Analysis of Hardware Usage Of Shuffle Instruction Based Performance Optimization in the Blinds-II Image Quality Assessment Algorithm

by

Ameya Wadekar

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

Approved June 2017 by the
Graduate Supervisory Committee:

Sohum Sohoni, Chair
Daniel Aukes
Sangram Redkar

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

With the advent of GPGPU, many applications are being accelerated by using CUDA programing paradigm. We are able to achieve around 10x -100x speedups by simply porting the application on to the GPU and running the parallel chunk of code on its multi cored SIMT (Single instruction multiple thread) architecture. But for optimal performance it is necessary to make sure that all the GPU resources are efficiently used, and the latencies in the application are minimized. For this, it is essential to monitor the Hardware usage of the algorithm and thus diagnose the compute and memory bottlenecks in the implementation. In the following thesis, we will be analyzing the mapping of CUDA implementation of BLIINDS-II algorithm on the underlying GPU hardware, and come up with a Kepler architecture specific solution of using shuffle instruction via CUB library to tackle the two major bottlenecks in the algorithm. Experiments were conducted to convey the advantage of using shuffle instruiction in algorithm over only using shared memory as a buffer to global memory. With the new implementation of

BLIINDS-II algorithm using CUB library, a speedup of around 13.7% was achieved.

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Dr. Sohum Sohoni of the Ira A Fulton College of Engineering at Arizona State University. I was successfully able to complete my thesis on account of his constant encouragement and motivation throughout my thesis research. I always received valuable input from him whenever I faced any difficulties and thus he always steered me in the right direction. Also, I would like to express my gratitude to NVIDIA for providing the development platform. And numerous NVIDIA technical support members who helped me solve the difficulties by providing appropriate clarifications through online forums.

I would also like to express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

I would also like to thank Arizona State University for providing resources and support required for completion of my thesis research.
Thank you,

Ameya Wadekar.

TABLE OF CONTENTS

Page

LIST OF TABLES ... vii

LIST OF FIGURES ... viii

CHAPTER

1 INTRODUCTION ... 1

2 RELATED WORK ... 4

3 BLIINDS-II ALGORITHM ... 6

4 GPGPU ... 10

Introduction of GPU ... 10

Introduction of GPGPU ... 11

Examples of GPU Acceleration ... 12

5 CUDA PROGRAMMING MODEL ... 17

Typical Sequence of CUDA Programming ... 17

CUDA Compilation Process ... 24

6 CUDA MEMORY MODEL ... 26

CHAPTER Page

Introduction 26

Types of GPU Memory 27

7 DATA TRANSFER 35

Introduction 35

Types of Optimizations in Data Transfers 36

8 EXPERIMENT 1 : REDUCE ALGORITHM 44

Reduce Algorithm 44

Modes of Implementation 47

Detailed Analysis of Results 62

9 CUDA IMPLEMENTATION OF BLIINDS-II ALGORITHM 67

Image Read on CPU And Linearization 70

Features Extraction in GPU 70

Computation of BLIINDS-II Quality Score on CPU 75

10 THRUST VS. CUB LIBRARY 76

THRUST Library 77

CHAPTER Page

CUB Library 77

Comparison of THRUST vs CUB 79

11 NEW BLIINDS-II IMPLEMENTATION AND ITS ANALYSIS 82

Implementing CUB for Sort and Reduce Function 82

Using CUB for Sort Function 83

Using CUB for Reduce Function 84

Results and Analysis of Implementation 85

12 DISCUSSION AND CONCLUSION 88

13 FUTURE WORK 90

REFRENCES 91

LIST OF TABLES

Table Page

1. Comparison of CPU And GPU Cache 27

2. Properties of Different Types of CUDA Memories 34

3. Kepler GK110 Specifications 48

4. Test System Specifications 50

5. Reduce Algorithm for Varied Array Size 61

6. Global and Shared Memory Transactions 63

7. Comparison of Thrust vs. CUB Library 80

8. Thrust Vs. CUB Implementation of BLIINDS-II Algorithm 87

LIST OF FIGURES

Figure Page

1. BLIINDS-II Algorithm Flow 7

2. GPU Accelerated Computing 11

3. Kernel Execution Policy 22

4. CUDA Project Compilation 25

5. GPU Memory Hierarchy. 33

6. Types of Shuffle Instructions 40

7. Results of Performance Experiment. 42

8. Results of Reduce Operation. 42

9. Results of Bitonic Sort Operation 43

10. Serial Implementation of Reduce Algorithm 45

11. Parallel Implementation of Reduce Algorithm 53

12. Parallel Reduce Algorithm Using Only Global Memory 56

13. Parallel Reduce Algorithm Using Shared Memory 58

14. __Shfl_Down Instruction 62

Figure Page

15. Reduction Using __Shfl_Down Instruction 58

16. Comparison of Timings For Global, Shared and Warp Shuffle
Implementations of Reduction Algorithm. 62

17. Global Memory Transactions 64

18. Shared Memory Transactions 65

19. Flowchart of CUDA Implemantation of BLIINDS-II Algorithm 69

20. NVVP Analysis of CUDA Implemantation of BLIINDS-II Algorithm 76

21. Comparison of Performance Portability Between Thrust and CUB 81

22. THRUST Vs. CUB Implementation of BLIINDS-II 86

CHAPTER 1

INTRODUCTION

With the rapid growth of internet, we are currently experiencing an enormous increase in data being generated and transferred across the globe (The Data Explosion in 2014 Minute by Minute – Infographic. (2014, July 12)). It must be noted that most of this data is in the form of digital images or videos which are streamed throughout the globe predominantly by applications such as YouTube, Skype, and WhatsApp. Image quality is basically a subjective comparison of received image with respect to the subject image. This subjective analysis can only be done on the basis of how the image is viewed and perceived by the human visual system. Even though personal preferences of humans change over time and from person to person, the underlying neural circuitry and biological processing strategies employed are strikingly similar. Based on this knowledge, an image's appearance can be altered in such a way that the received image will be considered as having a better image quality. (Chandler, D. M. (2013).) Based on this research, numerous image

quality assessment(IQA) algorithms have been developed in the past decade. These IQA algorithms are classified on the basis of availability of reference image as-

1. Full reference IQA algorithms – with both distorted and original image as input.
2. Reduced reference IQA algorithms – with distorted image and partial information about original image as input.
3. No reference IQA algorithm – with only distorted image as input. (Chandler, D. M. (2013).)

It must be noted that many real-life applications do not have access to the undistorted image as input (Yadav, A. (2016))., so No-reference IQA algorithms find application in such scenarios. This thesis is based on BLINDS-II image quality algorithm, which is a No-reference IQA algorithm. Poor computational performance is one of the major drawbacks, holding off IQA algorithms from being widely implemented ((Yadav, A. (2016); Chandler, D. M. (2013)). A single image smaller than one megapixel requires execution

time of order of seconds for calculation of a quality score. As a solution to this problem, Yadav A. presented a GPGPU based CUDA implementation of BLIINDS-II algorithm (Yadav, A. (2016)) for its computational efficiency.

His implementation was estimated to take around 9 ms for processing a 512x512 image, which means the implementation wan handle video feed of more than 100fps. According to his analysis, most of the time is spent in sorting and reduce operations based on Thrust Library, and serve as bottlenecks in the algorithm.

In this thesis, we will try to tackle these bottlenecks by analyzing the usage of hardware by the algorithm and implementing an architecture specific solution of using shuffle instruction via CUB library to reduce time required for these operations,

CHAPTER 2

RELATED WORK

In the GPU Tech conference(GTC) 2013(CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10)), Demouth J. introduced Shuffle instruction specifically designed for devices with underlying Kepler Architecture. The implementation of this instructions in various algorithms like sort, reduce, scan, etc and the eventual speedups achieved because of this was also discussed. On, February 13, 2014 Luitjens J. wrote an article in Parallel ForAll blog of NVIDIA (37. Faster Parallel Reductions on Kepler. (2016, June 22).), explaining how faster reduction can be achieved using shuffle instruction in devices with Kepler Architecture. This article was the inspiration behind evaluating Shuffle instruction and implementing it in BLIINDS-II algorithm via CUB library.

Shuffle instruction is widely used in many applications where warp-wide operation involving exchange of data among threads is needed. In a paper presented on “GPU Multisplit” (Ashkiani, S., Davidson, A., Meyer, U., &

Owens, J. D. (2016, February)) scan operation was performed using warp-wide shuffle. Also in an article on “Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler” (A. V. Monakov, V. A. Ivanishin,) Shuffle instruction was used for propagating the register holding the results. In an article on Optimized Filtering with Warp-Aggregated Atomics in Parallel ForAll blog by NVIDIA(CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics. (2014, October 06), it is suggested to use warp shuffle intrinsic for Kepler architecture to efficiently broadcast the results.

Considering the advantages of using shuffle instruction in various algorithms, flexible CUDA libraries like CUB which emphasize on making efficient use of underlying architecture, implement shuffle intrinsic when its various algorithms are run on devices with compute capability 3.0 and above(Introducing CUDA UnBound (CUB). (2014, April 14)).

CHAPTER 3

BLIINDS-II ALGORITHM

BLIINDS II is a general-purpose no-reference image quality assessment(IQA) algorithm which uses a natural scene statistics(NSS) model of discrete cosine transform(DCT) coefficients. (Saad, M. A., Bovik, A. C., & Charrier, C. (2012)) The theory of No Reference Image Quality Assessment algorithm (NR-IQA) states that the algorithm should process only the distorted image without referring the original image.

Accordingly, the input to BLIINDS-II algorithm is only the distorted image; it processes this image to measure a quality score corresponding to the image. (Yadav, A. (2016))

The algorithm runs on three different spatial scales of the image, in accordance with the HVS property of spatial local decomposition of visual stimulus (Blake and Sekuler 2006). Thus, The BLIINDS-II algorithm is implemented on the distorted image, single down sampled image and double down sampled image as shown in *Figure 1.*

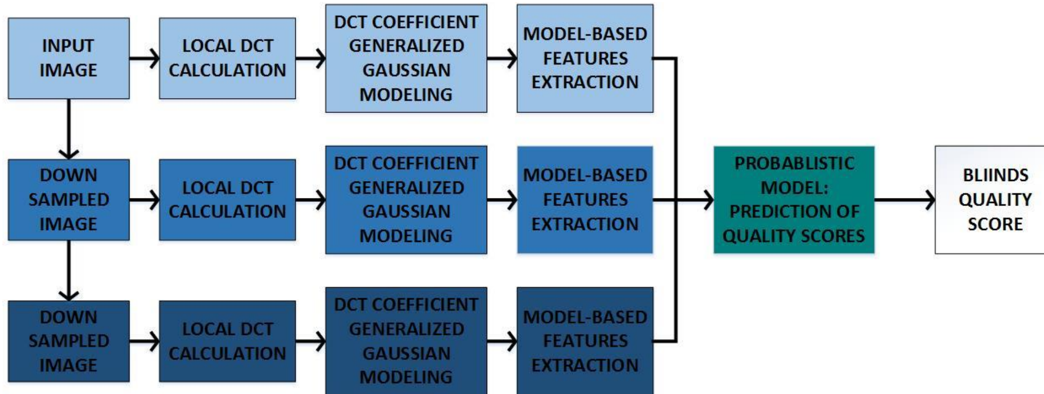

Figure 1. BLIINDS-II algorithm flow(Yadav, A. (2016))

As discussed in detail by Yadav, A. (Yadav, A. (2016)), The BLIINDS-II algorithm is carried out in four distinct steps: -

First, zero padding operation is carried out on the input image, which is then portioned into equally sized blocks of dimension 5x5 pixels, called local image patches. In order to simulate the Human Visual System's (HVS) property of local spatial visual processing, Local DCT is computed of these local image patches to find block DCT coefficients.

In the second step, the non-DC coefficients of each block are applied with a univariate generalized Gaussian density model. The generalized Gaussian density model (Blake and Sekuler 2006). is given by –

$$f(x|\alpha,\beta,\gamma) = \alpha e^{-(\beta|x-\mu|)^{\gamma}}$$

where μ is the mean, γ is the shape parameter, α and β are the normalizing and the scale

parameters given by-

$$\alpha = \frac{\beta\gamma}{2\Gamma(1/\gamma)}$$

$$\beta = \frac{1}{\sigma}\sqrt{\frac{\Gamma(3/\gamma)}{\Gamma(1/\gamma)}}$$

Where o is the standard deviation and Γ denotes the gamma, function given by

$$\Gamma(z) = \int_0^{\infty} t^{z-1}\, e^{-t} dz$$

Specific partitions within each block of local DCT coefficients across different orientations are also applied with this Gaussian model. A generalized Gaussian fit is obtained for each of the orientations and radial sub regions.

In the third step of algorithm, required features are extracted from Gaussian model parameters developed in the second step. Eight features each are extracted from the input image and the two down sampled images mentioned earlier. (Yadav, A. (2016))

In the fourth step, the image quality scores are predicted from the extracted features using a simple Bayesian inference approach. The probability that the distorted image has a certain quality score for given the model-based features is maximized using the Bayesian approach. (Yadav, A. (2016))

CHAPTER 4

GPGPU

Introduction of GPU

The technical definition of a GPU as given by NVIDIA (Graphics Processing Unit (GPU), NVIDIA) is "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second. "Traditionally, GPU is used in rendering images, animations and videos on video screen because of its parallel processing architecture, which allows it to perform multiple calculations at the same time. GPUs can be located in a chipset on the motherboard, located on plug in cards or even in the same chip as the CPU. (What is GPU? 2017)

Introduction of GPGPU

General Purpose computation on Graphic Processing Units(GPGPU) is a term coined by Mark Harris of NVIDIA referring to use of the GPU for general-purpose parallel processing applications rather than just for traditional graphical applications. As compared to the CPU which consists of a few cores optimized for sequential serial processing, the GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores which are capable of very high computation and data throughput (NVIDIA on GPU Computing and the Difference Between GPUs and CPUs)

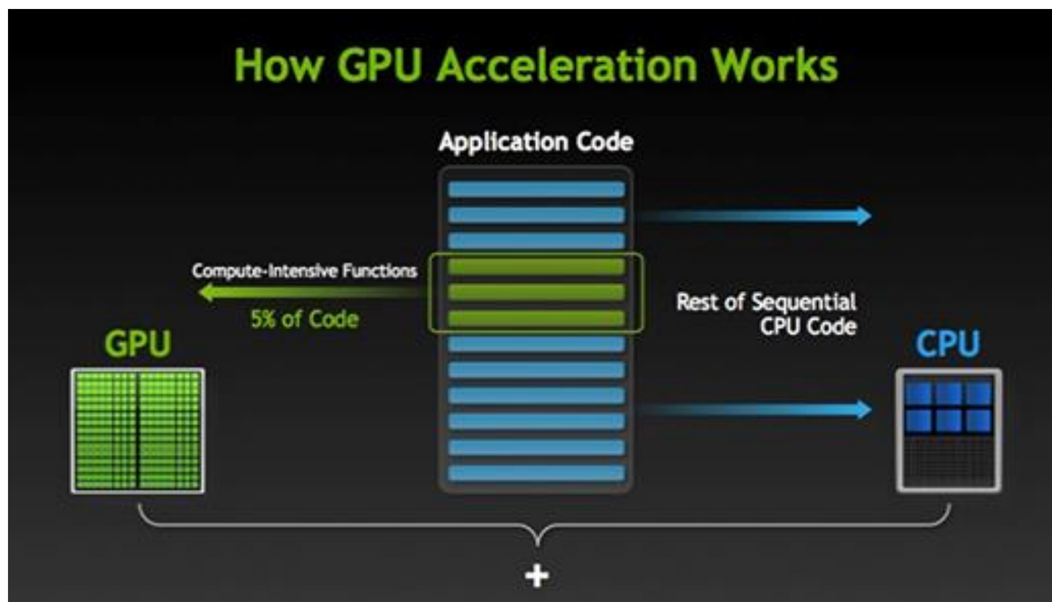

Figure 2. GPU accelerated computing(NVIDIA on GPU Computing and the Difference Between GPUs and CPUs)

Consider we have an application code which needs to be optimized as shown in *Figure 2*. In GPU-accelerated computing, the compute intensive and parallel portion of code is ported to the GPU, while the rest of the serial code still runs on the CPU. The GPU can process this parallel portion of code on its many-core architecture much faster as compared to CPU. From user's perspective, great speedup in application is observed. (NVIDIA on GPU Computing and the Difference Between GPUs and CPUs)

Examples of GPU acceleration

In recent times, many applications from diverse fields like bioinformatics, deep learning, computational finance, molecular dynamics, computer vision and imaging, medical imaging have been accelerated using GPGPU technology. Few examples of such applications highlighted in GPU-Accelerated Applications catalog (GPU-Accelerated Applications) released by NVIDIA are discussed below -

In the field of seismography, Seismic imaging is a technique often used for analysis of Tsunami waves. Reverse Time Migration (RTM) is an advanced migration method for seismic depth imaging. Until recently, RTM's widespread use was severely hindered by the enormous computing resources required to process the data. Acceleware, a company providing Parallel Computing software solutions was able to clear this computational bottleneck and achieving about 5x speed up, by designing a GPU accelerated library called AxRTM™, that can be integrated into an existing seismic processing framework (Reverse Time Migration).

In the field of Molecular dynamics, NAMD (Nanoscale Molecular Dynamics) is a production-quality molecular dynamics application designed for high-performance simulation of large biomolecular systems, developed by University of Illinois at Urbana-Champaign (UIUC). NAMD is distributed free of charge with binaries and source code. The latest version, NAMD 2.11, typically runs 7x faster on NVIDIA GPUs over CPU-only systems. (NAMD Introduction | GPU Accelerated Applications.)

MSc Software is a company providing softwares like Nastran in field of structural mechanics. MSC collaborated with Nvidia to deliver the power of GPU computing for Nastran customers. Available in the latest release of MSC Nastran 2013 (and in MSC Nastran 2012), NVIDIA GPU acceleration enables faster results for more efficient computation and job turnaround times, delivering more license utilization for the same investment. Recent performance studies conducted together with MSC engineers demonstrated that the Tesla K20X (and Tesla K20) GPU acceleration of Intel Xeon (Sandy Bridge) CPUs resulted in speed-ups in the range of 3-6X with a single GPU over a serial run and in the range of 1.5-2X with 2 GPUs over a 8 core DMP run. (MSC Nastran 2013 Acceleration on NVIDIA Kepler)

There are even some GPU accelerated libraries which find application in diverse fields. One such famous library is ArrayFire is a comprehensive, open source function library with interfaces for C, C++, Java, R and Fortran. It integrates with any CUDA application, and contains an array-based API for easy programmability. It contains excellent GPU implementations of hundreds

of matrix, signal, and image processing routines that enable it outperform CPU libraries like IPP, MKL, Eigen, Armadillo, and more. It is designed for use on the full range of systems, from single GPU systems to large multi-GPU supercomputers and can provide speedups of around 2-20x. The ArrayFire Library is fully open source, and can be accessed via the ArrayFire website at www.arrayfire.com. (ArrayFire. (2017, April 25))

It is evident from above examples that GPGPU is an efficient method for optimizing data parallel applications.

In 2007, NVIDIA released the initial public beta of their proprietary CUDA (Compute Unified Device Architecture) development framework. CUDA allows researchers and professionals from various fieldsto use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. In 2008, another parallel computing platform called OpenCL (Open Computing Language) was released by ATI/AMD which implemented a non-proprietary, committee-backed specification for parallel devices, including GPUs, multicore CPUs. (History of CUDA, OpenCL, and the GPGPU)

The GPGPU based implementation of BLIINDS-II algorithm is based on CUDA. Hence, in the next chapter we will be discussing CUDA in detail.

CHAPTER 5

CUDA PROGRAMMING MODEL

The CUDA programming model is a heterogeneous model where both the CPU and GPU come into play. In CUDA, CPU and its memory is referred to as host, whereas GPU and its memory is referred to as device. CUDA provides extension to C++ by providing a functionality to programmer to define C++ functions called kernels. These kernels, when called are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. The CUDA code is run on host and it can manage memory on both the host and device and also launch kernels which are executed on the device. (An Easy Introduction to CUDA C and C. (2017, January 25);

Typical Sequence of CUDA Programing

The traditional sequence of operations (An Easy Introduction to CUDA C and C. (2017, January 25) in a CUDA C++ program is –

1. Declaration of Host and Device memory

The first step is initialization of host and device pointers which point to respective host and device memory. These pointers are conventionally termed as h_in and d_in.

2. Allocation of Host and Device memory

The memory pointed by h_in and d_i n is allocated in host and device of appropriate size. The host memory is allocated using malloc in typical fashion. And device memory is allocated using cudaMalloc function from the CUDA runtime API as shown below.

```
void* malloc (size_t size);
cudaMalloc (void ** devPtr, size_t size)
```

3. Initialization of host data

Data which needs to me manipulated is stored in host memory.

4. Transferring relevant data from host to device

To transfer the data from host memory to device memory cudaMemcpy function from the CUDA runtime API as shown below.

```
cudaMemcpy  (void* dst, const void* src,size_t count,enum cudaMemcpyKind kind)
```

cudaMemcpy works just like the standard C memcpy function. It just takes a fourth argument which specifies the direction of the copy. For host to device data transfer this argument is cudaMemcpyHostToDevice, to specify that the first (destination) argument is a device pointer and the second (source) argument is a host pointer.

5. Start Profiling and timers

CUDA application can be profiled using Nvidia Visual Profiler provided as a part of CUDA toolkit. The cudaProfilerStart(void) is the function that should be used to mark the starting point of profiling for the profiler. Also, CPU or GPU timer is initialized at this point.

6. Execution of one or more kernels

In CUDA, kernels are defined using __global__ declaration specifier. The variables defined within device code are assumed to reside within device code and do not need to be specified as device variables. The arguments of a kernel are passed by value in the same way as the function arguments are passed by value by default in C/C++. The kernel definition in device code is as shown below –

```
__global__ void Func(float* parameter);
```

The syntax used to call a kernel is as shown below. The syntax of triple chevrons is called the execution configuration, this is used to set the number of threads that execute the kernel in parallel. In CUDA, "there is a hierarchy of threads in software which mimics how thread processors are grouped on the GPU" (An Easy Introduction to CUDA C and C. (2017, January 25). In CUDA programming model the kernel is launched as a grid of blocks. In the execution configuration –

i. The first argument specifies the number of thread blocks in the grid

ii. The second argument specifies the number of threads in a thread block.

iii. The third argument is of type size_t and specifies the number of bytes in shared memory that is dynamically allocated per block for this call-in addition to the statically allocated memory. This is an optional argument which defaults to zero. [Cuda guide]

iv. The fourth argument is of type cudaStream_t and specidies the associated stream. This is an optional argument which defaults to zero.

```
Func<<< Dimension of grid, Dimension of block,dynamically allocated shared memory, cudastream >>>(parameters);
```

The threadidx is defined as “a 3-component vector, in such a way that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This programming structure provides a natural way to invoke computation across the elements in a domain such as vector, matrix or volume” . (CUDA C Programming Guide. (2017).) Kernel calls are asynchronous from CPU point of view. That is the

control is returned back to CPU as soon as the kernel is launched. And thus, it can launch other kernel or CPU function as soon as the first kernel is launched. Whereas from GPU point of view, kernels are executed in the order they are called as they enter into default stream and execute serially (kernels can run in parallel if it has been specified to execute them in different streams). *Figure 3.* shows how a kernel is executed

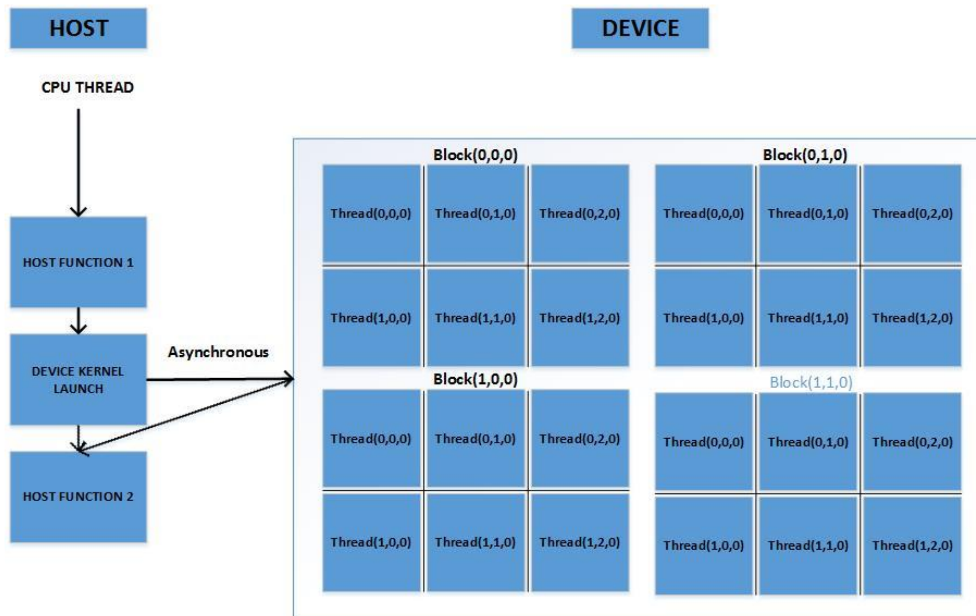

Figure 3. Kernel execution policy (Kannan, V. (2016).)

7. Stop Profiling and timers

The instruction cudaProfilerStop(void) is given to mark the end of profiling to profiler. Also, the CPU and GPU timers are stopped. The profiling data and timings recorded are analyzed.

8. Transfer results from host to device.

To transfer the data manipulated using kernels from device memory to host memory, cudaMemcpy function is used as before. The fourth argument in this case is cudaMemcpyDeviceToHost, to specify data is being transferred from device to host.

9. Reset device

To reset the device, it is necessary to free up any memory allocated. The host memory is freed using free() function in C. And the device memory allocated with cudaMalloc(), is freed by calling cudaFree() function of CUDA runtime API.

CUDA compilation process

CUDA toolkit provides NVCC (NVIDIA CUDA compiler) for compilation of CUDA code. CUDA project typically is made of two source files a .cu and a .cpp file. When the compilation begins the header files(#include) are initially expanded and comments are removed by the preprocessor. The .cu file is processed using cudafe and nvopencc(open64 based open source compiler provided by NVIDIA) (Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., & Aamodt, T. M. (2009, April)).to intermediate pseudo assembly file of extension .ptx. This .ptx file is assembled into native CUDA binary(cubin bin) using ptx assembler(ptxas).

This cubin binary is merged with the host C++ code and compiled into a single executable file. This file is then linked with the CUDA runtime API library(libcuda.a). In the final stage, the executable calls the CUDA Runtime API to initialize and invoke kernels onto GPU using NVIDIA CUDA driver. (Kannan, V. (2016).)

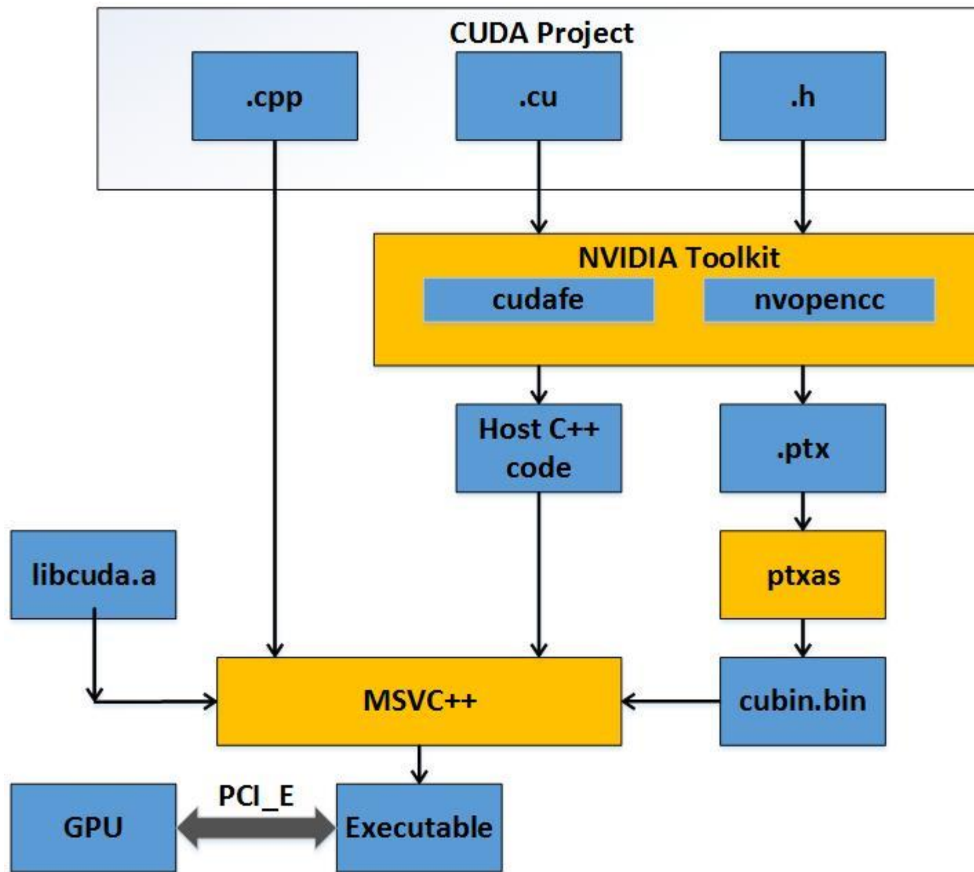

Figure 4. CUDA project compilation (Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., & Aamodt, T. M. (2009, April))

CHAPTER 6

CUDA MEMORY MODEL

Introduction

CUDA enabled GPUs have different types of memories. These memories have their own properties such as access latency, address space, scope, and lifetime. The different types of memories are register, shared, local, constant and texture memory.

All modern GPUs have fully coherent L2 cache, which is smaller than CPU L2 cache but has higher bandwidth. Streaming multiprocessors of high end GPUs have their own L1 cache. These L1 caches are also smaller than CPU L1 cache but have higher bandwidth. It is essential to note that L1 caches in GPUs are not coherent. (CUDA Memory and Cache Architecture.) Comparison of CPU and GPU caches is shown in Table 1.)

Table 1.

Comparison of CPU and GPU cache (CUDA Memory and Cache Architecture.)

	CPU	GPU
Memory	6->64GB	768->6GB
Memory Bandwidth	24->32GB	100-200Gb/s
L2 cache	8->15MB	512->768KB
L1 cache	256->512KB	16-48KB

Types of GPU memories

We will now discuss the types of memories –

1. Registers

Scalar variables that are declared in the scope of a kernel function and do not have any attribute associated with it are stored in register memory by default. Number of registers available per block is limited, but register memory access is very fast.

Each thread has its own set of private register variables. This means, threads in the same block gets private versions of each register variable. Register variables have same lifetime as that of thread. Each invocation of the kernel

function must initialize the variable each time it is invoked. Variables declared in register memory can be both read and written inside the kernel.(CUDA Memory Model. (2013, November 26).)

2. Local memory

Some automatic variables can cause local memory access. Automatic variables that the compiler is likely to place in local memory are:

when it cannot be determined if arrays are indexed with constant quantities.

If register space is consumed in excess by large structures or arrays.

In case of register spilling(if more registers than available are used)

To know if a variable is placed in local register, PTX code(pseudo code) needs to be analyzed. Local memory exhibits high latency and low bandwidth because it resides in device memory. Local memory accesses are always cached in L1 and L2 for devices with compute capability 2.x and 3.x, and in L2 cache for compute capability 5.x and 6.x. (CUDA C Programming Guide. (2017).)

3. Constant memory

Variables need to be given __constant__ attribute to be declared in constant memory. Constant variables must be declared in global scope. Constant variables share the same memory banks as global memory(device memory). There is limited amount of constant memory that can be declared which is equal to 64KB on all compute capabilities.

Variables that are decorated with the __constant__ attribute are declared in constant memory. Like global variables, constant variables must be declared in global scope (outside the scope of any kernel function). Constant variables share the same memory banks as global memory (device memory) but unlike global memory, there is only a limited amount of constant memory that can be declared (64KB on all compute capabilities). Constant variables have faster access latency as compared to global memory. Constant memory has a lifetime of the application. It can be accessed by all the threads of all kernels with no change in the value across kernel invocations. (CUDA Memory Model. (2013, November 26).)

4. Texture memory

Texture memory is read from kernels using the device functions described in Texture Functions. The process of reading a texture calling one of these functions is called a texture fetch. Each texture fetch specifies a parameter called a texture object for the texture object API or a texture reference for the texture reference API.

Like constant memory, texture memory is on-chip and can provide higher effective bandwidth by reducing memory requests to off-chip DRAM. texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality. (Gupta, N.)

5. Global memory

Variables with __device attribute and which are declared in global scope (outside the scope of kernel function) are stored in global memory. The main abstraction by which CUDA kernels read or write to device memory is the

global memory. Global memory can be accessed directly by CUDA kernels using device pointers.

Global memory has the highest access latency (~100 times slower than shared memory). But global memory is the largest of all memories available(up to 6GB). Global memory can be allocated and deallocated using cudaMalloc and cudaFree functions and also can be read from and written to by host using cudaMemcpy and function of CUDA runtime API.

Global memory has the lifetime of the application and is accessible to all threads of all kernels. Access to global memory cannot be synchronized across different blocks. The global memory can only be synchronized by splitting the problem into different kernels and synchronizing on host between kernel invocations.

On devices with compute capability 3.0x, the reads and writes are not cached.

On devices with compute capability higher than 2.0, reads from global memory are cached. But the writes to global memory will invalidate the cache, thus cancelling the benefit of cache.

Due to high latency of global memory, it becomes imperative to take care of access patterns in order to reduce the number of global memory transactions for a piece of data.(CUDA Memory Model. (2013, November 26);Wilt, N. (2013).)

6. Shared memory

Variables with attribute __shared__ are stored in Shared memory. Shared memory is lot faster (~100 times) than global memory. This can be accounted to the fact that it is present on-chip. Each Streaming Multiprocessor has a limited amount of Shared memory.

Shared memory must be declared within the scope of the kernel function. It has lifetime of a thread block. Shared memory can be both read from and written to within the kernel. For synchronization of shared memory access, block synchronization is necessary which can be achieved by using __syncthreads() barrier function inside the kernel function. Since access to shared memory is faster than accessing global memory, it is more efficient to

copy data from global memory to shared memory to be used within the kernel.(CUDA Memory Model. (2013, November 26))

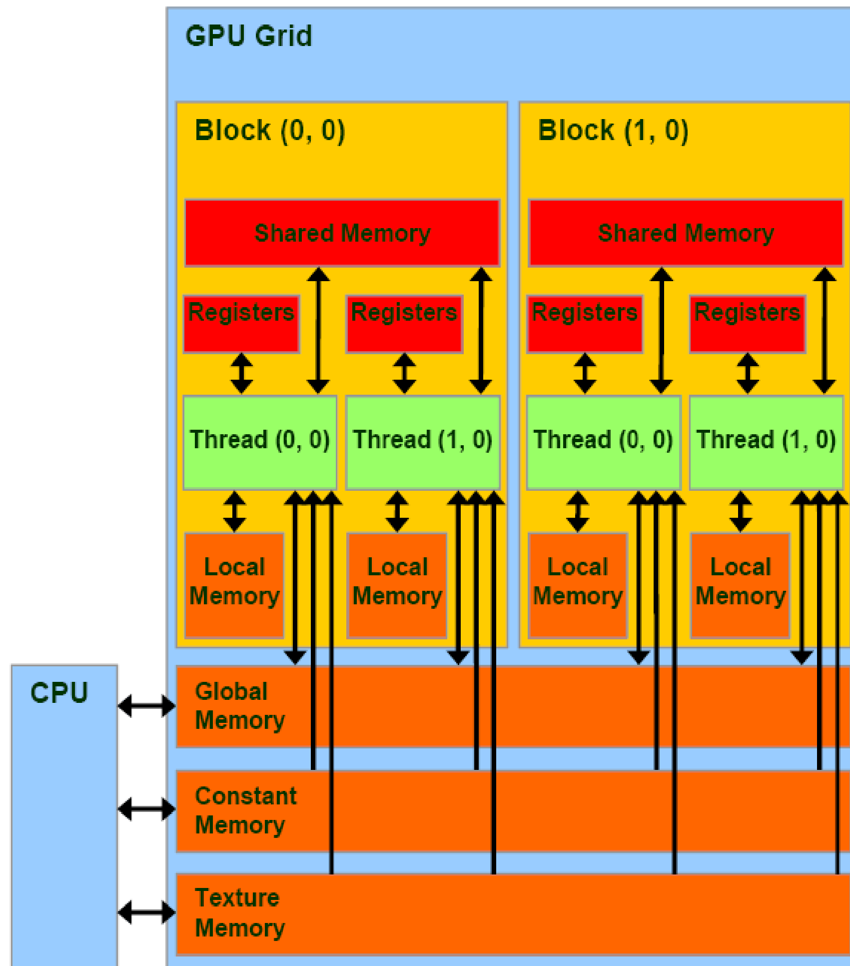

Figure 5. GPU Memory hierarchy. (CUDA Memory Model. (2013, November 26))

Following Table 2. shows properties of these different memory types: -

Table 2.

Properties of different types of CUDA memories (CUDA C Best Practices Guide.)

Memory	Located	Cached	Access	Scope	Lifetime
Register	Cache	n/a	Host: None Kernel: R/W	Thread	Thread
Local	Device	1x -No 2x -Yes	Host: None Kernel: R/W	Thread	Thread
Shared	Cache	n/a	Host: None Kernel: R/W	Block	Block
Global	Device	1x -No 2x -Yes	Host: R/W Kernel: R/W	Application	Application
Constant	Device	Yes	Host: R/W Kernel: R	Application	Application
Texture	Device	Yes	Host: R/W Kernel: R	Application	Application

CHAPTER 7

DATA TRANSFER

Introduction

Now that we know the basic essentials of CUDA programming, it is important to know how we can optimize a CUDA program. Optimization of CUDA program can be achieved by employing various techniques like (CUDA C Best Practices Guide.) –

1. Instruction level parallelism
2. Coalescing memory accesses
3. Maximizing arithmetic intensity
4. Optimizing Data transfers
5. Stream synchronization
6. Dynamic parallelism

and so on

In this chapter, we will discuss data transfers in CUDA programing and explore methods for optimizing it.

Types of Optimizations in Data Transfers

1. Global memory data access and its optimization–

Global memory, of device resides in DRAM, for transfers between host and device as well as for data input to and output from kernels. Global memory has high latency(400-800cycles) because of its presence in off-chip DRAM. But most of the data accesses begin in global memory. Thus, optimizing global memory bandwidth is fundamentally necessary. (Tsutsui, S., & Collet, P. (2016).)

During execution, there is a finer grouping of threads into warps. Streaming multiprocessors execute instructions for each warp in SIMD (Single Instruction Multiple Data) fashion with warpsize of 32 threads. Grouping of threads into warps is not only relevant to computation but also to global memory accesses. To minimize DRAM bandwidth, it is imperative to coalesce global memory loads and stores issued by threads of a warp into as few transactions as possible. Global memory can have maximum throughput of up

to 177GB/s. (How to Access Global Memory Efficiently in CUDA C/C Kernels. (2014, June 09); Farber, R. (2012))

2. Caching data to shared memory from global memory

As we know, shared memory is nearly 100 times faster than global memory on account of its presence on-chip. Hence, it can be used to eliminate redundant accesses to global memory.

Shared memory is divided into equally sized memory modules(banks) that can be accessed simultaneously, which helps in concurrent accesses for higher memory bandwidth. Thus, n distinct memory banks can be accessed simultaneously to service n respective addresses, giving an effective bandwidth, which is n times higher than that of a single bank. However, if multiple addresses of a memory request map to the same memory bank, the accesses are serialized. (Using Shared Memory in CUDA C/C++. (2014, July 21; Bank conflicts in shared memory in CUDA) The most important aspect of shared memory is the facilitation of reciprocity between threads in a block. Shared memory can be used as a buffer for data from global memory, when

same data is accessed multiple times by multiple threads from global memory. Data can be stored in shared memory in a coalesced pattern from global memory and then reordered, thus avoiding uncoalesced access to global memory. Non-sequential or unaligned accesses by a warp in a shared memory has the only disadvantage of memory bank conflicts. (CUDA C Best Practices Guide.) Many CUDA libraries are optimized in a way that they make use of shared memory to optimize data transfer involved in the algorithm.

3. Warp shuffle instruction

In CUDA programming, while invoking a kernel, we have to specify how many threads we are going to launch in a block and how many such blocks are there.

Every such kernel is executed on SM(Streaming Multiprocessors). These blocks are mapped on the SMs as they execute. The memory on-chip is shared by all the threads in the block as they are on same Streaming multiprocessor. But, from micro-architectural point of view, during execution the threads in a block are divided into warps having 32 threads each. The

warps in each block exhibit SIMD execution (Single Instruction Multiple data). (CUDA C Programming Guide. (2017)) As we know, memory accesses take more cycles than compute operations. So, if there is a memory access to any thread in a warp, SM switches to next warp. Thus in such case, SM never remains idle. Thus, programmer has to design the code considering how the warps are mapped to the SMs and that there is minimum compute latency. GPUs based on Kepler architecture, have a new shuffle instruction, which allows data within a warp to be shared by threads. On earlier hardware, we needed to write data to shared memory, synchronize it, and then reading the data back from shared memory. On other hand, Kepler's shuffle instruction (SHFL) enables a thread to directly read a register from another thread in the same warp (32 threads). (Whitepaper | NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110)

The PTX instruction relating to shuffle intrinsic is given as-

```
shfl.mode.b32 d[|p], a, b, c;
```

where, d- destination register, |p- optional destination predicate, a- source register, b- lane/offset/mask, c- Bound. (CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

With the Shuffle instruction, threads within a warp can read values from other threads in the warp in all possible ways. Shuffle supports arbitrary indexed references – i.e. any thread reads from any other thread. CUDA intrinsics of useful shuffle subsets are available including (offset up or down by a fixed amount) and XOR "butterfly" style permutations among the threads in a warp.

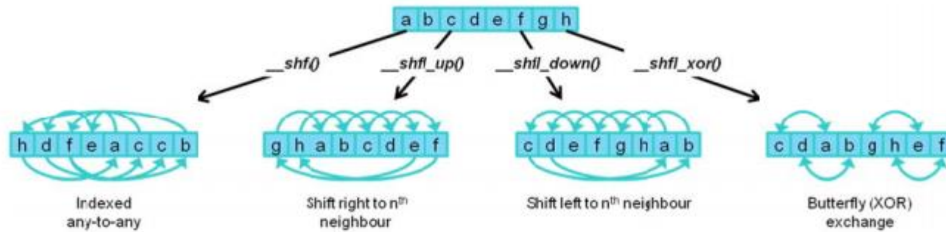

Figure 6. Types of shuffle instructions (Whitepaper | NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110)

In warp-shuffle operation, store and load operation is carried out in a single step, which offers a performance advantage over using shared memory.

Shuffle also reduces the amount of shared memory needed per thread block, since data exchanged at the warp level never needs to be placed in shared memory. (CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

Warp shuffle instruction was announced at GTC (GPU tech Conference) 2013 CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10)), in a talk by Julien Demouth. In this talk, he talks about the performance experiments he carried out to verify performance boost by shuffle. As, a part of his first experiment, he conducted experiment of transferring data of thread which is on right side of the given thread. He conducted this using shuffle instruction, using shared memory and using shared memory without synchronization. He launched 26 blocks of 1024 threads on Kepler K20. The results he obtained for f32(floating point 32-bit numbers) numbers are as shown below –

Figure 7. Results of performance experiment. (CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

He also carries out warp level reduce and bitonic sort operations on f32 numbers using warp shuffle instruction. The results he obtained were as follows –

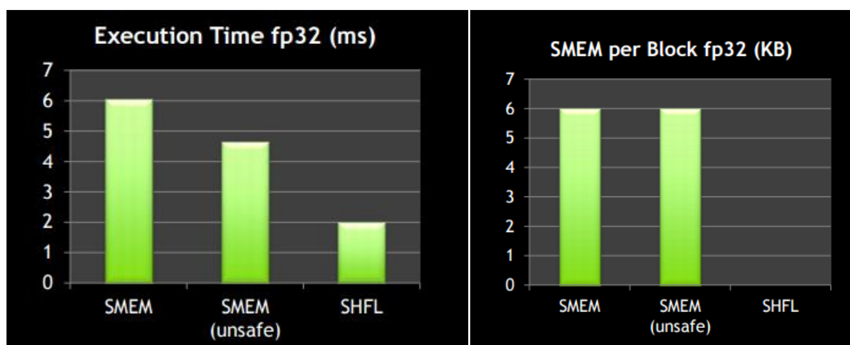

Figure 8. Results of reduce operation. (CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

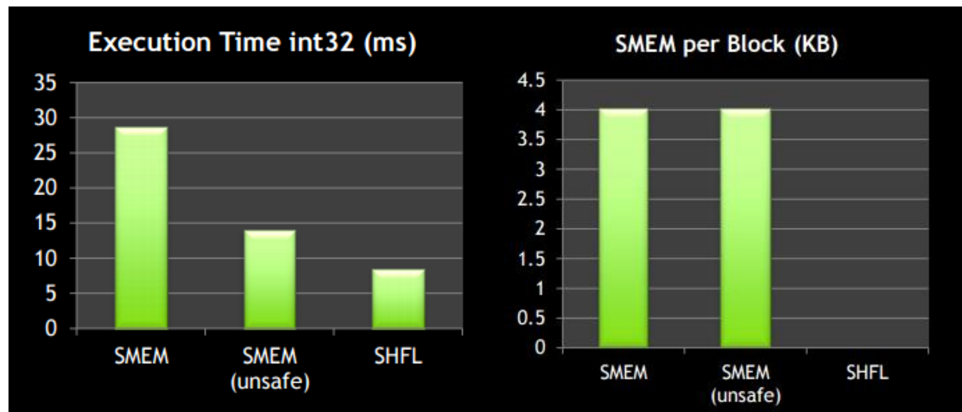

Figure 9. Results of bitonic sort operation. (CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

From the above experiments, it was concluded that the use of shuffle instruction always proved faster than using shared memory, even when the shared memory was used without synchronization. Also, it must be noted that the shuffle operation itself does not require any shared memory, and can thus prove useful in cases where occupancy is limited by Shared memory usage.

CHAPTER 8

EXPERIMENT: REDUCE ALGORITHM

To analyze how the warp shuffle instruction hides the latencies instigated because of global and shared memory data transfer operations, a sample program computing Reduction algorithm on a 1-D Array was performed.

Reduce algorithm

A reduction algorithm extracts a single value from an array of values or a set of arrays of values. Reduced values include the sum of an array's elements, their maximum value, their minimum value, their average value, etc. (Szalwinski, C.) In the aforementioned program, the sum of all the elements of the array was calculated using reduction algorithm.

A serial reduction algorithm moves through the elements of an array or set of arrays one element at a time. *Figure 10.* shows how the serial reduction algorithm works. In a reduction algorithm for an array, we can store the results of each round in the element with the lower index. In the end, the final result will be in the first element of the array (Szalwinski, C.). Serial

reduction algorithm has a work complexity of O(n) as the amount of work done is linear with respect to the size of the array. And also, a step complexity of O(n) as no. of steps is linear with respect to the size of array (Owens john and luebke dave).

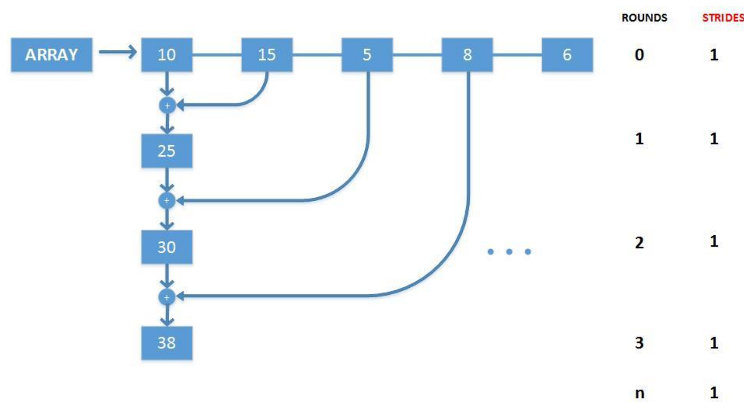

Figure 10. Serial implementation of Reduce algorithm(Szalwinski, C.)

A parallel reduction algorithm moves through the elements of an array or set of arrays concurrently as shown in *Figure 11.*

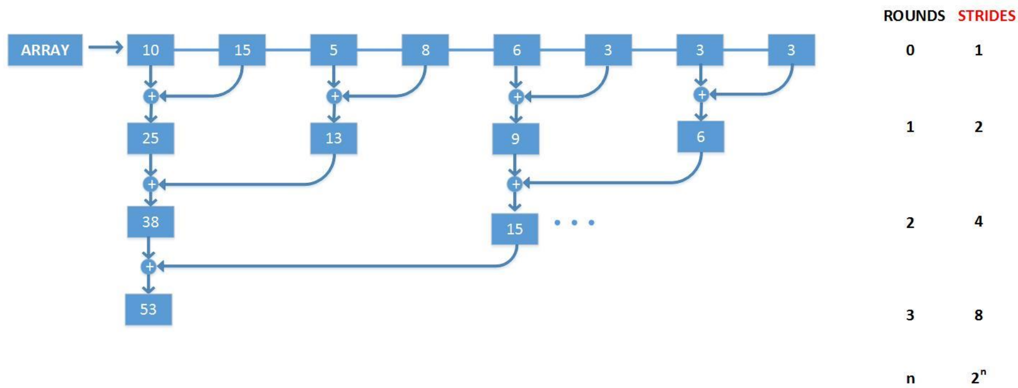

Figure 11. Parallel implementation of reduce algorithm (Szalwinski, C.)

As we move from round to round in a reduction, the stride doubles. The stride is the difference between the indices of the two elements being processed. In any round, we only process those elements with indices that are multiples of the stride for that round. The stride grows as a power of 2 as we move from round to round (Szalwinski, C.). The work complexity of this parallel reduction algorithm is O(n). Whereas, its step complexity is O(log2n) as the number of steps is a logarithmic function of the size of the array (Owens john and luebke dave).

Our sample program is based on approach discussed by Owens John and Luebke Dave in Fundamental GPU Algorithms class in Intro to Parallel Programming course of Udacity and NVIDIA(Owens john and luebke dave) and by Luitjens J. in his post on “Faster Parallel Reductions on Kepler“ in Parallel for all, a GPU Computing developer blog by NVIDIA(Faster Parallel Reductions on Kepler. (2016, June 22)).

Modes of implementation

In the sample program, the reduction algorithm is implemented in three modes –

Method 1. Using only Global memory.

Method 2. Using Shared memory.

Method 3. Using Warp shuffle instruction.

The details of hardware used for above implementation is given in Table 3 and Table 4.

Table 3.

Kepler GK110 Specifications.

Kepler GK110 Base Core Clock-Rate	889 MHz
Number of Cores	2,880
Computational Throughput	4290 GFLOPS Single Precision 1430 GFLOPS Double Precision
Memory Clock (Transfer) Rate	7000 Gbps
Memory Bus Support	PCIe 3.0 x16
Memory Bus Width	384 bits
Memory Size	12 GB
Memory Interface	GDDR5
Global Device DRAM Bandwidth	336 GB/s
Memory Controllers	64-bit (Quantity 6)
Streaming Multiprocessors (SM)	15 (192 Cores per SM)
L1-Cache (per SM)	64 KB

L2-Cache	1.5 MB
32-bit Registers (per SM)	65,536
Maximum Registers per Thread	255
Threads per Warp	32
Maximum Warps (per SM)	64
Maximum Thread Blocks (per SM)	16
Maximum Threads (per SM)	2,048
Maximum Threads per Thread Block	1,024
Pixel Fill-Rate	42.7 (GP/s)2
Texture Fill-Rate	213 (GT/s)2

Table 4.

Test System Specifications

CPU	Intel Xeon E5 1620 v2 @3.70 GHz
Microarchitecture	Ivy Bridge
No of Cores	4
No of Threads	8
L1 Cache	64 KB per core
L2 Cache	256 KB per core
L3 Cache	10 MB shared
Operating System	Windows 10 (64-bit)
Compiler	Visual Studio 2013, CUDA 7.5
GPU	NVidia Tesla K40c

Method 1. Using only Global memory

i. An array consisting of 2048 floats is copied from Host(CPU memory) to Device(GPU global memory) using the cudaMemcpy function as shown below.

```
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
```

ii. For block level reduction operation, kernel 'global_reduce_kernel' is launched as shown below with block dimension – 1024 threads and grid dimension – 2 blocks (Array size/block dimension). And d_in and d_intermediate are passed as parameters. Here, d_in is the device memory where input array is stored and d_intermediate is the device memory where the output of block level reduction is stored.

```
global_reduce_kernel << <blocks, threads >> >(d_intermediate, d_in);
```

iii. Each block operates on 1024 elements of the Array stored in d_in. Initially, each thread adds the elements of the array with a stride of 512(Block dimension/2). So, the first element, d_in[0] is added to d_in[512], d_in[2] is added to d_in[513] and so on till d_in[511] is added to d_in[1023]. And the

result of the operation is stored in the element with a lower index. This iteration is repeated by varying stride by a factor of 2, till the final result of reduction operation is stored in d_in[0]. This operation is achieved by using for-loop as shown below.

```
for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
{
    if (tid < s)
    {
        d_in[myId] += d_in[myId + s];
    }
    __syncthreads();        // make sure all adds at one stage are done!
}
```

Figure 12. represents the parallel reduce algorithm implemented on an array of 8 elements using global memory.

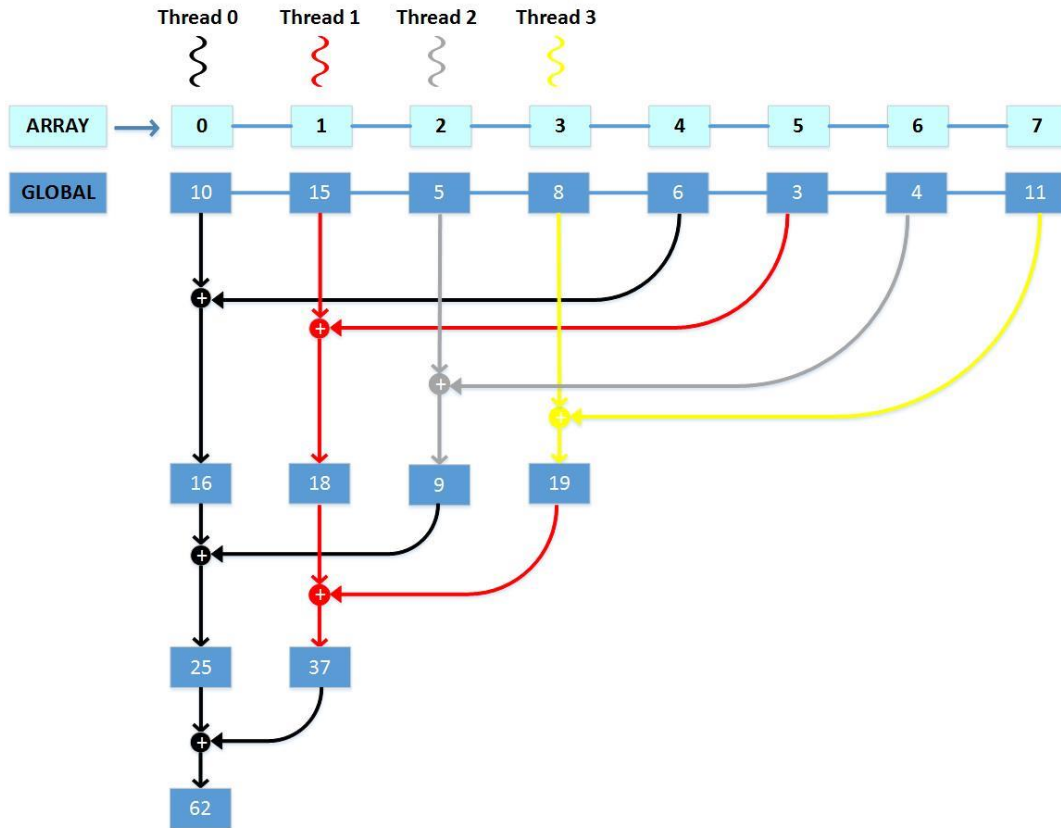

Figure 12. Parallel reduce algorithm using only global memory (Szalwinski, C.)

iv. The final result of every block level reduction is stored in d_intermediate.

```
if (tid == 0)
{
    d_out[blockIdx.x] = d_in[myId];
}
```

v. The ‘global_reduce_kernel’ kernel is launched for a second time for running reduce operation on elements in d_intermediate. It has a grid dimension of 1 block and block dimension of 2 threads(block dimension of the first kernel). Thus, the values stored in d_intermediate are reduced and the output value is obtained in d_intermediate[0]. This value is transferred to d_out[0].

```
global_reduce_kernel << <1, 2 >> >(d_out, d_intermediate);
```

vi. The output of the reduction operation stored in d_out is transferred to h_out in host memory(CPU memory) using the cudaMemcpy function as shown below.

```
cudaMemcpy(&h_out, d_out, sizeof(float), cudaMemcpyDeviceToHost);
```

Time taken for algorithm using only Global memory for an array size of 2048 – 0.015041ms.(average of 100 iterations)

Method 2. Using Shared memory

i. The Array of size 2048 is copied from Host memory(h_in) to device memory using cudaMemcpy function.

ii. The first kernel launch is similar to that shown in Method 1.

iii. The only difference in this kernel is that the array is copied from global memory to shared memory before implementing reduce operation on it as shown below.

```
sdata[tid] = d_in[myId];
```

iv. The reduce operation with strides is similar to that of Method 1. using for-loop as shown below. The final result is in sdata[0]. This data is copied from sdata(Shared memory) to d_intermediate in Global memory.

```
for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
{
    if (tid < s)
    {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();        // make sure all adds at one stage are done!
}
if (tid == 0)
{
    d_out[blockIdx.x] = sdata[0];
}
```

Figure 13. shows the reduce operation using shared memory on an Array of size = 8.

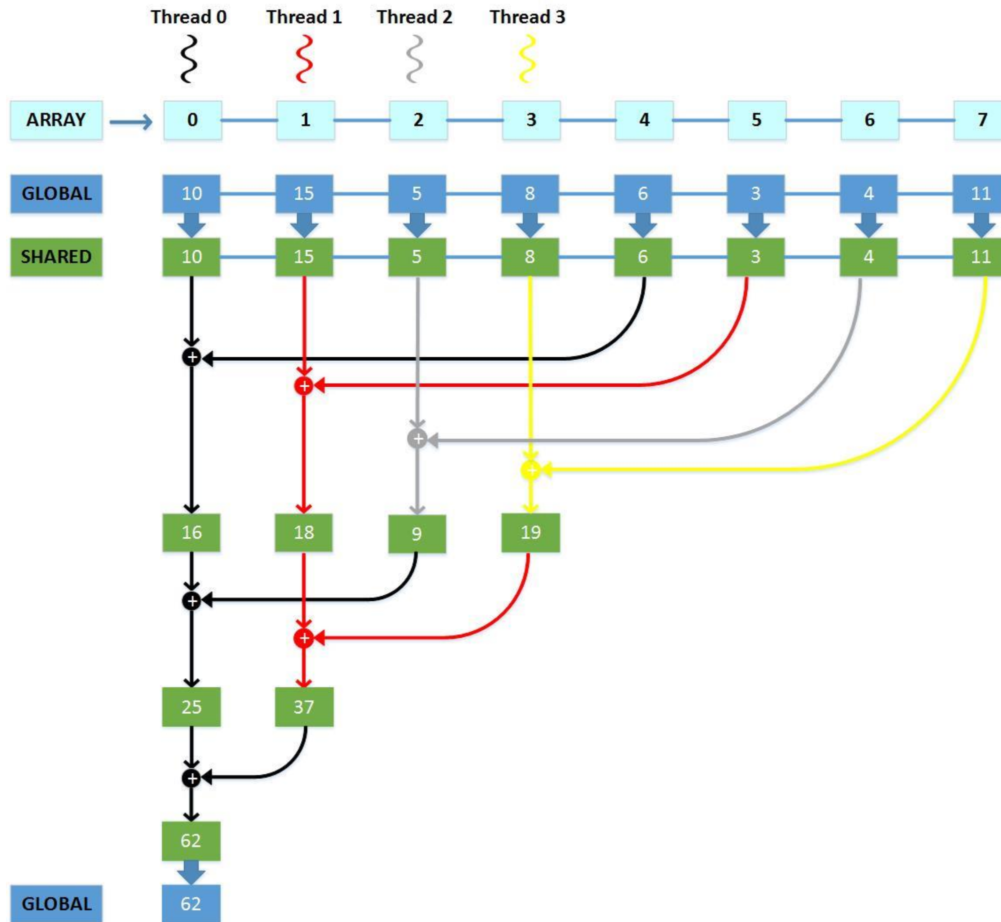

Figure 13. Parallel reduce algorithm using shared memory(Szalwinski, C.)

v. The second kernel is launched in similar fashion as in Method 1 and utilizing shared memory.

vi. Finally the output is transferred to host memory (CPU memory) from d_out in GPU memory using cudaMemcpy function.

Time taken for algorithm using Shared memory for an array size of 2048 – 0.012597ms.(average of 100 iterations)

Method 3. Using Warp shuffle instruction (CUDA Pro Tip: Do The Kepler Shuffle.(2016, January 10)):

i. The Array of size 2048 is copied from Host memory(h_in) to device memory using cudaMemcpy function as before.

ii. In this implementation of reduction alogorithm, __shfl_down() is used. __shfl_down() calculates a source lane ID by adding delta to the caller’s lane ID (the lane ID is a thread’s index within its warp, from 0 to 31). The value of var held by the resulting lane ID is returned: this has the effect of shifting var down the warp by delta lanes.

```
int __shfl_down(int var, unsigned int delta, int width=warpSize);
```

Figure 14. shows how the values are shifted down by 2 threads.

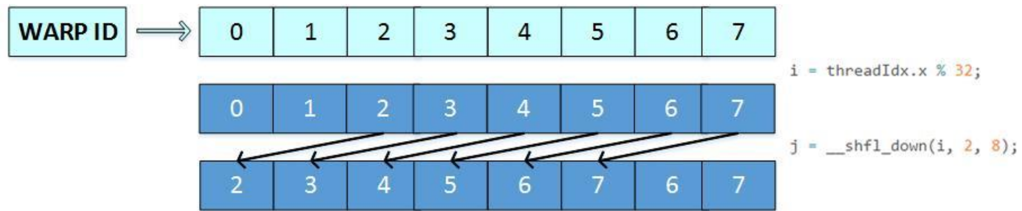

Figure 14. __shfl_down instruction (CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

iii. Using shuffle down instruction we can build a reduction tree as shown below. *Figure 15.* shows how we can use shuffle down to build a reduction tree for 8 elements.

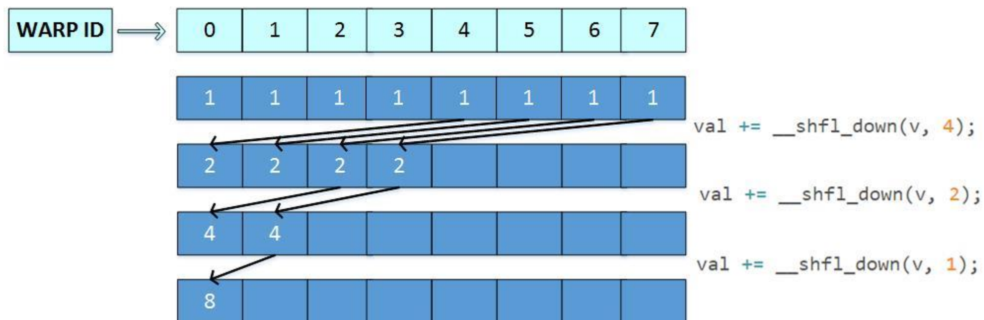

Figure 15. reduction using __shfl_down instruction(CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10))

After executing the three reduction thread 0 has the total reduced value in its variable v.

iv. Using the warpReduceSum function we can now easily build a reduction across the entire block. To do this we first reduce within warps. Then the first thread of each warp writes its partial sum to shared memory. Finally, after synchronizing, the first warp reads from shared memory and reduces again. []

v. For reducing the reduced values in blocks we need to synchronization across the grid, and that requires breaking our computation into two separate kernel launches. The first kernel generates and stores partial reduction results, and the second kernel reduces the partial results into a single total. We can do both steps with the following kernel code.

```
__global__ void deviceReduceKernel(int *in, int* out, int N) {
  int sum = 0;
  //reduce multiple elements per thread
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    sum += in[i];
  }
  sum = blockReduceSum(sum);
  if (threadIdx.x==0)
    out[blockIdx.x]=sum;
}
```

vi. The final result is then copied from device global memory to host memory(CPU memory) using the cudaMemcpy function.

Time taken for algorithm using shuffle instruction for an array size of 2048 – 0.010982ms.(average of 100 iterations)

It is evident from above results that the implementation of reduction algorithm using shuffle instruction(Method 3) is the faster than Method 1 and Method 2. The faster execution of Method 2 than Method 1 was expected because shared memory latency is roughly 100x lower than uncached global memory latency[ll forum]. And the faster execution implementation using warp-shuffle instruction(Method 3) than Method 2 was because of the less

shared memory transactions achieved on account of use of warp level shuffle of data.

Analysis of Results for different Array sizes –

- Array sizes are varied from – 1024 to 131072(1024*128).
- 100 iterations are ran of the code and its average is calculated.

Table 5.

Reduce algorithm for varied array size

Array Size(floats)	Method 1	Method 2	Method 3
1024	0.011045	0.012314	0.014524
1024*32	0.013156	0.017797	0.022619
1024*64	0.015645	0.023119	0.029562
1024*128	0.020319	0.033254	0.043313

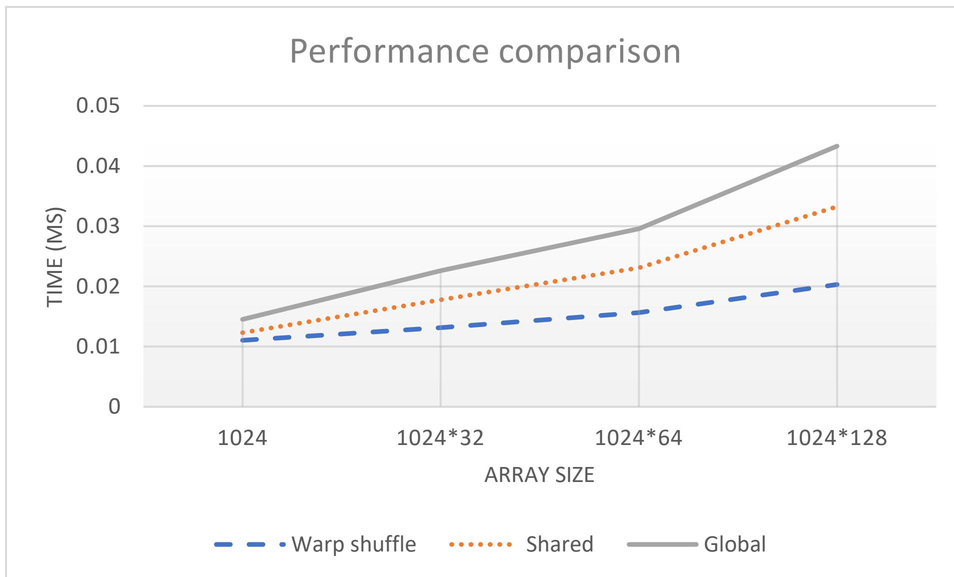

Figure 16. Comparison of timings for Method 1,Method 2 and Method 3.

It is evident from the graph in *Figure 16.* that for the given range of Array values the Method 3 was faster than Method 1 and Method 2.

Detailed analysis of results –

For detailed analysis, an array of 131072(1024*128) elements was reduced using the three methods mentioned above. And the application data obtained was studied using Nvidia Visual Profiler.

The data transfer operations in GPU have more latency than computation operations. So, it is imperative to study the number of Global and shared memory accesses in each of the methods. Table 4. given below shows the Global and Shared memory transactions in the three methods.

Table 6.

Global and Shared memory transactions

	Method 1	Method 2	Method 3
Global memory transactions	14106	4229	4229
Shared memory transactions	0	127298	12888

Global memory transactions –

Figure 17. shows a chart representing the Global memory transactions in all three methods. It is evident from that the first method which uses only global memory has the most global memory transactions. Whereas the other two

methods have limited global memory transactions, which are necessary to access array copied to global memory from host memory.

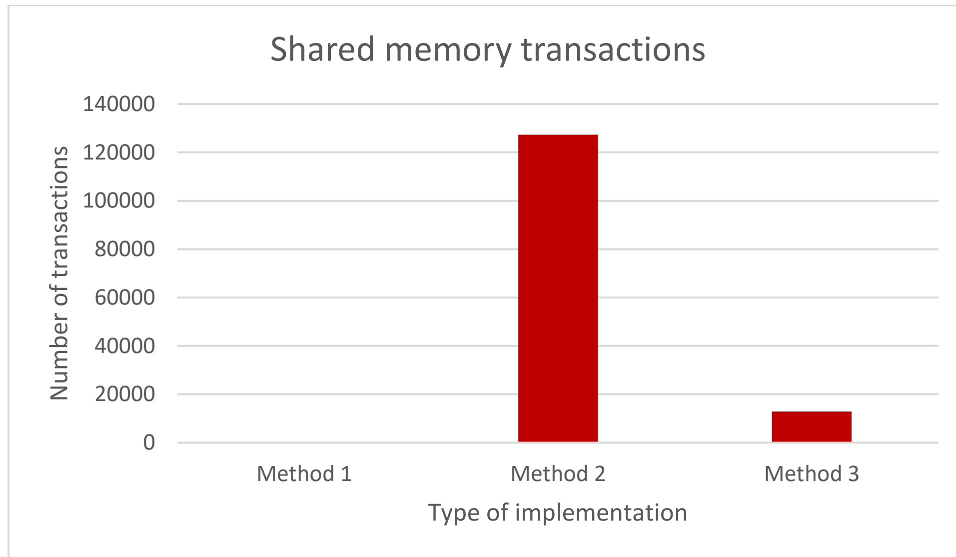

Figure 17. Global memory transactions

The higher number of Global memory transactions is the reason the Method 1 has the highest latency, as the memory bandwidth of Global memory is nearly 100x of Shared memory.

Shared memory transactions –

Figure 18. shows a chart representing the Shared memory transactions in all three methods.

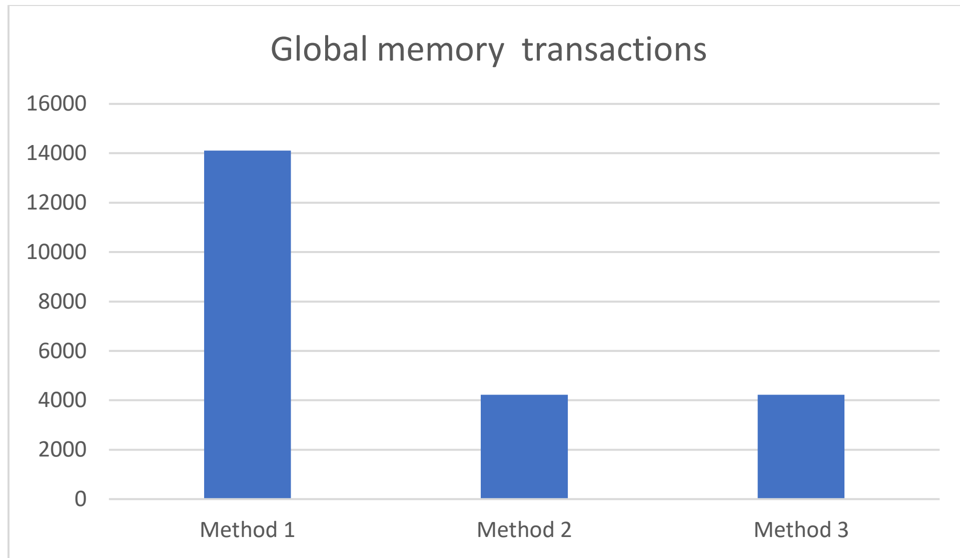

Figure 18. Shared memory transactions

As shown in the chart, the Method 1. directly accesses Global memory and makes no use of on-chip shared memory. The Method 2. makes a noticeably higher use of Shared memory as compared to Method 3. The third method carries out warp level reduction using Shuffle instruction and hence makes use of Shared memory for only block level reduction. The lower shared memory transactions in the Method 3 is the reason for its lower execution time than Method 1 and Method 2.

It is evident from this analysis that by minimizing global and shared memory transactions using warp- shuffle instruction, we are able to achieve a speed-up in implementation of parallel reduce algorithm.

CHAPTER 9

CUDA IMPLEMENTATION OF BLIINDS-II ALGORITHM

In this chapter will discuss the CUDA implementation of BLIINDS-II algorithm as carried out by Aman Yadav as a part of his Master's thesis and discussed in Chapter 4. of his Thesis Document (Yadav, A. (2016)).

Here, we will discuss how the steps of BLIINDS-II algorithm are mapped onto both CPU and GPU. Here, distorted image is first cast as a linear array of floats on the CPU and then copied across to the GPU via PCIe bus. The GPU then performs the DCT model based feature extraction of 24 floating point features and copies the features array back to the CPU. And finally, the BLIINDS-II quality score is computed by the CPU from the obtained features. The detailed flowchart describing the CUDA implementation of BLIINDS-II is as shown in *Figure 19.* The three main parts of the flow are:

a) CPU: Distorted image is read and cast as float.

b) GPU: The DCT and statistical modeling is performed across the image on three spatial scales.

c) CPU: Using the features array populated by the GPU, the BLIINDS-II quality score is calculated.

The number of transactions across the PCIe bus are minimized as it is known as a source of potential bottleneck.

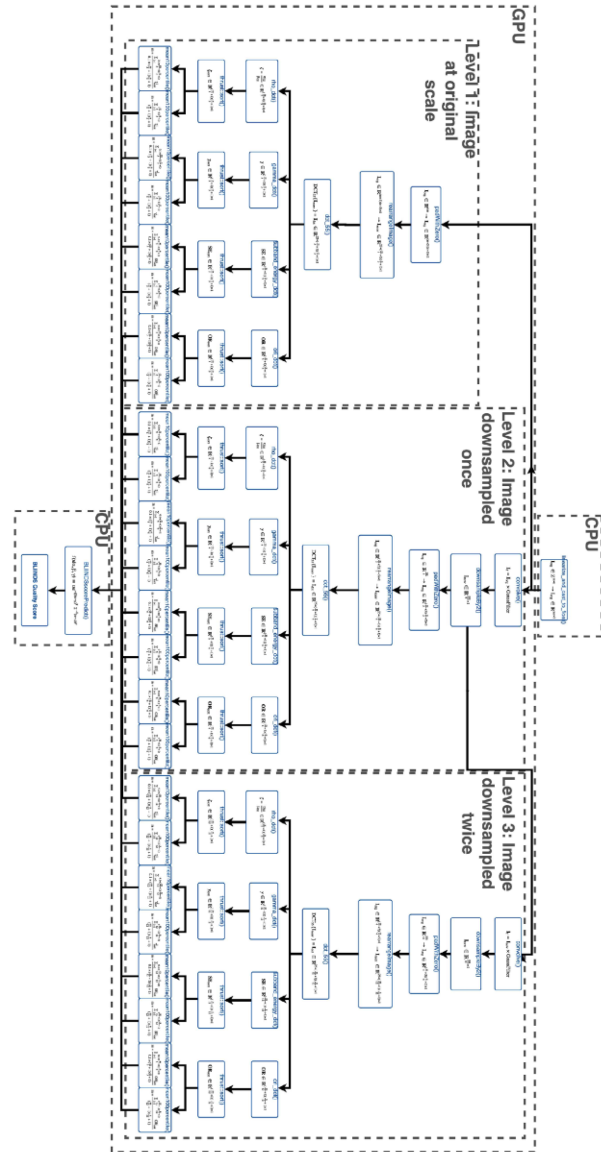

Figure 19. Flowchart of CUDA implementation of BLIINDS-II algorithm.

(Yadav, A. (2016))

Image Read On CPU And Linearization: -

Using OpenCV, the input distorted image is read as a 2D array of 8-bit unsigned char which is linearized into a 1D array of floats. A single channel input image of dimensions 512x512 pixel is converted to a 1D array of size 1MB. This array is then transferred across to GPU using PCIe bus.

Features extraction in GPU

At Original spatial scale

1. padWithZero() : This kernel is responsible for padding the image with zeros. The grid size of this kernel is m+5 and block size is n+5 where dimension of image is nxm. That is execution configuration of (517,517) for a 512x512 image. Here, 5 is added because it is the size of Local square DCT block used in the algorithm, Presence of a conditional 'if statement' can cause some amount of thread divergence in the code.

2. rearrangeImage(): The image is in form of a linear array as it is was linearized in CPU. This linear array is rearranged in this kernel as the

elements within the same 5x5 block should fall in contiguous memory locations. Hence, the kernel forms a linear array of 25 elements within a 5x5 block, which is then stored sequentially in global memory. For coalesced global memory accesses of each thread, the image is rearranged according to 5x5 block of pixels. Here on account of duplication of data, the size of image array(512x512) is increased by a factor of 115.

3. Dct_55(): The 5x5 blocks stored in contiguous memory by previous kernel is applied with 2D 5x5 DCT. There are two versions of local DCT computation-

i. DCT version 1

Here first DFT is computed using bundled library in the form of cuFFT which provides fast computation of DFT, and then DCT is computed from it.

ii. DCT version 2

Now, DCT coefficient matrix is obtained using double precision matrix multiplication method.

4. Rho_dct(): This kernel models each of set of 5x5 DCT coefficients to univariate generalized Gaussian density. The local DCT coefficients are stored in shared memory. Here, one warp is dedicated to each 5x5 block of DCT coefficients to ensure that the coefficients are copied in parallel from the global memory. In the Kepler GK110, there are thrice as many single precision cores a double precision cores hence only floating point multiplication and divions are carried out.

5. Gamma_dct(): Here, first rho = mean / (variance + 0.000001) is calculated. Then a lookup table containing values of rho is searched linearly. Gamma value from another lookup table coreesponding to the rho values are obtained from another lookup table. These both look up tables are 1D arrays of floats.

6. Subband_energy_dct(): The ratios of energy stored in radial frequency bands is computed by this kernel. The kernel is designed in such a way that one warp of threads is dedicated to 5x5 block of DCT coefficients.

7. Ori_dct(): The orientation model parameter is computed in this function. The computed values are stored in an array of floats. This function can be computed in two ways-

i. Having three kernels for each orient and a fourth kernel to find variance value of each orient

ii. Having a single kernel which is formed by merging the four kernels. This removes the need for of writing intermediate results in global memory.

8. Thrust::sort(): Each array of model parameters is sorted using sort function provided in Thrust library. Thrust::sort uses radix sort to sort these arrays and the results are written back to respective arrays

9. Mean10percentile() and mean100percentile(): Reduce function of Thrust library, thrust::reduce() is used to implement both mean10percentile() and mean100percentile() functions. Thrust::reduce on the 10% of the sorted array returns the sum of the lowest 10% values in the array. The sum is then divided by 10% of the array size. The mean of 100percentiles is calculated in similar fashion.

Down sample image to lower spatial scales

Here the calculations are carried out on downsampled image, that is at second spatial scale. For downsampling following two functions are implememted.

1. ***Convolove():*** Here the input image is convolved with a 3x3 Gaussian filter. This operation is implemented in two kernels, one for row wise convolution and other for column wise convolution. This is necessary as both operations need to be implemented sequentially.

2. ***downSampleBy2():*** This image copies over the filtered image to give the downsampled image. After this stage the steps 1-9 followed for original image are carried out for downsampled image and thus next eight features are obtained.

One more iteration of downsampling and extraction is carried out for level 3 computations.

Computation of BLIINDS-II quality score on CPU

The thrust::reduce is designed to send the result of reduction on GPU arrays back to CPU over PCIe bus. Thus, the score prediction stage is carried on CPU.

BLIINDSscorePredict():

This function is executed in CPU. It calculates the posterior probability of the score being of a specific value for a given set of extracted features. The probability is calculated in the range of 0-100 in steps of 0.5 for each quality score.

Thus, we finally have the predicted BLIINDS-II score of the given image in CPU.

CHAPTER 10

THRUST VS. CUB LIBRARY

In CUDA implementation of BLIINDS-II algorithm, thrust::sort and thrust::reduce functions are used for sorting and reducing every model parameter in every spatial scale. And thus is responsible for nearly 50% of compute time on GPU. These two functions are two major bottlenecks in CUDA implementation of BLIINDS-II (Yadav, A. (2016)). This can be clearly viewed by analyzing the application using NVIDIA visual profiler as shown below in *Figure 20.*

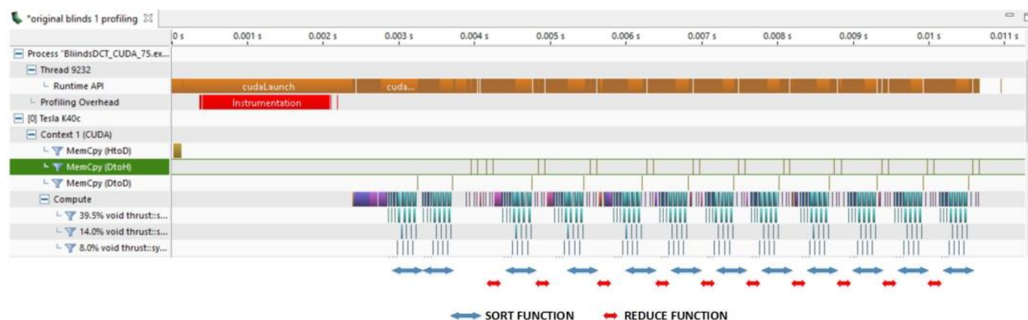

Figure 20. NVVP analysis of CUDA implementation of BLIINDS-II algorithm

Thus, in this chapter we will discuss in detail about Thrust library, which is extensively used in CUDA implementation of BLIINDS-II algorithm.

THRUST library

Thrust is a C++ template library of parallel algorithms and data structures for CUDA based on Standard Template Library(STL). THRUST library offers a high-level interface to users that is interoperable with CUDA C. High-level interface increases developer productivity accounting to the minimal programming effort required. Data parallel primitives such as scan, reduce and sort to perform GPU-accelerated complex algorithms can be implemented using Thrust library. Thrust library takes care of lower level functionality like memory access and allocation thus a person with limited C++ or CUDA experience will be able to provide rapid prototyping of CUDA applications. (Thrust| Cuda Guide; Thrust. (2017, April 25))

CUB library

CUB or “CUDA Unbound” was developed by Duane Merrill of NVIDIA Research. This library targets multiple levels of application development by providing high-performance generic primitives. Similar to Thrust library, CUB

library provides a set of device-wide primitives, which can be called from host. But along with that it also provides access to Kernel components that operate on block and warp level. Thus we get software components for every level of CUDA programming model. CUB is one of the few libraries which provide reusable kernel primitives. CUB is flexible and tunable to kernel needs because CUB's primitives are not bound to any particular width of parallelism or to a specific set of data types

CUB provides a wide spectrum of parallel contexts like

1. Flexibility in use of data types

2. Threads per Block can be defined

3.Data items per thread can also be altered

4. Makes complete use of underlying architecture

5. Takes care of Tuning requirements

NVIDIA has provided a great diversity in GPU hardware, which is continuously evolving to accommodate new architecture specific features and instructions. CUB primitives are specially designed to take advantage of the underlying

architecture and performance tune them to match processor resources in the architecture of the CUDA processor. This leads to much better performance portability in CUB implementations as compared to rigidly-coded libraries like Thrust.

It must be noted that the CUB v1.1.1 version of CUB implements warp shuffle instruction to facilitate warp wide communication of arbitrary threads in devices with CUDA compute capability of 3.0 and higher.(What is CUB?; Introducing CUDA UnBound (CUB).)

Comparison of THRUST vs CUB library.

Table 7.

Comparison of Thrust vs CUB library(Thrust| Cuda Guide; Thrust. (2017, April 25)) .(What is CUB?; Introducing CUDA UnBound (CUB).)

THRUST	CUB
Access to only High-level interface	Access to even lower level interface (kernel components at warp and block level)
Reduce function has the functionality of returning data to host	No such functionality is documented. cudaMemcpy needs to be used
Does not make use of underlying hardware	Makes use of underlying hardware
Rigidly coded library	Flexibly coded for higher portability.
Width of parallelism cannot be defined	Width of parallelism can be defined
Data items per thread cannot be controlled	Gives control over underlying architecture

Designed for fast prototyping of CUDA applications even for developers who have limited CUDA C++ knowledge.	Designed for developers who need to fine tune the use of library according to the algorithm and architecture.

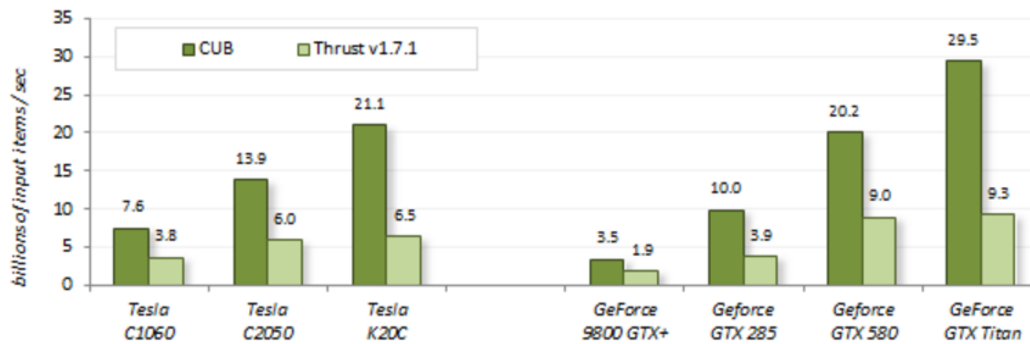

Figure 21. Comparison of performance portability between Thrust and CUB for device wide prefix scan. (32M int32 items) (What is CUB?)

From the above discussion and the results shown in *Figure 21* it can be seen that CUB provides great edge over Thrust library.

CHAPTER 11

NEW BLIINDS-II IMPLEMENTATION AND ITS ANALYSIS

Thus, based on above comparisons and based on analysis provided in Chapter 8, we will use CUB library in CUDA implementation of BLIINDS.

In this implementation, the specifications of hardware are given in Table 3 and Table 4, which is the same hardware which was used by Yadav A in his implementation of BLIINDS-II algorithm (Yadav, A. (2016).

Implementation of Thrust library in sort and reduce functions of BLIINDS-II algorithm.

Following sort function from CUB library is used in CUDA implementation of BLIINDS-II by Yadav A (Yadav, A. (2016).

```
thrust::sort (const thrust::detail::execution_policy_base< DerivedPolicy > &exec,
RandomAccessIterator first, RandomAccessIterator last)
```

The parameters entered in this case are-

1. execution policy – thrust::device

2. first iterator – first element of model parameters

3. last iterator- first element + total no. of DCT blocks in current scale

Thus, sort function of Thrust library will be implemented as follows : -

```
thrust::sort(thrust::device, model_parameter, model_parameter + total no.of DCT blocks);
```

For reduce function, following function from Thrust library is used: -

```
thrust::reduce (const thrust::detail::execution_policy_base< DerivedPolicy > &exec,
InputIterator first, InputIterator last)
```

Where parameters are-

1. execution policy – thrust::device

2. Input iterator first – model parameter

3. Input iterator last - first element + total no. of DCT blocks in current scale

Thus, reduce function of Thrust library will be implemented as follows : -

```
features[n] = thrust::reduce(thrust::device, model_parameter, model_parameter + total no.of DCT blocks);
```

It must be noted that reduce function in Thrust library gives functionality of transferring the result directly to the host without calling cudaMemcpy.

Using CUB for sort function

The sort function of CUB library which suits our application of sorting is -

```
cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_in, num_items);
```

For device wide sorting, this function needs to be called twice wherein first

sorting of all the blocks takes place and in the second call device wide sorting

takes place. This is because two kernels need to be launched for block wide

synchronization.

We create a function my_cub_sort() for this.

```
void my_cub_sort(float *d_keys_in, int num_items)
{
    void     *d_temp_storage = NULL;
    size_t   temp_storage_bytes = 0;
    cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_in, num_items);
    // Allocate temporary storage
    cudaMalloc(&d_temp_storage, temp_storage_bytes);
    // Run sorting operation
    cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_in, num_items);
}
```

For transfer of data from first function to second temporary storage is

assigned in the function.

Using CUB for reduce function

The reduce function of CUB library which suits our application of reducing is -

```
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);
```

As with sort, this function too needs to be called twice for device wide reduction. So, a function my_cub_reduce is created to carry out reduction as shown below: -

```
float my_cub_reduce(float *begin1, float *end1){
    size_t num_items1 = end1 - begin1;
    float *d_in_image1 = begin1;
    float  result1;
    size_t   temp_storage_bytes = 0;
    cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in_image1, d_out1, num_items1);
    cudaMalloc(&d_temp_storage, temp_storage_bytes);
    cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in_image1, d_out1, num_items1);
    cudaMemcpy(&result1, d_out1, sizeof(float), cudaMemcpyDeviceToHost);
    return result1;
}
```

Here, it must be noted that the function has cudaMemcpy to return the reduced value back to CPU. Thus, my_cub_reduce function is able to mimic the functionality provided by thrust::reduce function of returning value back to CPU.

Results and analysis of implementation

When the algorithm was run for one image, it was observed that the original implementation with thrust library took 7.73983 ms.

Whereas, the new implementation with CUB library took 6.6995 ms. This means a speedup of about 13.44% was achieved.

But, in practical applications, an image quality algorithm will be applied on videos which is composed of many different images. Hence, for further analysis, we applied the algorithm on 43 different assorted images and ran 10 iterations of the image.

Figure 22. shows the graphical representation of time taken versus the number of images in the implementation.

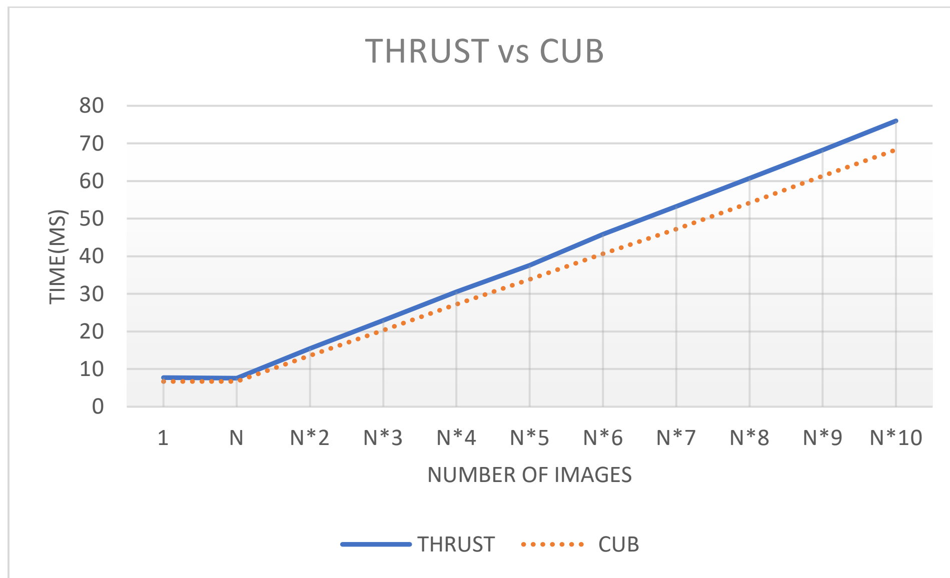

Figure 22. THRUST vs. CUB implementation of BLIINDS-II

Here the dataset is of 43 assorted images of dimension 512*512 for which 1-10iteerations are ran with BLIINDS-II algorithm. The timings for this dataset are given in Table 6.

Table 8. Thrust vs. CUB implementation of BLIINDS-II algorithm

IMAGE	Using Thrust	Using CUB
1	7.73983	6.6995
N	7.58516	6.72336
N*2	15.4391	13.5651
N*3	22.981	20.3447
N*4	30.5025	27.2407
N*5	37.5745	33.8455
N*6	45.8544	40.6269
N*7	53.2077	47.2163
N*8	60.751	54.2133
N*9	68.2051	61.3173
N*10	75.9846	68.3108

It must be observed that in every case a evident speedup is observed in the CUB implementation of BLIINDS-II.

This can be accounted to the fact that CUB library makes use of the underlying Kepler architecture and uses the warp level shuffle instruction for implementing reduce and sort functions.

CHAPTER 12

DISCUSSION AND CONCLUSION

In the Experiment carried out in Chapter 8 on the Reduce algorithm, we have found that Reduce algorithm implemented using shuffle instruction is evidently faster than performing reduction algorithm using global memory or even shared memory. We have also established that this holds even true for incrementing array sizes.

After microarchitectural analysis of this method, it was found that the speedup can be accounted to the fact that the implementation of reduce algorithm using shuffle instruction, entails minimum number of shared memory and global memory transactions.

Based on this experiment, we decide to replace the rigidly coded Thrust library with a flexible CUB library, which makes use of shuffle instruction in BLIINDS-II algorithm. This implementation took an estimated time of 7ms as opposed to 9ms taken by the original implementation as documented by Yadav A. (Yadav A. (2016)). The original implementation was supposed to be

able to handle a video feed of more than 100fps(around 110fps). The new implementation will now be able to handle a video feed of around 140fps on account of the speedup obtained in algorithm.

CHAPTER 12

FUTURE WORK

Following optimization techniques are recommended for future work: -

- Using Cuda streams – Introduce synchronization in various kernel calls and data transfers using different streams
- Dynamic parallelism – Due to sequential nature of algorithm, it must be explored, if one kernel can be made to launch another kernel using CUDA dynamic parallelism.
- More use of warp level shuffle instruction in algorithm – We have used warp level shuffle instruction only for reduce and sort function in our implementation. It must be explored if this approach can be used for other kernels as well.

REFERENCES

1. Saad, M. A., Bovik, A. C., & Charrier, C. (2012). Blind image quality assessment: A natural scene statistics approach in the DCT domain. IEEE Transactions on Image Processing, 21(8), 3339-3352.

2. Yadav, A. (2016). GPGPU based Implementation of BLIINDS-II NR-IQA (Doctoral dissertation, Arizona State University).

3. Graphics Processing Unit (GPU). (NVIDIA). Retrieved from http://www.nvidia.com/object/gpu.html

4. What is GPU (graphics processing unit)? - Definition from WhatIs.com. (n.d.). Retrieved from http://searchvirtualdesktop.techtarget.com/definition/GPU-graphics-processing-unit

5. NVIDIA on GPU Computing and the Difference Between GPUs and CPUs. (n.d.). Retrieved from http://www.nvidia.com/object/what-is-gpu-computing.html

6. GPU-Accelerated Applications. (n.d.). Retrieved from http://www.nvidia.com/object/gpu-applications.html

7. Reverse Time Migration. (n.d.). Retrieved from http://www.acceleware.com/rtm

8. NAMD Introduction | GPU Accelerated Applications. (n.d.). Retrieved from https://www.nvidia.com/object/gpu-accelerated-applications-namd.html

9. MSC Nastran 2013 Acceleration on NVIDIA Kepler (n.d.). Retrieved from http://www.nvidia.com/object/tesla-msc-nastran-accelerations.html

10. ArrayFire. (2017, April 25). Retrieved from https://developer.nvidia.com/arrayfire

11. History of CUDA, OpenCL, and the GPGPU. (n.d.). Retrieved from https://www.olcf.ornl.gov/kb_articles/history-of-the-gpgpu/

12. CUDA C Programming Guide. (2017). Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide

13. An Easy Introduction to CUDA C and C. (2017, January 25). Retrieved from https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

14. Kannan, V. (2016). An Analysis of the Memory Bottleneck and Cache Performance of Most Apparent Distortion Image Quality Assessment Algorithm on GPU (Doctoral dissertation, Arizona State University).

15. CUDA Memory and Cache Architecture. (2011, September 10). Retrieved from http://supercomputingblog.com/cuda/cuda-memory-and-cache-architecture/

16. CUDA Memory Model. (2013, November 26). Retrieved from https://www.3dgep.com/cuda-memory-model/

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

18. Gupta, N. (n.d.). CUDA Programming. Retrieved from http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

19. Wilt, N. (2013). The CUDA handbook: a comprehensive guide to GPU programming. Upper Saddle River, NJ: Addison-Wesley.

20. Tsutsui, S., & Collet, P. (2016). Massively Parallel Evolutionary Computation on GPGPUs. Berlin: Springer Berlin.

21. CUDA C Best Practices Guide. (n.d.). Retrieved from http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

22. How to Access Global Memory Efficiently in CUDA C/C Kernels. (2014, June 09). Retrieved from https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

23. Farber, R. (2012). CUDA application design and development. Waltham, MA: Morgan Kaufmann.

24. Using Shared Memory in CUDA C/C++. (2014, July 21). Retrieved from https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

25. Bank conflicts in shared memory in CUDA | CUDA Programming. (n.d.). Retrieved from http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html

26. Whitepaper | NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110 [PDF]. (n.d.). NVIDIA. Retrived from https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

27. CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 10). Retrieved from https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/

28. Szalwinski, C. (n.d.). Memory Model |CUDA Programming Model . Retrieved from https://scs.senecac.on.ca/~chris.szalwinski/archives/gpu610.131/pages/content/cudam_p.html

29. Owens john and luebke dave. Fundamental GPU Algorithms | Intro to Parallel Programming by NVIDIA. Retrieved from https://www.udacity.com/course/intro-to-parallel-programming--cs344

30. Blake, R. and Sekuler, R. Perception, 5th ed. New York: McGraw Hill, 2006.

31. Thrust. | CUDA guide (n.d.). Retrieved from http://docs.nvidia.com/cuda/thrust/index.html#axzz4kR8xsM6O

32. Thrust. (2017, April 25). Retrieved from https://developer.nvidia.com/thrust.

33. Ashkiani, S., Davidson, A., Meyer, U., & Owens, J. D. (2016, February). GPU multisplit. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (p. 12). ACM.

34. A. V. Monakov, V. A. Ivanishin, “Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler”, Proceedings of ISP RAS, 28:4 (2016), 169–182

35. The Data Explosion in 2014 Minute by Minute – Infographic. (2014, July 12). Retrieved from https://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic

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

37. Faster Parallel Reductions on Kepler. (2016, June 22). Retrieved from https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

38. CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics. (2014, October 06). Retrieved from

https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

39. What is CUB? (n.d.). Retrieved from https://nvlabs.github.io/cub/

40. Introducing CUDA UnBound (CUB). (2014, April 14). Retrieved from https://www.microway.com/hpc-tech-tips/introducing-cuda-unbound-cub/

96