry in the filtered image array removing the imaginary component from FFT.
- Kernel A6 extracts three statistical matrices from each sub band, corresponding to the standard deviation, kurtosis, and skewness of each 16x16 sub-block, each with four pixels of overlap between neighboring blocks, in each sub band.

The detection stage from the algorithm is not of interest in this research.

Section 3: Microarchitectural Analysis of Current MAD Implementation on a GPU

Profiling Strategy:

NVIDIA Visual Profiler allows the programmer to visualize and optimize the performance of the application. The profiling tool provides a graphical view of the timeline of the application's activity on both the GPU and the CPU. The visual profiler can also detect potential performance limiters and provides a list of the kernels, which are ordered by optimization importance, based on execution time and achieved occupancy.

A warp gets active between the time of starting of its execution in an SM and the time where the kernel leaves the SM finishing its last execution. In the Tesla K40 device, every SM can keep at-most 64 warps active at a time and each warp can have 32 threads. Occupancy can vary as warps begin and terminate, and can be different for each SM. Low occupancy affects instruction issue efficiency because, GPUs can context switch warps to hide latency caused by cache miss. If occupancy is low, GPUs do not have enough warps to context switch, thereby unable to hide the latency. In addition, occupancy can be seen as a two-pronged sword: when there are enough warps to hide latency, increasing the warps affects resources per thread.

Based on identifying the primary performance limiter, overall application optimization strategy is decided. Application analysis is performed on NVVP profiler, which provides insights into the microarchitectural bottlenecks, which can be seen as optimization opportunities. Hence, first step is determined from the profiler is shown as Figure 9. As soon as the application is loaded into the profiler, it runs multiple times to sample the data and displays the kernels based on performance limiters.

Rank	Description
100	[40 kernel instances] A5(float*, float*, float*, float*)
29	[20 kernel instances] A1_pt1(float*, int, int)
16	[1 kernel instances] D9(float*, float*)
5	[40 kernel instances] A4(float2*, float*)
5	[42 kernel instances] void spRadix0016A::kernel1Tex<unsigned int, float, fftDirection_t=1, unsigned int=128, unsign
4	[1 kernel instances] D7_pt1(float*, float*, float*, float*, float*, float*, float*, float*)
4	[20 kernel instances] A1_pt2(float*)
3	[40 kernel instances] A2(float2*, float*, float2*)
2	[1 kernel instances] D0_pt1(float*)
2	[1 kernel instances] D0_pt2(float*)
2	[20 kernel instances] A6(float*, float*, float*, float*, float*, float*, float, float*)

Figure 9. Rank Based Kernels Arrangement

The A5 kernel, which does fast lo stats thus calculating mean, skewness, kurtosis and standard deviation, is selected as the kernel, which has the most bottlenecks. It should also be noted that, the kernel runs a sliding window of size 16 x16 on the image.

<u>Description of A5 Kernel:</u>

1. Every thread declares a 1D array of 256 elements.
2. Every thread gathers 16x16 data from the global memory and stores onto its local memory.
3. Sum of all the elements is collected through a 1D traversal and with that data, mean is calculated.
4. Using the mean value, standard deviation, skewness and kurtosis are calculated.
5. The calculated values are scattered across the corresponding memory locations in global memory.

A5 kernel code is given in Appendix A.

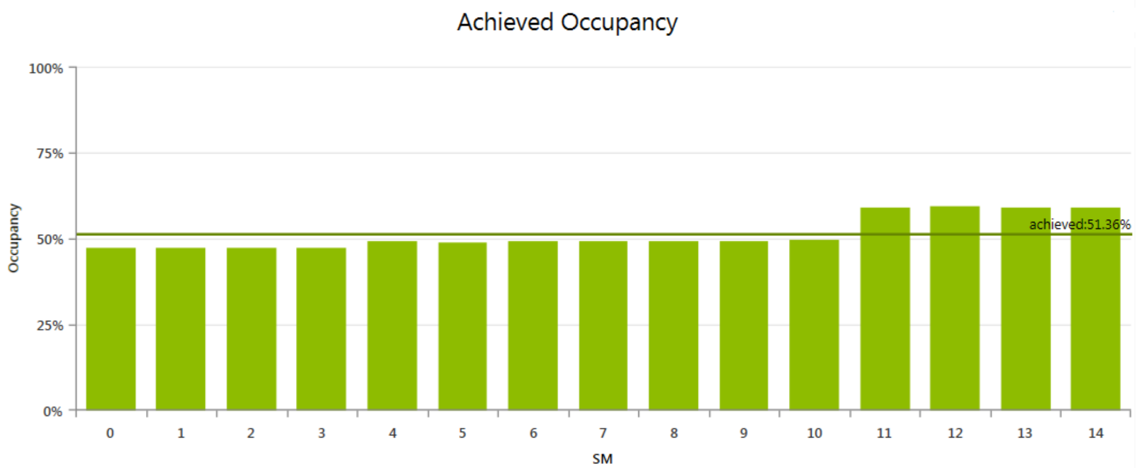


Figure 10. Achieved Occupancy of the A5 Kernel in Current Implementation

From the above image, it is observed that the Kernel A5 can achieve only 51.36 % occupancy. In K40 hardware,

Active warps/SM = (Number of warps/ Block) * (Number of Active blocks/SM)

= 8 * 8

= 64 active warps /SM

Hence, a kernel capable of spanning 64 active warps at any point in execution will give a theoretical occupancy of 100%. However, the statistics for A5 given by Nsight provides only half the number of active warps, 32 as opposed to 64. This can be attributed to kernel launch parameters.

A5 is launched with Grid configuration of (8, 8, 1) and Block configuration of (16, 16, 1).

Hence the total number of threads = 8 * 8 * 16* 16 = 16384.

Number of warps = Total number of threads/ Number of threads in a warp

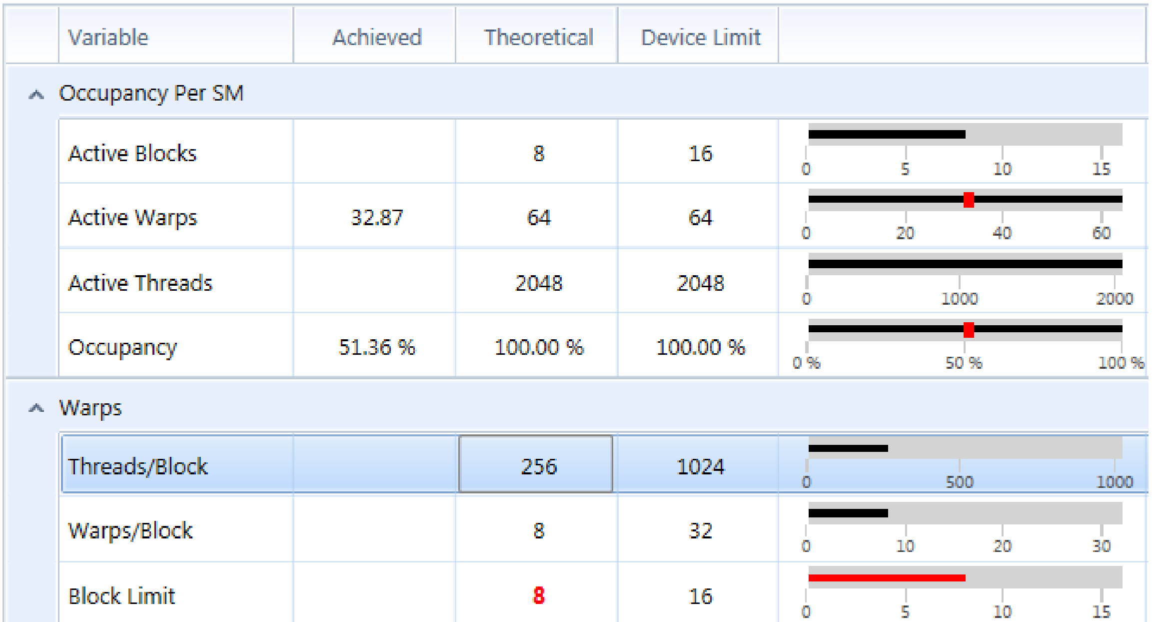
= 16384/32 = 512.

Total number of active warps supported = 64 * Number of SM on the GPU

= 64 * 15 = 960

Theoretical value of Achieved occupancy = 512 / 960 = 53.33 %.

Practical value of Achieved occupancy = 51.36 %.



Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		8	16
Active Warps	32.87	64	64
Active Threads		2048	2048
Occupancy	51.36 %	100.00 %	100.00 %
Warps			
Threads/Block		256	1024
Warps/Block		8	32
Block Limit		8	16

Figure 11. Occupancy Table of A5 Kernel

The 2% difference in practical and theoretical achieved occupancy can be attributed to the factors listed below (Nsight, N. V. I. D. I. A., & Edition, V. S. (2013)).

- <u>Unbalanced workload within blocks</u>
    - o If not all the warps within a block execute at the same point, the workload is unbalanced.
- <u>Unbalanced workload across blocks</u>
    - o The workload is unbalanced if blocks within a grid do not all execute for the same amount of time.
- <u>Too few blocks launched</u>

- A5 is launched only with 512 warps whereas the Tesla K40 GPU is capable of handling 960 warps at a time.

In addition to these, the kernel is executing for 1.047ms.

In order to evaluate the development process guided by the profiler, in this study, the current MAD implementation is profiled in terms of

1. Memory Bandwidth
2. Compute Resources
3. Instruction and Memory latency

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: Performs profiling on memory operations to the global memory. Specifically focuses on the communication between SMs and L2 cache.
- Local: Performs profiling on memory operations to the local memory. Specifically focuses on the communication between the SMs and the L1 cache.
- Cache: Performs profiling on the communication between L1 cache and texture cache with the L2 cache for all executed memory operations.
- Buffers: Performs profiling on the communication between the L2 cache with the device memory and system memory.

In addition to these, Memory statistics on atomics, texture and shared memory are not collected because the current implementation does not exploit any of those features.

<u>Memory Statistics – Global:</u>

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 provided. From the statistic given in Figure 12, cached loads uses the L1 cache or texture cache as well as L2 whereas uncached loads uses only the L2 cache.

On Tesla K40, L1 cache line size is 128-bytes, memory accesses that are cached in both L1 and L2 are serviced with 128-byte cache line size. Memory accesses that are cached only in L2 are serviced with 32-byte cache line size. This is to reduce over-fetch for instance, in case of scatter memory operations.

A warp in execution accessing device memory (LD or ST assembly instructions), coalesces the memory accesses of all the threads (32 threads share a program counter) in a warp into one or more of these memory transactions depending on the size of the word accessed by each thread as well as the distribution of the memory addresses across the threads. 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.

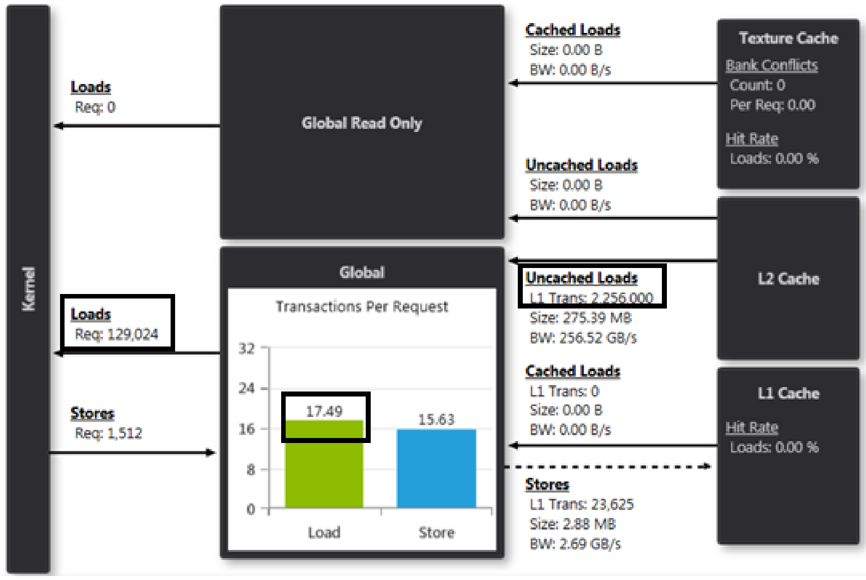


Figure 12. [A5 Kernel] Memory Statistics - Global

The figure above 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 4byte access (32 threads * 4 byte = 128 byte cache line), 2 transactions for a 8byte access (32 threads * 8 byte = 256 byte; 2 cache lines) access.

The code in Table 4 accesses the global memory.

Table 4

Global Memory Stride in Original Implementation

```
for(ib = i; ib < i + 16; iB++)
{
        for(jb = j; jb < j + 16; jB++)
        {
                xVal_local[idx] = xVal[iB * 512 + jB];
                id++;
```

```
        }
}
```

In current A5 kernel, 129,024 requests are made resulting in 2,256,000 transactions. Each of the requests are 4-byte requests (float). Hence,

Transactions for load = 2256000/129024 = 17.485

Transactions for store = 23625/1512 = 15.625

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. From the profiler, it is seen that 129024 requests have caused 3008000 transactions causing 23.3 L2 transactions per request.

Memory Statistics – Local:

Local memory resides in device memory, so, access to local memory takes the same latency as global memory access. Arrays that are declared in the kernel are automatically saved in global memory. In addition, if there are not enough registers to accommodate the entire auto, variables (register spilling); the variables are saved in local memory.

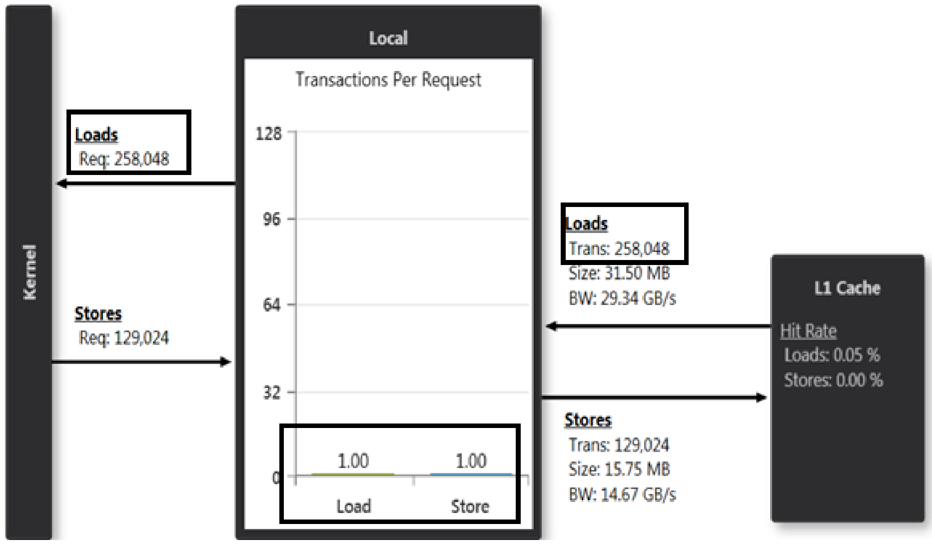


Figure 13. [A5 Kernel] Memory Statistics - Local

Table 5 is responsible for the transactions

Table 5

Global Memory Stride in Original Implementation and Mean Calculation

```
for(ib = i; ib < i + 16; iB++)
{
        for(jb = j; jb < j + 16; jB++)
        {
              xVal_local[idx] = xVal[iB * 512 + jB];
              id++;
        }
}
```

```
for(idx = 0 ; idx < 256; idx++)
   mean += xVal_local[idx];
```

Every thread can use maximum of 32 registers (Nvidia, C. U. D. A., 2014). If a kernel uses more than 32 registers, then the data gets spilled over to the local memory. Every thread executing the kernel A5 declares a local storage of 256 elements. The launch configuration of the kernel has 256 threads per block.

Total local memory needed by a block = 256 threads * 256 elements per thread * 4-byte each

= 262144 bytes

= 256KB

The on-chip memory of an SM in Tesla K40 is of size 48KB. Hence, the register spill gets carried over to the global memory resulting in bad performance as global memory incurs 200-400 cycle latency. Even though load requests made by the kernel and load transactions are linear, local memory access makes the transactions expensive.

Memory Statistics – Caches:

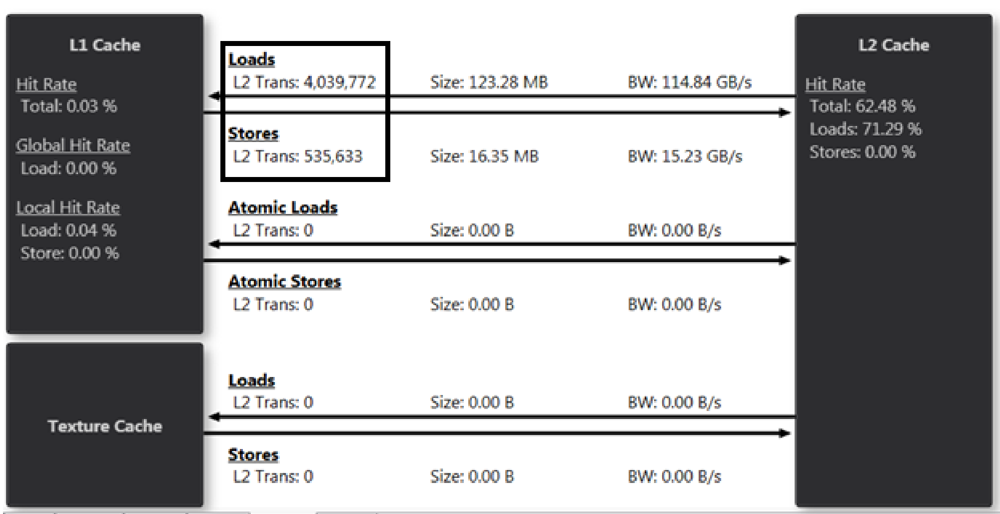


Figure 14. [A5 Kernel] Memory Statistics - Cache

There are 3 data caches: L1, L2 and texture/Read-only. If the data is present in both L1 and L2, 128-byte cache line transactions are done otherwise 32-byte transaction. If the data block is cached in both L2 and L1, and if every thread in a warp accesses a 4-byte value from random sparse locations which miss in L1 cache, each thread will cause one 128-byte L1 transaction and four 32-byte L2 transactions. This will cause the load instruction to reissue 32 times more had the values were adjacent and cache-aligned.

Here, the cache hit rate is very low because by default on Tesla K40 device, L1 is not used for load/store purpose.

Memory statistics – Buffer:

Buffers are memory locations either in system or as device memory. Latency of access to buffer is higher than shared/L1/L2. Hence it is better to have the data stored in one of the memory subsystem rather than accessing from buffer. Also, it is better to avoid re-accessing the same buffer data multiple times; its better to have them stored in memory subsytem. An initial access to buffer is mandatory, however it's the programmers responsibility to store them on-chip.

From the chart Figure 15, it can be seen that, GPUs do not access the system memroy directly (unified memory addressing). It can also be seen that Tesla K40 is capable of providing 288GB/s bandwidth. However, small sparse transfers as opposed to making larger transfer is causing the reduction in bandwidth.

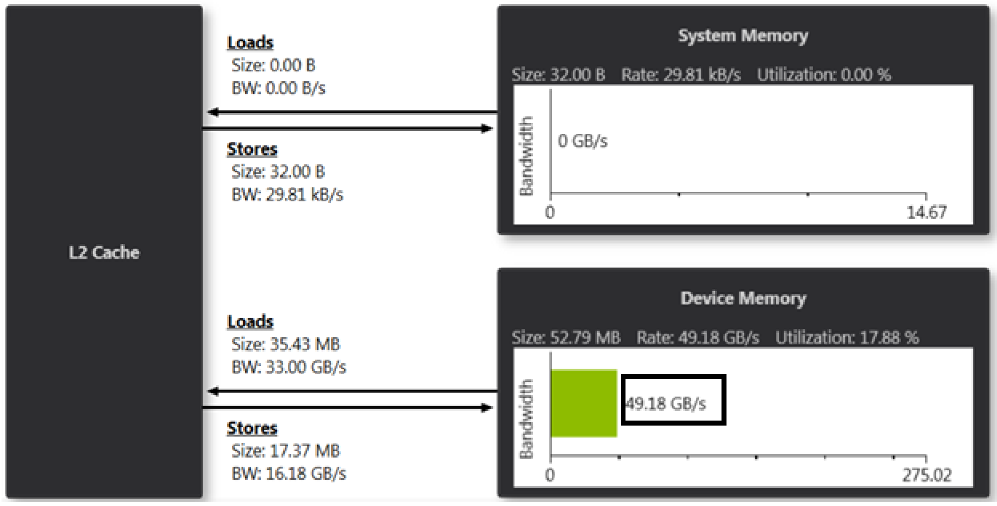


Figure 15. [A5 Kernel] Memory Statistics - Buffers

Hence, from the above memory statistics, it can be observed that each kernel is performing 256 global memory access even though there is an overlap between the data used by threads seems to be an issue causing bottleneck.

<u>Compute Resources:</u>

The factors that affect the compute resources are

1. Divergent branches
2. Low warp execution efficiency
3. Over subscribed funcitonal units.

Since all the threads within a warp share the program counter, flow control can have serious impact on the efficiency of kernel execution. If there are lot of divergent branches through the kernel code, then the time taken by kernel execution takes longer resulting in imbalance. When a flow control instruction is executed, threads are diverged such that different threads take different execution paths. In this case, all the paths must be serialized because all the threads share the program counter hence increasing the number of instrucitons in this warp.

When all the different paths have executed, the threads converge back into the same execution path. Branch efficiency is the ratio of executed flow control decisions to all the executed conditionals. On the other hand, Control flow efficiency depends on how many threads are not predicated off.

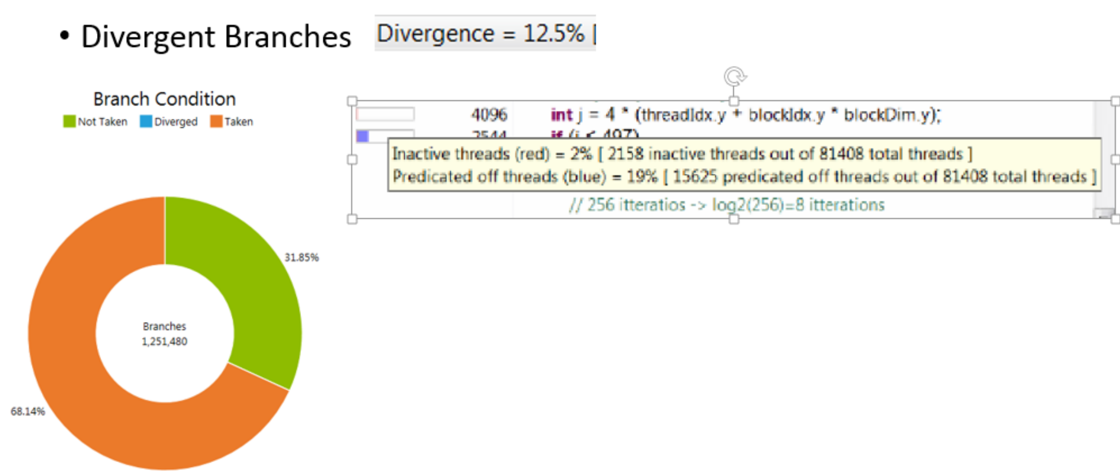


Figure 16. [A5 Kernel] Branch Divergence

The chart above shows the distribution of executed branches that causes divergence. The percentage of divergent branches seems less.

Inactive threads: One of the reasons where a thread within a warp can be disabled. A) If the block size is not a multiple of warp size, then the last warp in the block will have inactive threads. B) When some threads in a warp finish execution and exit the warp whereas the other threads still continue their execution. The 2% inactive threads are due to the fact that only 1/4th of threads launched can execute.

Predicated off threads: When a flow control instruction is encountered, divergent branches can occur because a set of threads take a path whereas the other set of threads take another path.

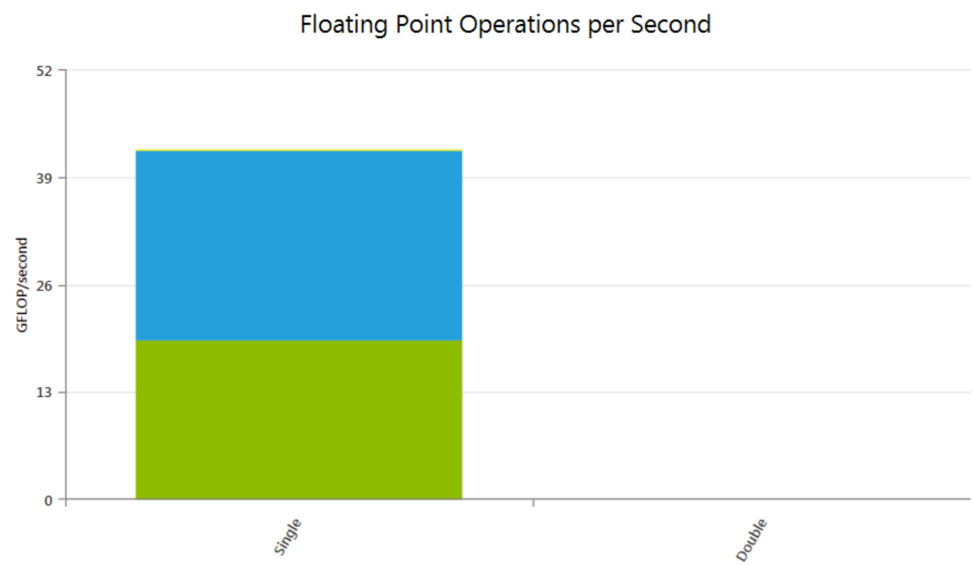


Figure 17. [A5 Kernel] Throughput

Theoretically, Tesla K40 is capable of achieving 5.12TFLOPS. In the above chart, the blue bar indicates Single ADD and blue bar indicates single MUL. The chart displays the weighted sum of all executed single precision operations per second. It can be seen that A5 has achieved 42.16 GFLOPS. The reduction in the FLOPS achieved can be tied back to the reduction in achieved occupancy, as the compute units are prone to lay idle when there is a memory dependency.

<u>Warp-issue efficiency:</u>

This experiment deals with the device's ability to issue instructions. Active warp is the number of warps that can be active at any cycle. It is also possible that the warps get context switched when waiting on a resource (global memory access, barrier synchronization). In this case, the warps can become stalled.

Active warp = stalled warp + eligible warp.

Warps per SM is shown in Figure 18.

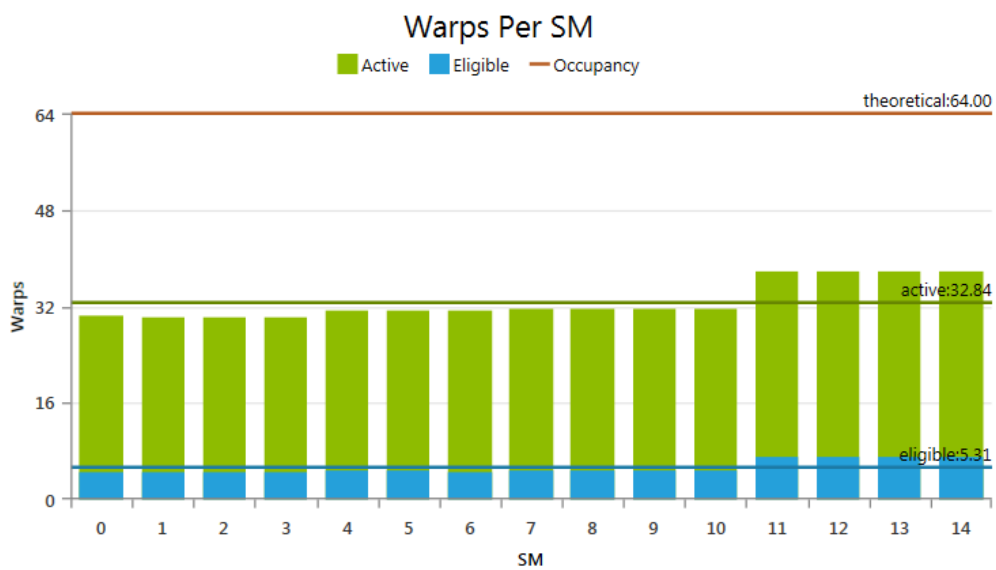


Figure 18. [A5 Kernel] Warps Per SM

Active warps are active from the time they are brought in for execution, until the time they terminate. Each warp scheduler maintains a warp pool of active warps. Warps are eligible if they are able to issue next instruction.

Less active threads are attributed to less occupancy, unbalanced workloads, and execution dependencies.

<u>Over-subscribed functional units:</u>

The TeslaK40 GPU is capable of performing 192 32-bit floating point add, multiply and multiply-add instructions every cycle. However, the achieved throughput depends on the application usage of these units.

Figure 19 shows the distribution of Load/Store, Arithmetic and Control-Flow operations on the system.

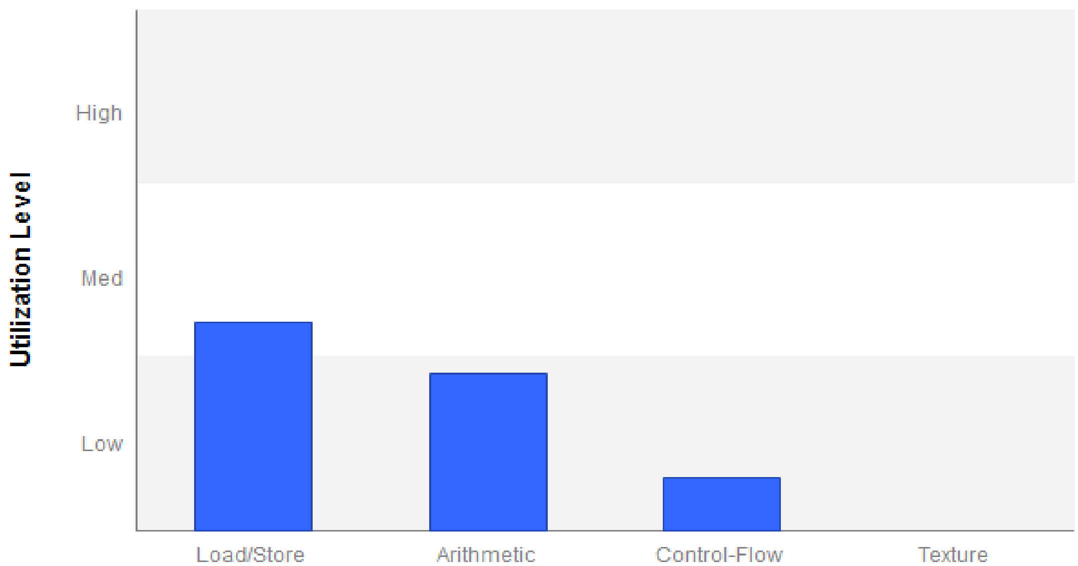


Figure 19. [A5 Kernel] Compute Units' Utilization

This not-so-optimum utilization can be attributed to stalls or the workload is not enough to saturate the compute units.

Latency limited:

Latency is influenced by occupancy and instruction stalls. Occupancy has been covered earlier. Low occupancy causes low instruction issue efficiency because, there are not many warps to hide the latency.

Instruction stalls:

Typically, when a memory (LD/ST) instruction is issued by a warp and if the requests are coalesced, then a thread requesting 4-bytes data will fetch 128-byte chunk enough to supply to all the threads in the warp. However, if the memory request is non-coalesced, then the warp scheduler needs to reissue the instructions for all the 32 threads in a warp causing significant overhead.

Thread divergence and bank conflicts on shared memory can also cause instruction replay. Each replay hinders the progress of the warp scheduler being able to issue further instructions.

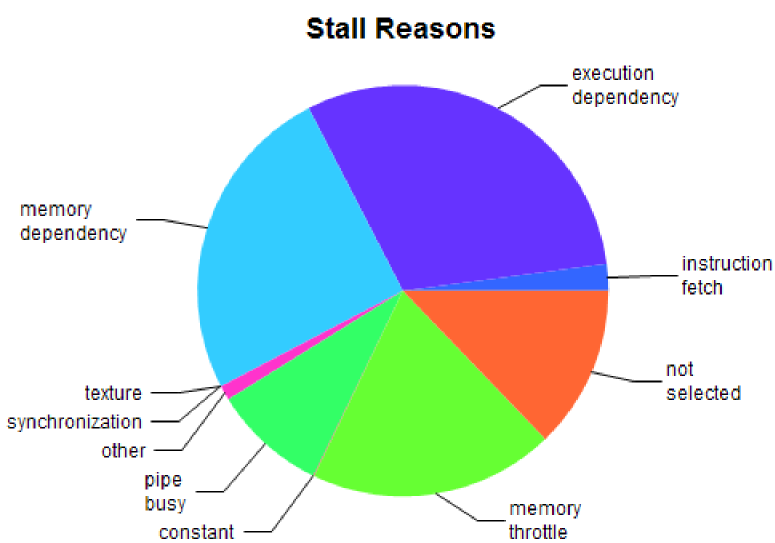


Figure 20. [A5 Kernel] Stall Reasons

The above chart shows most of the kernel depends on the memory operations from global memory. A load/store cannot be made because too many requests of given type are outstanding. In this case, over-dependent on global memory access by every thread. This fact is further bolstered by memory throttle which indicates a large number of pending memory operations that prevent forward progress. Execution dependency occurs when an input required by the instruction is not yet available. Since the calculation of higher statistics depend on mean, execution dependency is observed.

From the above microarchitectural analysis, it can be concluded that

1. The kernel is memory bandwidth limited. Every single thread within a warp is making a 128-byte request from global memory thus aggravating the global memory access. By analyzing the global memory stride and by utilizing the unused shared memory, it is possible to improve the performance.
2. A5 Kernel has very low occupancy because the number of launched warps are not enough to hide the latency. It is possible to increase the occupancy by launching a thread for each individual pixel of the input image.
3. About 2% of the launched threads are inactive as a result of branch divergence. This can be resolved by efficiently restructuring the code.

As part of this thesis, the first two performance limiters mentioned above will be analyzed in detail and resolved because, 2% thread divergence does not hurt the performance as much as memory bandwidth as well as occupancy. Also, in the shared memory implementation discussed in Chapter 3, this divergence is automatically resolved since there will be a thread launched for every single pixel of the input image.

CHAPTER 3

PERFORMANCE IMPROVEMENT

In this chapter, proposed changes from the last section will be taken into consideration. The current A5 kernel will be modified accordingly to reduce global memory access and the results will be compared.

Section 1: **Global memory access pattern**

In short, kernel A5 performs the following

1. Every thread gathers data from global memory and stores onto its local memory using nested for loops.
2. Iterate over the local memory and sum all the elements.
3. Using the sum, calculate mean.
4. Using the mean value, calculate standard deviation, kurtosis and skewness.
5. Store the calculated values onto appropriate locations in global memory.

<u>Analysis of the memory access pattern of the gather operation:</u>

From the code presented in Appendix A, the gather operation is shown in Table 6.

1. Given a thread block of 256 threads, every thread gathers 256 elements from global memory corresponding to its global thread index.

   Table 6

   An Instance of Global Memory Stride in Original Implementation

```
int x_index = 4 * (threadIdx.x + blockIdx.x * blockDim.x);
int y_index = 4 * (threadIdx.y + blockIdx.y * blockDim.y);
xVal_local[0] = xVal[x_index * 512 + y_index];
```

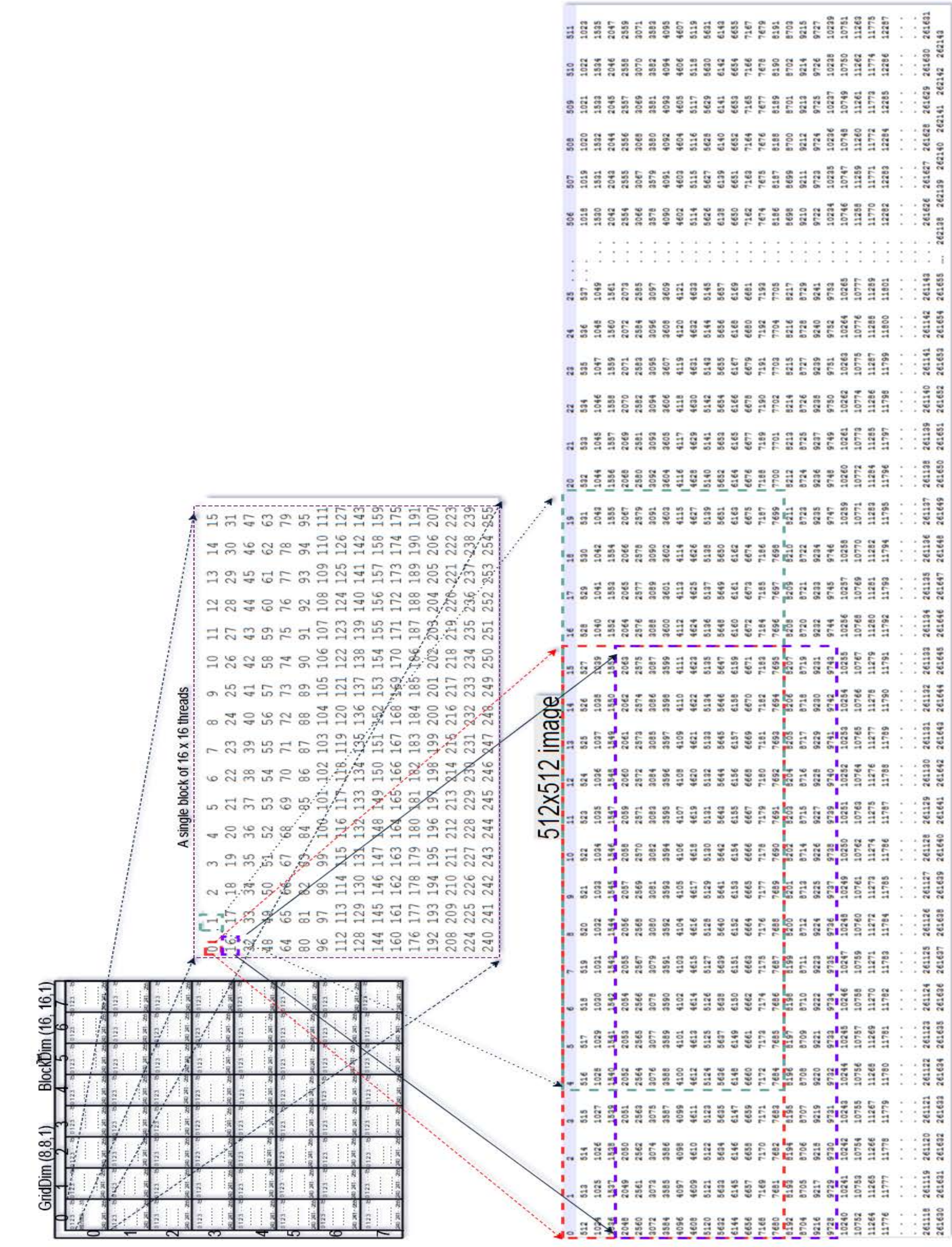


Figure 21. [A5 Kernel] Memory Stride Visualization

From the image above, there is overlap among the elements gathered from global memory by the 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 half-warp utilizing only 60 bytes data and thus discarding the rest of fetched in data. Hence, for a 16 * 16 iteration, 32 * 32 4-byte data are fetched. For the performance analysis, only the mean computation of A5 is taken into account as described in Appendix B and Appendix C. Only the mean computation in A5 kernel takes 1.000.512 ms.

<u>Proposed changes in shared memory implementation:</u>

1. In order to improve occupancy, grid size of shared memory implementation is changed to (32, 32, 1). Block size remains the same as original implementation (16, 16, 1).
2. Unlike the original implementation, there is loop involved in fetching the data from global to shared memory. Instead, every thread will access a memory location based on its global thread id. It can be calculated as

Table 7

Manual Loop Unroll for Mean Computation

```
int global_idx = (threadIdx.x + blockIdx.x * blockDim.x);

int global_idy = (threadIdx.y + blockIdx.y * blockDim.y);

xVal_smem[threadIdx.x][threadIdx.y] = xVal[global_idx* N + global_idy];
```

Figure 22 is an illustration of the shared memory implementation.

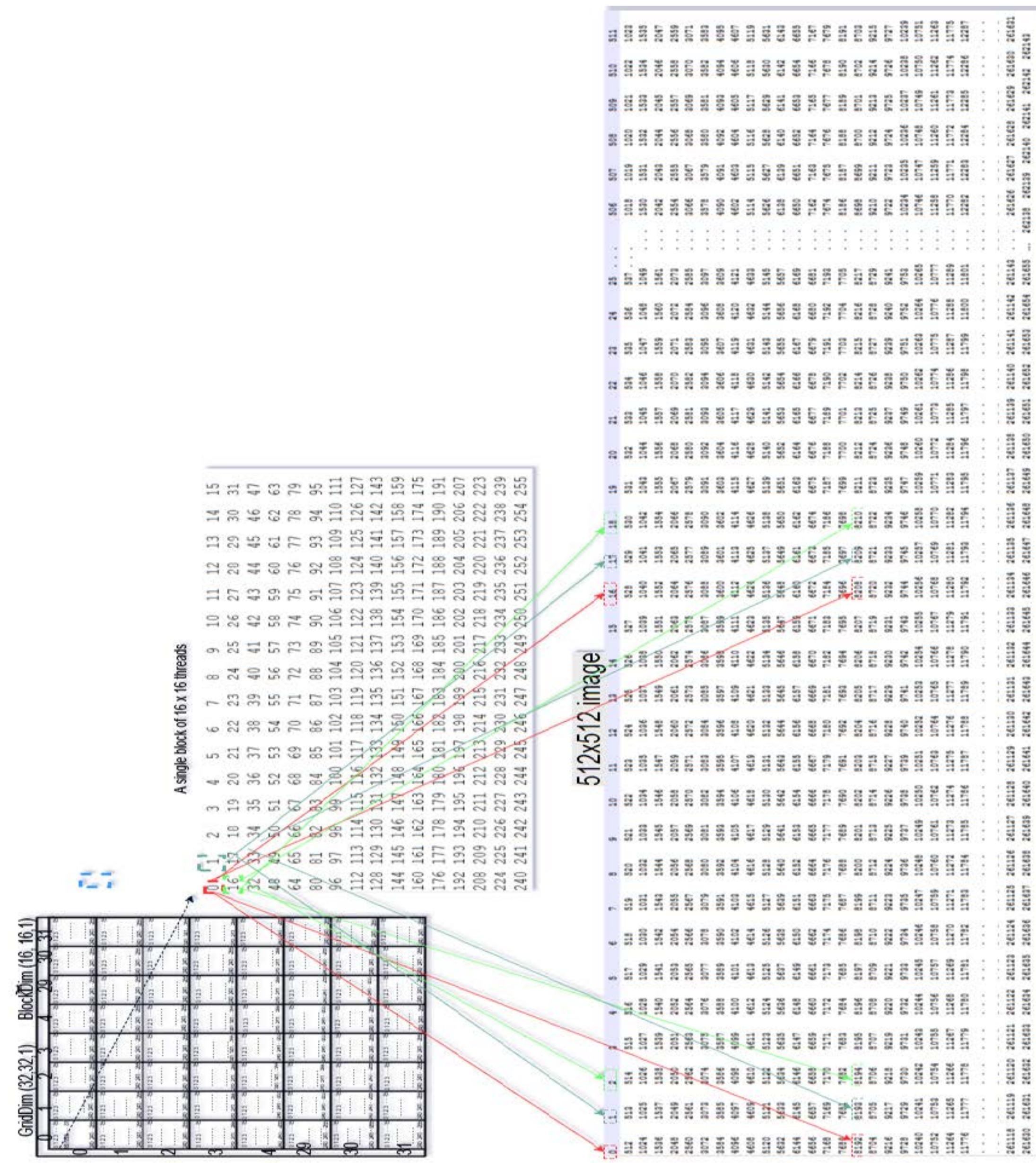


Figure 22. [A5 Kernel With Shared Memory] Memory Stride Visualization

As soon as all the threads bring in the data, (explicit barrier synchronization is done using __syncthreads), a 2D sliding window of 16 * 16 size is iterated over the shared memory to calculate the mean.

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 as shown in Table 8. This is done by exploiting the fact that the window comprises of 16 x 16 elements.

Table 8

Shared Memory Banks Visualization

```
    for (int x = threadIdx.x; x < WIN_SIZE + threadIdx.x; x++)
    {
        mean += (xVal_smem[x][y] + xVal_smem[x][y + 1] + xVal_smem[x][y+2]
    + xVal_smem[x][y+3] + xVal_smem[x][y+4] + xVal_smem[x][y+5]+
    xVal_smem[x][y+6]+ xVal_smem[x][y+7] + xVal_smem[x][y+8] +
    xVal_smem[x][y+9]+ xVal_smem[x][y+10]+ xVal_smem[x][y+11] +
    xVal_smem[x][y+12] + xVal_smem[x][y+13]+ xVal_smem[x][y+14]+
    xVal_smem[x][y+15]);
}
```

Bank conflict:

In order to achieve high memory bandwidth on concurrent accesses to the shared memory, 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 access 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. Ideally, only one bank should be accessed by a threads from a warp per cycle.

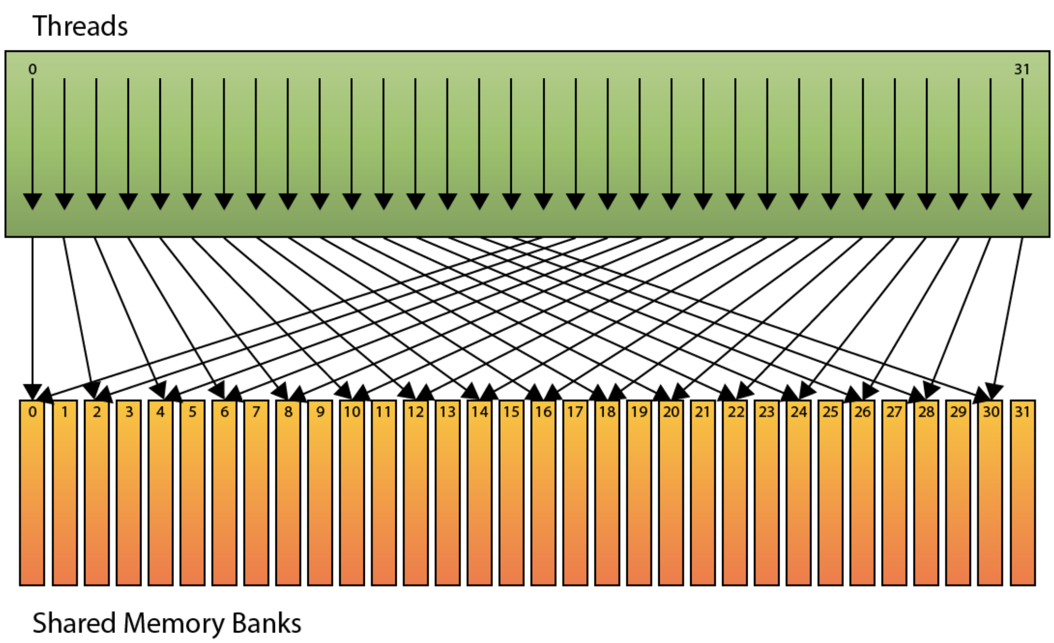


Figure 23. Shared Memory Bank Conflict. Adapted from "http://www.3dgep.com/optimizing-cuda-applications", n.d..

Threads within a block are numbered in the equivalent of column major order. Hence using smem[threadId.x][threadId.y] causes threads in a warp reading from the same column which means they are reading from the same memory bank resulting in bank conflicts. Figure 24 image provides the order of threads executed within a warp.

threadId.x	threadId.y
0	0
1	0
2	0
0	1
1	1
2	1
0	2
1	2
2	2

Figure 24. Threads Launch Order Within a Warp

Typically, bank conflict is resolved by using Smem[threadId.y][threadId.x] instead of Smem[threadId.x][threadId.y].

Shared memory bank conflict can be seen from the profiler in Figure 25.

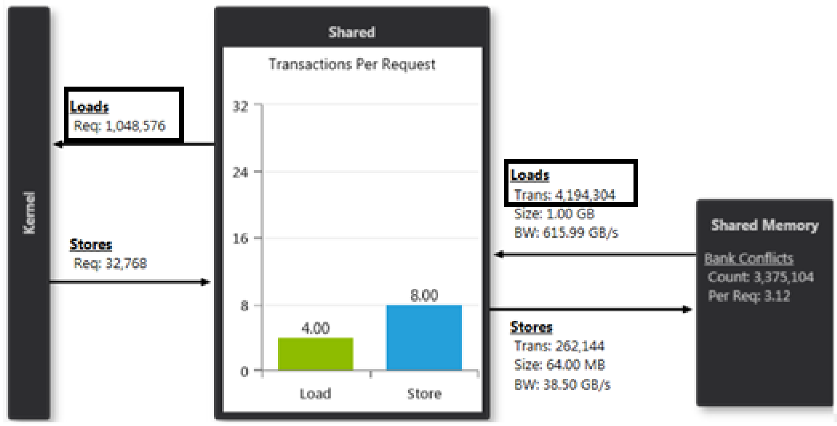


Figure 25. [A5 Kernel With Shared Memory] Memory Statistics - Shared Memory

After resolving the bank conflicts, the run time is improved and it is reduced to 757.984 us. 25 % improvement over the original implementation.

Section 2: **Results**

The following microarchitectural analysis is for the shared memory implementation, without nested-for loops and bank conflict resolved.

Memory statistics: Global:

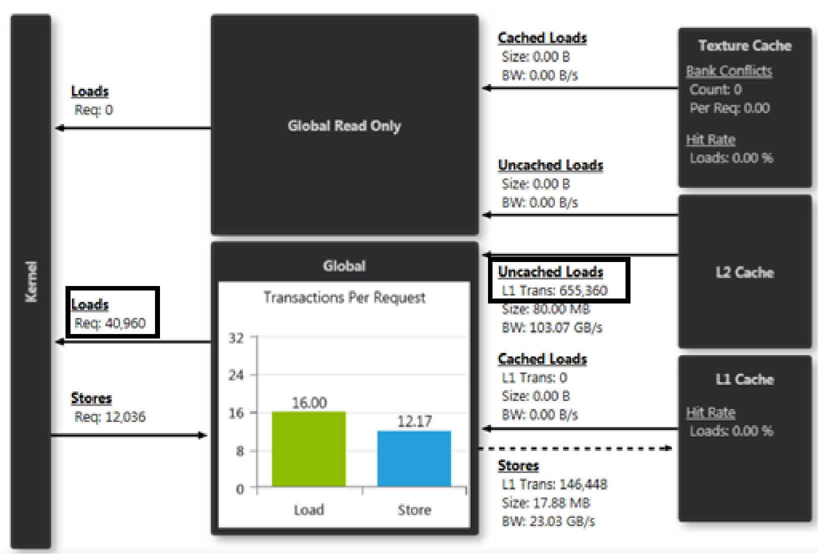


Figure 26. [A5 Kernel With Shared Memory] Memory Statistics - Global

It can be observed that the number of requests to global memory has reduced drastically (40,960 vs 129,024) improving the runtime.

<u>Memory statistics: Shared:</u>

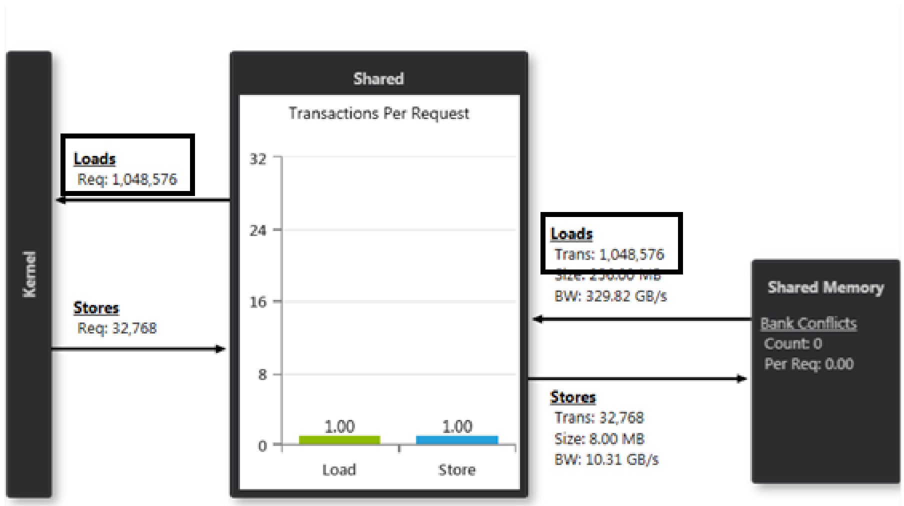


Figure 27. [A5 Kernel With Shared Memory] Memory Statistics - Shared Memory Bank Conflict Resolved

1. No bank conflict
2. Effective use of shared memory showing 1 to 1 correspondence between load and store.

Table 9

Results

Original implementation	1.000512ms
Shared memory implementation with nested for loop for mean calculation	1.623392ms

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

CHAPTER 4

CONCLUSION

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 thesis, 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 A5 kernel which performs local statistics computation as the most problematic kernel. Further analysis of A5 kernel has shown that the kernel is memory bandwidth limited with very less occupancy. 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.

So, the A5 kernel was restructured to bring in the data from global memory and store it in on-chip shared memory and then perform the mean calculation. Initially, a nested for loop to sum all the elements in the shared memory was implemented. But, the nested for loop worsened the runtime of the kernel. Hence a manual loop unroll along with resolving bank conflict reduced the runtime of the kernel. 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 document, only the mean calculation is taken into account and the performance is improved. It can be extended to kurtosis, standard deviation and skewness. As an extension to this thesis work, the higher order statistics can be optimized by resolving the bottleneck limiting its performance. It is expected to lead to much higher performance gains. In addition to that, only the memory hierarchy has been dealt in detail and efficient shared memory implementation has been provided. 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. The Tesla K40 GPU offers fast accessibility to read-only data cache, architecture specific instructions (shuffle), and ptx-optimization which offers ultimate control over the hardware than high level language. It is also possible that, by merging smaller kernels together, performance can be improved.

REFERENCES

"In-Depth Comparison of NVIDIA Tesla "Kepler" GPU Accelerators," Retrieved from https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-kepler-gpu-accelerators/

Adapted by author from "Lossless_compression," Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc., 27 Sept. 2016. Web. 07 Oct. 2016.

Alam, S. R., Agarwal, P. K., Smith, M. C., Vetter, J. S., & Caliga, D. (2007). Using FPGA devices to accelerate biomolecular simulations. *Computer*, *40*(3), 66-73.

Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., & Aamodt, T. M. (2009, April). Analyzing CUDA workloads using a detailed GPU simulator. InPerformance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on (pp. 163-174). IEEE.

Barrett, H. H. (1990). Objective assessment of image quality: effects of quantum noise and object variability. JOSA A, 7(7), 1266-1278.

Bauer, M. W., & Gaskell, G. (Eds.). (2000). *Qualitative researching with text, image and sound: A practical handbook for social research*. Sage.

Brodtkorb, A. R., Hagen, T. R., Schulz, C., & Hasle, G. (2013). GPU computing in discrete optimization. Part I: Introduction to the GPU. EURO journal on transportation and logistics, 2(1-2), 129-157.

Brosnan, T., & Sun, D. W. (2004). Improving quality inspection of food products by computer vision—a review. Journal of Food Engineering, 61(1), 3-16.

Buiten, H. J., & Van Putten, B. (1997). Quality assessment of remote sensing image registration—analysis and testing of control point residuals.ISPRS journal of photogrammetry and remote sensing, 52(2), 57-73.

Chandler, D. M. (2013). Seven challenges in image quality assessment: past, present, and future research. ISRN Signal Processing, 2013.

Charrier, C., Knoblauch, K., Moorthy, A. K., Bovik, A. C., & Maloney, L. T. (2010, January). Comparison of image quality assessment algorithms on compressed images. In IS&T/SPIE Electronic Imaging (pp. 75290B-75290B). International Society for Optics and Photonics.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., & Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. Journal of parallel and distributed computing,68(10), 1370-1380.

Chen, W. H., Smith, C. H., & Fralick, S. C. (1977). A fast computational algorithm for the discrete cosine transform. IEEE Transactions on communications, 25(9), 1004-1009.

CUDA toolkit documentation. Parallel thread execution ISA Version 5.0. Retrieved from http://docs.nvidia.com/cuda/parallel-thread-execution/

Cuda Toolkit documentation."L1 cache". Retrieved from http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., ... & Yelick, K. (2008, November). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (p. 4). IEEE Press.

Davidson, J. W., & Jinturkar, S. (1994, August). Memory access coalescing: a technique for eliminating redundant memory accesses. In ACM SIGPLAN Notices (Vol. 29, No. 6, pp. 186-195). ACM.

Fung, W. W., & Aamodt, T. M. (2011, February). Thread block compaction for efficient SIMT control flow. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture (pp. 25-36). IEEE.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., ... & Volkov, V. (2008). Parallel computing experiences with CUDA. *Micro, IEEE*, *28*(4), 13-27.

George, A., & Livingston, S. J. (2013). A survey on full reference image quality assessment algorithms. International Journal of Research in Engineering and Technology, 2(12), 303-307.

Gepner, P., & Kowalik, M. F. (2006, September). Multi-core processors: New way to achieve high system performance. In International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06) (pp. 9-13). IEEE.

Glaskowsky, P. N. (2009). NVIDIA's Fermi: the first complete GPU computing architecture. White paper, 18.

Goldmark, P. C., & Dyer, J. N. (1940). Quality in television pictures. Journal of the Society of Motion Picture Engineers, 35(9), 234-253.

Hill, F., & Kelley, S. (2007). *Computer graphics using OpenGL, 3/E*. Pearson.

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.

Hong, S., & Kim, H. (2009, June). An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. InACM SIGARCH Computer Architecture News (Vol. 37, No. 3, pp. 152-163). ACM.

Huang, J., & Shi, Y. Q. (1998). Adaptive image watermarking scheme based on visual masking. Electronics letters, 34(8), 748-749.

Iwai, K., Kurokawa, T., & Nisikawa, N. (2010, November). AES encryption implementation on CUDA GPU and its analysis. In Networking and Computing (ICNC), 2010 First International Conference on (pp. 209-214). IEEE.

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Jog, A., Kayiran, O., Mishra, A. K., Kandemir, M. T., Mutlu, O., Iyer, R., & Das, C. R. (2013, June). Orchestrated scheduling and prefetching for GPGPUs. In ACM SIGARCH Computer Architecture News (Vol. 41, No. 3, pp. 332-343). ACM.

Jones, S. (2012, May). Introduction to dynamic parallelism. In GPU Technology Conference Presentation S (Vol. 338, p. 2012).

Kamble, V., & Bhurchandi, K. M. (2015). No-reference image quality assessment algorithms: A survey. Optik-International Journal for Light and Electron Optics, 126(11), 1090-1097.
Kirk, D. B., & Wen-mei, W. H. (2012). Programming massively parallel processors: a hands-on approach. Newnes.

Kunt, M., & van den Branden Lambrecht, C. (1998). Special issue on image and video quality metrics. Signal Processing, 70(3).

Larson, E. C., & Chandler, D. M. (2010). Most apparent distortion: full-reference image quality assessment and the role of strategy. *Journal of Electronic Imaging*, *19*(1), 011006-011006.

Lee, Y. H., Khalil-Hani, M., Bakhteri, R., & Nambiar, V. P. (2016). A real-time near infrared image acquisition system based on image quality assessment.Journal of Real-Time Image Processing, 1-18.

Li, C., Wu, H., Chen, S., Li, X., & Guo, D. (2009, August). Efficient implementation for MD5-RC4 encryption using GPU with CUDA. In 2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication (pp. 167-170). IEEE.
Lundström, C. (2006). Technical report: Measuring digital image quality.

Manavski, S. A. (2007, November). CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on (pp. 65-68). IEEE.

Manavski, S. A. (2007, November). CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on (pp. 65-68). IEEE.

Moore, G. E. (2006). Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, *3*(20), 33-35.

Harris, M. (2007). Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology*, *2*(4).

Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Manavski, S. A. (2007, November). CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on* (pp. 65-68). IEEE.

Michalakes, J., & Vachharajani, M. (2008). GPU acceleration of numerical weather prediction. Parallel Processing Letters, 18(04), 531-548.

Micikevicius, P. (2010). Analysis-driven optimization. In GPU technology conference (pp. 1-55).

Mittal, A., Soundararajan, R., Muralidhar, G. S., Bovik, A. C., & Ghosh, J. (2013, March). Blind image quality assessment without training on human opinion scores. In IS&T/SPIE Electronic Imaging (pp. 86510T-86510T). International Society for Optics and Photonics.

Mohammadi, P., Ebrahimi-Moghadam, A., & Shirani, S. (2014). Subjective and objective quality assessment of image: A survey. arXiv preprint arXiv:1406.7799.

Moorthy, A. K., & Bovik, A. C. (2011). Visual quality assessment algorithms: what does the future hold?. Multimedia Tools and Applications, 51(2), 675-696.

Nickolls, J. (2007). Nvidia GPU parallel computing architecture. IEEE Hot Chips, 19.

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. Queue, 6(2), 40-53.

Nsight, N. V. I. D. I. A., & Edition, V. S. (2013). 3.0 User Guide. NVIDIA Corporation.

Nvidia, C. (2012). *NVIDIAs next generation CUDA compute architecture: Kepler GK110*. Technical report, Technical report, 2012.

Nvidia, C. (2007). NVIDIA CUDA programming guide (version 1.0). NVIDIA: Santa Clara, CA.

Nvidia, C. U. D. A. (2007). Compute unified device architecture programming guide.

Nvidia, C. U. D. A. (2011). Nvidia cuda c programming guide. Nvidia Corporation, 120(18), 8.

NVidia, F. (2009). Nvidia's next generation cuda compute architecture.NVidia, Santa Clara, Calif, USA.

Okarma, K., & Mazurek, P. (2011). GPGPU based estimation of the combined video quality metric. In Image Processing and Communications Challenges 3 (pp. 285-292). Springer Berlin Heidelberg.

Developers, O. (2001). Open64 compiler and tools

Pappas, T. N., Safranek, R. J., & Chen, J. (2000). Perceptual criteria for image quality evaluation. Handbook of image and video processing, 669-684.

Phan, T. D., Shah, S. K., Chandler, D. M., & Sohoni, S. (2014). Microarchitectural analysis of image quality assessment algorithms. Journal of Electronic Imaging, 23(1), 013030-013030.

Phan, T., Sohoni, S., Chandler, D. M., & Larson, E. C. (2012, April). Performance-analysis-based acceleration of image quality assessment. InImage Analysis and Interpretation (SSIAI), 2012 IEEE Southwest Symposium on (pp. 81-84). IEEE.

Pitas, I. (2000). Digital image processing algorithms and applications. John Wiley & Sons.

Rui, R., Li, H., & Tu, Y. C. (2015, October). Join algorithms on GPUs: A revisit after seven years. In Big Data (Big Data), 2015 IEEE International Conference on (pp. 2541-2550). IEEE.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W. M. W. (2008, February). 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.

Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S. Z., Stratton, J. A., & Hwu, W. M. W. (2008, April). Program optimization space pruning for a multithreaded gpu. In Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization (pp. 195-204). ACM.

Said, A., & Pearlman, W. A. (1996). An image multiresolution representation for lossless and lossy compression. *IEEE Transactions on image processing*, *5*(9), 1303-1310.

Silverstein, D. A., & Farrell, J. E. (1996, September). The relationship between image fidelity and image quality. In Image Processing, 1996. Proceedings., International Conference on (Vol. 1, pp. 881-884). IEEE.

Suresh, S., Babu, R. V., & Kim, H. J. (2009). No-reference image quality assessment using modified extreme learning machine classifier. Applied Soft Computing, 9(2), 541-552.

T. Dokken, T.R. Hagen, and J.M. Hjelmervik. The GPU as a high performance computational resource. In Proceedings of Spring Conference on Computer Graphics 2005, pages 21–26, 2005

Trapnell, C., & Schatz, M. C. (2009). Optimizing data intensive GPGPU computations for DNA sequence alignment. Parallel computing, 35(8), 429-440.

Wang, Z., & Bovik, A. C. (2006). Modern image quality assessment.Synthesis Lectures on Image, Video, and Multimedia Processing, 2(1), 1-156.

Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. IEEE transactions on image processing, 13(4), 600-612.

Wang, Z., Lu, L., & Bovik, A. C. (2004). Video quality assessment based on structural distortion measurement. Signal processing: Image communication,19(2), 121-132.

Wang, Z., Simoncelli, E. P., & Bovik, A. C. (2003, November). 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.

Winkler, S. Image quality in the era of personal and social media. In Proc. of the 5th International Workshop on Image Media Quality and its Applications(pp. 1-2).

Wittenbrink, C. M., Kilgariff, E., & Prabhu, A. (2011). Fermi GF100 GPU architecture. IEEE Micro, 31(2), 50-59.

Wu, B., Zhao, Z., Zhang, E. Z., Jiang, Y., & Shen, X. (2013, February). 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.

Xu, C., Kirk, S. R., & Jenkins, S. (2009, December). Tiling for performance tuning on different models of gpus. In 2009 Second International Symposium on Information Science and Engineering (pp. 500-504). IEEE.

Zhang, L., Zhang, L., Mou, X., & Zhang, D. (2011). FSIM: a feature similarity index for image quality assessment. IEEE transactions on Image Processing, 20(8), 2378-2386.

Zhao, L., Iyer, R., Makineni, S., & Bhuyan, L. (2005, March). Anatomy and performance of SSL processing. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* (pp. 197-206). IEEE.

APPENDIX A

KERNEL A5- ORIGINAL IMPLEMENTATION

```
__global__ void A5(float* xVal, float* outStd, float* outSkw, float* outKrt)
{
	//Declarations
	//__shared__ float xVal_Shm[256];
	float xVal_local[256];

	float mean, stdev, skw, krt, stmp;
	int iB, jB;

	//for (i = 0; i<512 - 15; i += 4)
	int i = 4 * (threadIdx.x + blockIdx.x * blockDim.x);
	if (i < 497) //512-15=497
	{
		//for (j = 0; j<512 - 15; j += 4)
		int j = 4 * (threadIdx.y + blockIdx.y * blockDim.y);
		if (j < 497)
		{
				// THE FOLLOWING SET OF RUNNING SUMS
CAN BE A set of PARALLEL REDUCTIONs (in shared memory?)
			// 256 itteratios -> log2(256)=8 itterations

			// Store block into registers (256 x 4Bytes = 1kB)
			int idx = 0;
			for (iB = i; iB < i + 16; iB++)
			{
				for (jB = j; jB < j + 16; jB++)
				{
					xVal_local[idx] = xVal[iB * 512 + jB];
					idx++;
				}
			}
```

```
                    //Traverse through and get mean
                    float mean = 0;
                    for (idx = 0; idx < 256; idx++)
                            mean += xVal_local[idx];
        //this can be a simple reduction in shared memory
                    mean = mean / 256.0f;

                    //Traverse through and get stdev, skew and kurtosis
                    stdev = 0;
                    skw = 0;
                    krt = 0;
                    float xV_mean = 0;
                    for (idx = 0; idx < 256; idx++)
                    {
                            // Place this commonly re-used value into a register to
preserve temporal localitiy
                            xV_mean = xVal_local[idx] - mean;
                            stdev += xV_mean*xV_mean;
                            skw += xV_mean*xV_mean*xV_mean;
                            krt += xV_mean*xV_mean*xV_mean*xV_mean;
                    }
```

```
                    stmp = sqrt(stdev / 256.0f);
                    stdev = sqrt(stdev / 255.0f);//MATLAB's std is a bit different

                    if (stmp != 0){
                            skw = (skw / 256.0f) / ((stmp)*(stmp)*(stmp));
                            krt = (krt / 256.0f) / ((stmp)*(stmp)*(stmp)*(stmp));
                    }
                    else{
                            skw = 0;
                            krt = 0;
                    }

                    // Only this final output should be written to global memory:
                    outStd[(i / 4 * P) + j / 4] = stdev;
                    outSkw[(i / 4 * P) + j / 4] = skw;
                    outKrt[(i / 4 * P) + j / 4] = krt;

                }
        }
}
```

APPENDIX B

CURRENT IMPLEMENTATION - MEAN COMPUTATION

```
__global__ void A5_fast_lo_stats_kernel(float* xVal, float* out)
{
    //Declarations
    float xVal_local[WIN_SIZE * WIN_SIZE];
    float mean = 0, stdev = 0, skw = 0, krt = 0, stmp = 0;
    int iB, jB;
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y *
blockDim.x) + threadIdx.x;
    int i = 4 * (threadIdx.x + blockIdx.x * blockDim.x);
    int j = 4 * (threadIdx.y + blockIdx.y * blockDim.y);
    int global_idx = (threadIdx.x + blockIdx.x * blockDim.x);
    int global_idy = (threadIdx.y + blockIdx.y * blockDim.y);
    if ((i < N - WIN_SIZE) && (j < N - WIN_SIZE))//512-15=497
    {
        //for (j = 0; j<512 - 15; j += 4)
        // THE FOLLOWING SET OF RUNNING SUMS CAN BE A
set of PARALLEL REDUCTIONs (in shared memory?)
        // 256 itteratios -> log2(256)=8 itterations
        // Store block into registers (256 x 4Bytes = 1kB)
        int idx = 0;
        for (iB = i; iB < i + WIN_SIZE; iB++)
        {
```

```
                                        for (jB = j; jB < j + WIN_SIZE; jB++)
                                        {
                                                xVal_local[idx++] = xVal[iB * N + jB];
                                        }
                                }
                                //      //Traverse through and get mean
                                float mean = 0;
                                for (idx = 0; idx < WIN_SIZE * WIN_SIZE; idx++)
                                        mean += xVal_local[idx];
            //this can be a simple reduction in shared memory
                                mean = mean / 256.0f;


                                //      printf("%d %d %d %d\n", threadIdx.x, threadIdx.y,
blockIdx.x, blockIdx.y, global_idx*N + global_idy);
                                out[global_idx * N + global_idy] = mean;
                                //
                        }
                        else
                                out[global_idx * N + global_idy] = xVal[global_idx * N +
global_idy];
            }
```

APPENDIX C

SHARED MEMORY IMPLEMENTATION - MEAN COMPUTATION

```
__global__ void A5_low_stats_kernel(float* xVal, float* out)
{
    //Declarations
    //__shared__ float xVal_Shm[256];
    __shared__ float xVal_smem[WIN_SIZE + WIN_SIZE][WIN_SIZE +
WIN_SIZE]; //threadDim.x, threadDim.y size
    float mean = 0, stdev = 0, skw = 0, krt = 0, stmp = 0;
    float iB, jB;
    //https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-
Indexing-Cheatsheet.pdf

    //Used in code
    int global_idx = (threadIdx.x + blockIdx.x * blockDim.x);
    int global_idy = (threadIdx.y + blockIdx.y * blockDim.y);
    int global_idx1 = 4 * (threadIdx.x + blockIdx.x * blockDim.x);
    int global_idy1 = 4 * (threadIdx.y + blockIdx.y * blockDim.y);
    out[global_idx*N + global_idy] = xVal[global_idx* N + global_idy];



    xVal_smem[threadIdx.y][threadIdx.x] = xVal[global_idx* N + global_idy];
    xVal_smem[threadIdx.y + WIN_SIZE][threadIdx.x] = xVal[global_idx* N
+ global_idy + WIN_SIZE];
```

```
		xVal_smem[threadIdx.y][threadIdx.x + WIN_SIZE] = xVal[(global_idx +
WIN_SIZE)* N + global_idy];
		xVal_smem[threadIdx.y + WIN_SIZE][threadIdx.x + WIN_SIZE] =
xVal[(global_idx + WIN_SIZE)* N + global_idy + WIN_SIZE];
		__syncthreads();

		if ((threadIdx.x % 4 == 0 && threadIdx.y % 4 == 0))
		{
			int y = threadIdx.y;
			for (int x = threadIdx.x; x < WIN_SIZE + threadIdx.x; x++)
			{
					mean += (xVal_smem[y][x] + xVal_smem[y+1][x]
+ xVal_smem[y+2][x] + xVal_smem[y+3][x]
											+
xVal_smem[y+4][x] + xVal_smem[y+5][x]+ xVal_smem[y+6][x]+ xVal_smem[y+7][x]
											+
xVal_smem[y+8][x] + xVal_smem[y+9][x]+ xVal_smem[y+10][x]+ xVal_smem[y+11][x]
											+
xVal_smem[y+12][x] + xVal_smem[y+13][x]+ xVal_smem[y+14][x]+
xVal_smem[y+15][x]);
			}

			mean = mean / 256.0f;
```

```
            int x = blockIdx.x;
            y = blockIdx.y;
            //out[global_idx*N + global_idy] = xVal[global_idx*N +
global_idy];
            if (blockIdx.x != 31 && blockIdx.y != 31)
            {
                out[((x * 4) + threadIdx.x / 4) * N + (y * 4) + threadIdx.y
/ 4] = mean;
            }


        }
    }
```