Energy Efficient GPGPU :Data Specific Thread Scheduling for GPGPU Architechture

submitted in partial fulfillment of the requirements

for the degree of

MASTER OF TECHNOLOGY

in

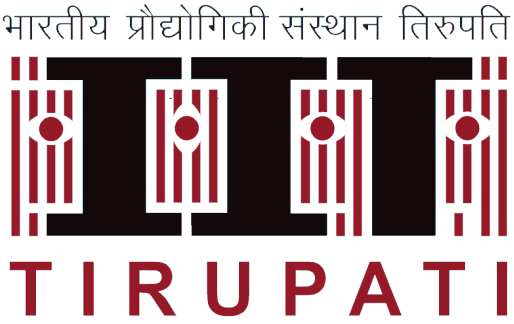
COMPUTER SCIENCE AND ENGINEERING

by

HEMANT KUMAR PATHAK CS20M007

Supervisor(s)

Prof. Jaynarayan T Tudu



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY TIRUPATI

May 2022

DECLARATION

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/-data/fact/source in my submission to the best of my knowledge. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Place: Tirupati
Date: 24-05-2022

Signature
Hemant Kumar Pathak
CS20M007

BONA FIDE CERTIFICATE

This is to certify that the report titled **Energy Efficient GPGPU :Data Specific Thread Scheduling for GPGPU Architechture**, submitted by **Hemant Kumar Pathak CS20M007**, to the Indian Institute of Technology, Tirupati, for the award of the degree of **Master of Technology**, is a bonafide record of the project work done by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Tirupati
Date: 24-05-2022

Prof. Jaynarayan T Tudu
Guide
Assistant Professor
Department of Computer Science and Engineering
IIT Tirupati - 517501

ACKNOWLEDGMENTS

This project work has had an incredible impact on me; It's been a period of great learning; I have developed myself at technical as well as personal levels. I am very thankful for the support helped me a lot throughout this work as a token of gratitude to my guide, Dr. Jayanarayan Thakurdas Tudu, Department of Computer Science and Engineering, IIT Tirupati.His guidance and useful suggestions, which helped me in completing this project work in time. And also, thanks for helping me in every situation and taking the pain in teaching me though I had to stay away from the institute. I 'am grateful to both the Director, IIT Tirupati, Deans of IIT Tirupati, and Head of the Department, Dept. of CSE, IIT Tirupati, for having permitted me to carry out this project work at IIT Tirupati. Lastly , I' am thankful to all the faculty members of the CSE Department, IIT Tirupati, and all my classmates who gave wonderful learning experience in this institute.

ABSTRACT

KEYWORDS: Computer Architecture,GPGPU, Dynamic Thread Scheduling, Fetch & Decode Unit, Optimization

Power remains one of the crucial problems in the computing domain to be solved. GPGPU based computing machines are currently being used extensively to solve computational problems in the field of machine learning and matrix-based computations.Currently, most of the GPU architecture executes the thread as per the order provided by the programmer. However, the ordering of the thread could be altered for the purpose of reducing power without making any impact on the correctness of the program. The real challenge would be to find the power-efficient ordering during runtime. The template-based thread ordering could be one of the possible methodologies.In this paper,we are explore the methodology that how dynamically regroup data specific thread into new warp based on minimal hamming distance of Register content and how we reduce the dynamic power and increase the performance to make the GPGPU energy efficient.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

IITT	Indian Institute of Technology, Tirupati
GPGPU	General Purpose Graphics Processing Unit
PTX	Parallel Thread Execution
TLP	Thread Level Parallelism
SIMT	Single Instruction Multiple Thread
GFLOPs	Giga floating point operation per second
SMs	Streaming multiprocessors
BS	block-size
PTX	Parallel Thread Execution
ALU	Arithmatic Logic Unit
RAM	Random Access Memory
CUDA	Compute Unified Device Architecture
DMA	data memory access
NOC	Network-on-chip
HBR	highly banked register

CHAPTER 1

INTRODUCTION

1.1 Objectives and Scope

GP-Graphics Processing Unit (GPU) have been one of the strongest candidates to run workloads with fine-grained parallelism for the past decade. To reduce the total performance overhead of fetching and decoded instructions, GPUs adopt the Single Instruction Multiple Thread (SIMT) technique for execution. GPUs' massive processing and on-chip memory capacity allow them to run hundreds of threads at once with minimal context switching overhead. GPUs are utilized for general-purpose applications as well as multimedia because they could achieve thousands of GFLOP's of peak performance with a manageable power cost. As a result, several programmers implemented their parallel application using General Purpose Graphics Processing Units. GPGPU, on the other hand, has a low power efficiency due to the fact that most GPU resources are underused due to high TLP lagging Aghilinasab *et al.* (2016); Abdel-Majeed *et al.* (2013*a*); Jog *et al.* (2013).

I will propose data specific thread ordering algorithm to reduce the dynamic power to make the GPGPU energy efficient. Currently, most of the GPU architecture executes the thread as per the order provided by the programmer or used template which is application oriented. However, the ordering of the thread could be altered for the purpose of reducing power without making any impact on the correctness of the program Aghilinasab *et al.* (2016).

1.2 Nvidia GTX480 Architecture Overview

The baseline architecture is the NVIDIA- GTX480 -Fermi GPGPU. The GTX480 is made up of 15 Streaming Multiprocessors (SMs). The overall design of SM is depicted using Figure 1. A huge register file unit, thread or warp scheduler, and execution units

make up each of SM. Each SM has its own 64KB cache and shared memory. Per SM, Fermi can accommodate maximum 48 active warps at a time. The Single Instruction Multiple Thread (SIMT) executing model consists of 32 threads performing the same instruction in lockstep on each warp. As a result, each SM has 1,536 active threads Kayiran *et al.* (2016*a*).

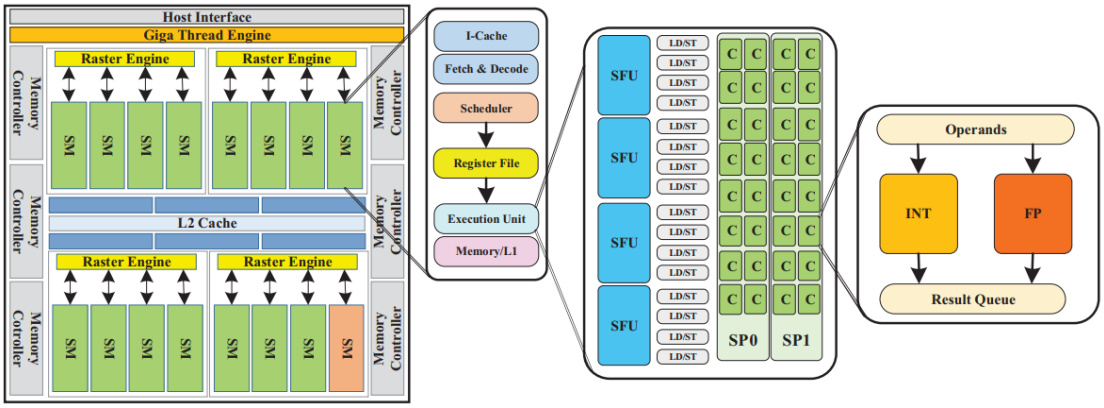


Figure 1.1: Baseline GPGPU Architecture (NVIDIA GTX480)

The block diagram of the execution units within each SM is shown in Figure 1. they have Two shader core processors (SP), 16 Load/Store units for memory related operations, and 4 SFUs for specific arithmetic functions like tanh and cosh make up the execution units. Each of SP has sixteen SIMT processing lane cores (also known as cores of CUDA) that run at twice the frequency of the main clock. Therefore result, during a single issue cycle, each SP can operate 32 concurrent threads. One one FP units and int units are included in each CUDA core. Annavaram and M. Abdel-Majeed (2013) Kayiran *et al.* (2016*a*). According to earlier studies, GPGPU has no one power-consuming component. However, execution units in GPGPU account for more than 20

1.3 SIMD SP Architecture

Figure 2 depicts the GPU's basic architecture. Multiple simultaneous threads operating on each shader core execute the identical shader programme in this diagram, with the hardware executing each of the thread instruction in-order fashion. The scheduler divides several threads on core as a warps in SIMD pathway. Using pipeline, each of the warp executing the same command on several data parallelly. Instructions read their operands

in parallel from a HBR file. The Memory requests and cache misses are accessible by highly banked cache misses and data cache.

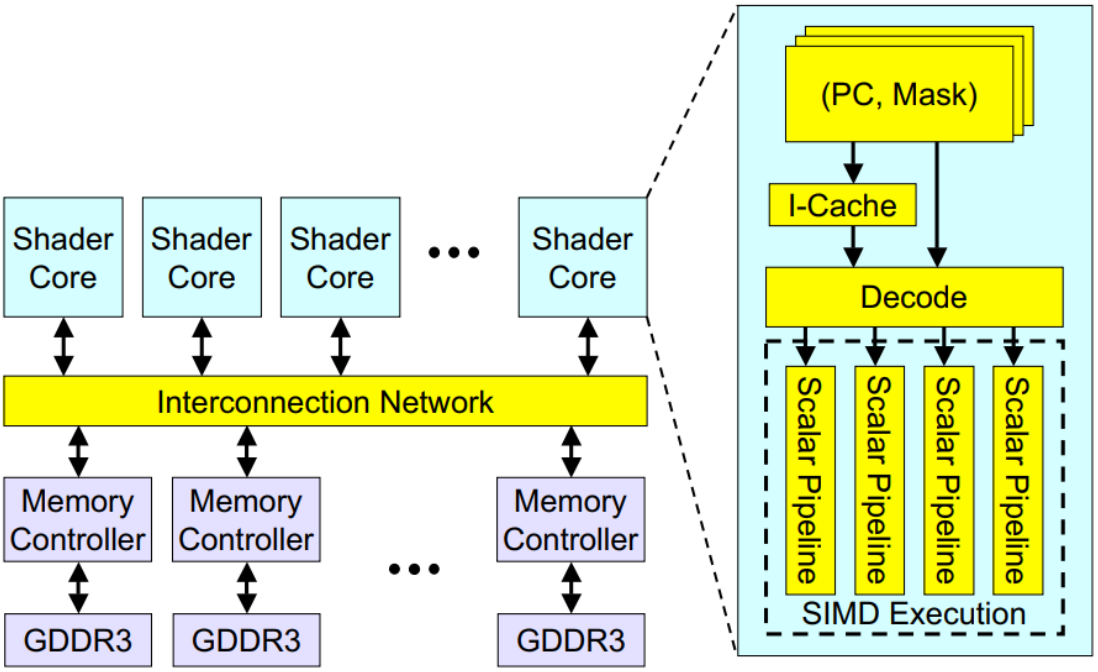


Figure 1.2: Baseline SP Architecture:Reg read, execute, memory, and write-back are all part of the Scalar Pipeline.

Through an interconnection network, data is routed into memory and next-level cache. To reduce row activation and preload cost, each of the memory controller can processes the memory requests by accessing its own corresponding DRAM, sometimes it may differ in order than bids received. A crossbar that contain parallel and iterative mapping allocator is the link network that we simulate. Figure 2 is missing graphics-related features because this paper focuses on non-graphics applications. Traditional graphics processing, on the other hand, continues to have a strong influence on this design: using SIMD h/w to run single-programming and multiple data (SPMD) s/w (that can communicate between threads) is widely promoted.While balancing the efficient execution of a "general purpose" computing kernel with large amount of legacy and essential (for GPU seller) graphics software with little or no control thread activity control within shader (shaders programs of graphics will most likely use flow control operations more and more in coming era, i.e.,, to achieve believable lighting effects), which increases the significance of this work) Fung *et al.* (2007).

1.4 Latency Hiding

Because hit rates in cache are typically lowest in streaming applications,where performance suffers greatly if the path fails for each missed memory request. It is mainly true when requesting memory is delayed by multiple cycles because to the combination of contention in the connected network and triggering an overload at the DRAM data point. While old microprocessors can reduce impact of the cache, Without the use of out of order execution, a more advisable approach to use software to achieve parallelism Fung *et al.* (2007). is to switch between threads for the execution of instructions. With

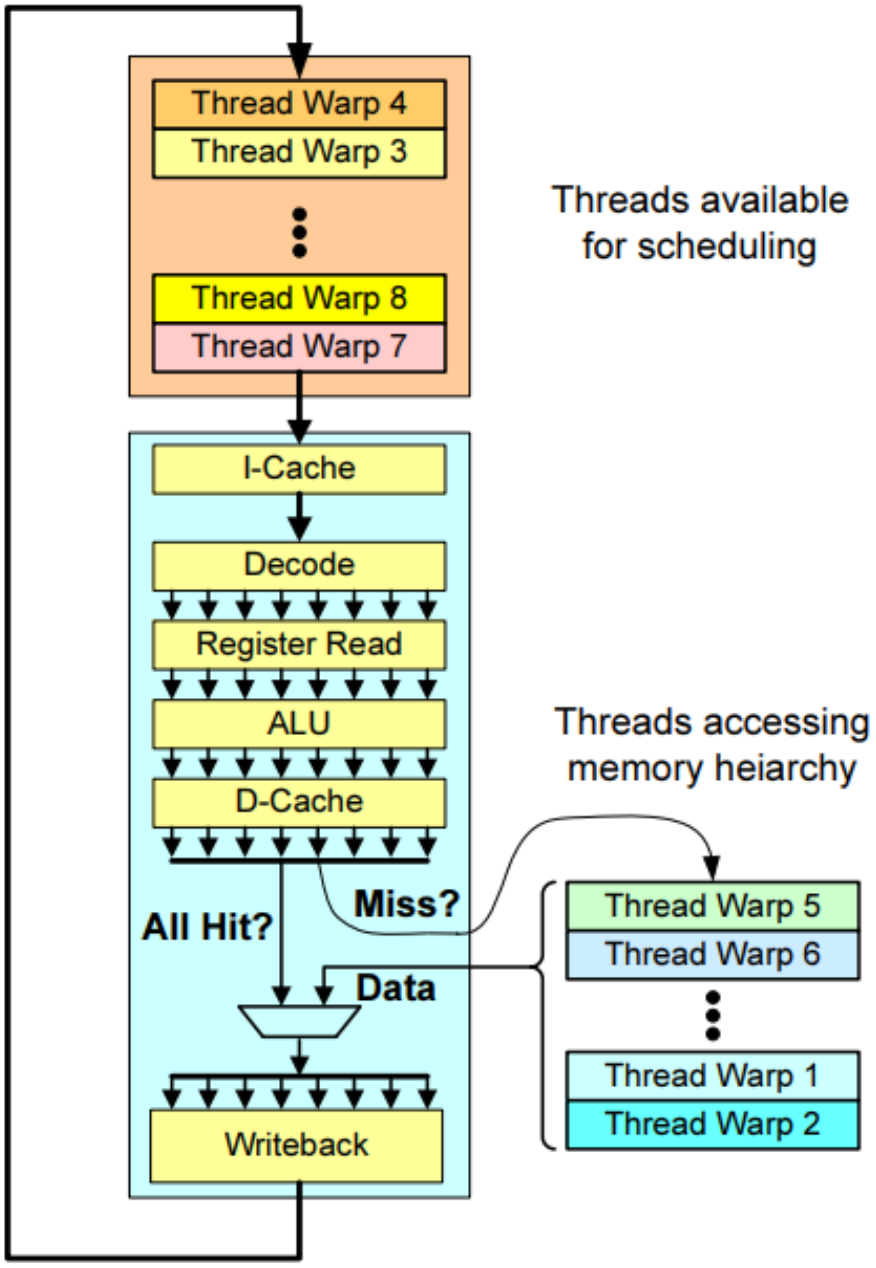


Figure 1.3: To hide DMA latency, use barrel processing.

a "large number of shader strings concatenated on the same execution resources, our architecture can use thin multithreading" (so called "bin processing"), in which individual thread is overlayed by fetching unit to actively conceal potential dropout time before it occurs.Figure 3 shows how instructions from multi shader core threads are distributed equally in a round-robin line When a shader thread is stalled due to a memory request, the shader core simply removes it from the "ready thread pool"., allowing other shader threads to continue while system memory processes your request. Because the pipe is being used by other threads while memory operations are being completed, handling concealed crates effectively reduces the latency of most memory operations.

1.5 SIMD Execution of Scalar Threads

Memory latency can be hidden using barrel processing. Modern GPUs should be able to take advantage of the clear-cut parallelism provided by the sp model associated with programmable on relatively simple h/w. Shader hardware designed for maximum performance at the lowest possible cost. SIMD hardware can be supported effectively. SPMD programme execution provided that a single thread follows a similar control flow. Figure 4 depicts the grouping of instructions from multiple shader threads. With a single warp which follow SIMD principle, you can plan multiple things at once. The scalar pipeline In, multiple scalar pipelines are running. You can share all data independent logic as well as lockstep. Reduces area significantly when compared to MIMD architecture. A significant source of space savings for SIMD pipelines For specific cases, simpler statement cache support is required. Fung *et al.* (2007).

CHAPTER 2

Literature Review

• static power without affecting performance out of order (Ooo), inter type instruction will be clustered togetherto extend the idle period of execution units which improves the power around 25% and performance by 8% Aghilinasab *et al.* (2016).

• Power Saving Techniques. Abdel-Majeed *et al.* (2013*b*) proposea wavefront scheduler that clusters instructions of the same type and sends them to nearby execution units in time before switching to another instruction type (s). This lengthens the idle time of certain execution units, the PG opportunitiesAbdel-Majeed *et al.* (2013*b*).

•We, on the other hand, concentrate on computation-based datapath components. NVIDIA Tegra 4 Kayiran *et al.* (2016*b*)When executing graphics and compute applications, uses CG and DVFS techniques for computing and graphics units, respectively. Based on our detailed analyses of fine-grained component granularity, our mechanism operates at a more acceptable and component granularity utilization Kayiran *et al.* (2016*a*).

•Architectures that are configurable and heterogeneous. Many works aim to make architectures more configurable and heterogeneous in order to improve power consumption and/or performance. Park *et al.* (2012) To improve energy efficiency, adaptively customise the SIMD lanes in a multi-threaded architecture. a number of works Ipek *et al.* (2007); Kayiran *et al.* (2016*a*) propose mechanisms for transforming larger cores into several smaller cores, or integrate tiny cores to form larger ones Kayiran *et al.* (2016*a*).

•Many researchers have concentrated on lowering GPGPU power consumption. According to GPU-WATCH, no GPGPU component consumes significantly more power than any other component Aghilinasab *et al.* (2016); Leng *et al.* (2013). To reduce GPGPU's total power consumption, several components such as register files, execution units, NOC, caches, and so on should be considered. This section describes some notable works in this field Aghilinasab *et al.* (2016); Leng *et al.* (2013).

2.1 GPGPU Simulator

GPGPU simulator is cycle level simulator that simulates modern gpus running CUDA or OpenCL GPU based computing workloads. GPGPU simulator also includes AerialVision which is a performance visualisation tool, and GPU Wattch which is a configurable and extensible energy model. GPGPU simulator and GPU Wattch have been rigorously validated using real-world GPU performance and power measurements Khairy *et al.* (2020). After establishing all of the necessary conditions, the proposed work will be analysed and tested using GPGPU simulator and GPU wattch.

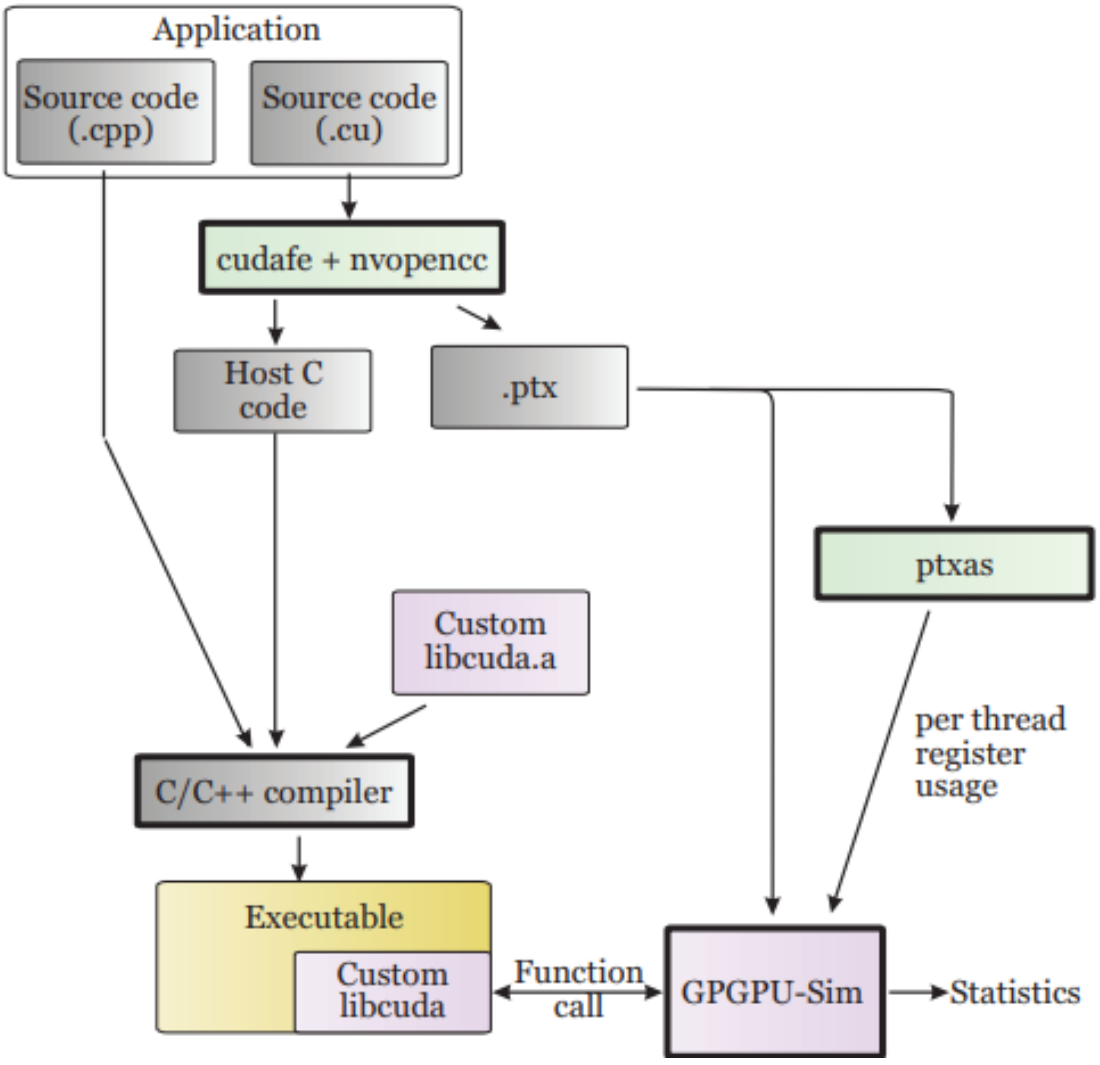


Figure 2.1: GPGPU Sim
Bakhoda *et al.* (2009)

CHAPTER 3

Methodology

3.1 Problem Definition

Problem: Data-specific dynamic thread scheduling based on the signature of the register file changed during the instruction execution within the GPU pipeline

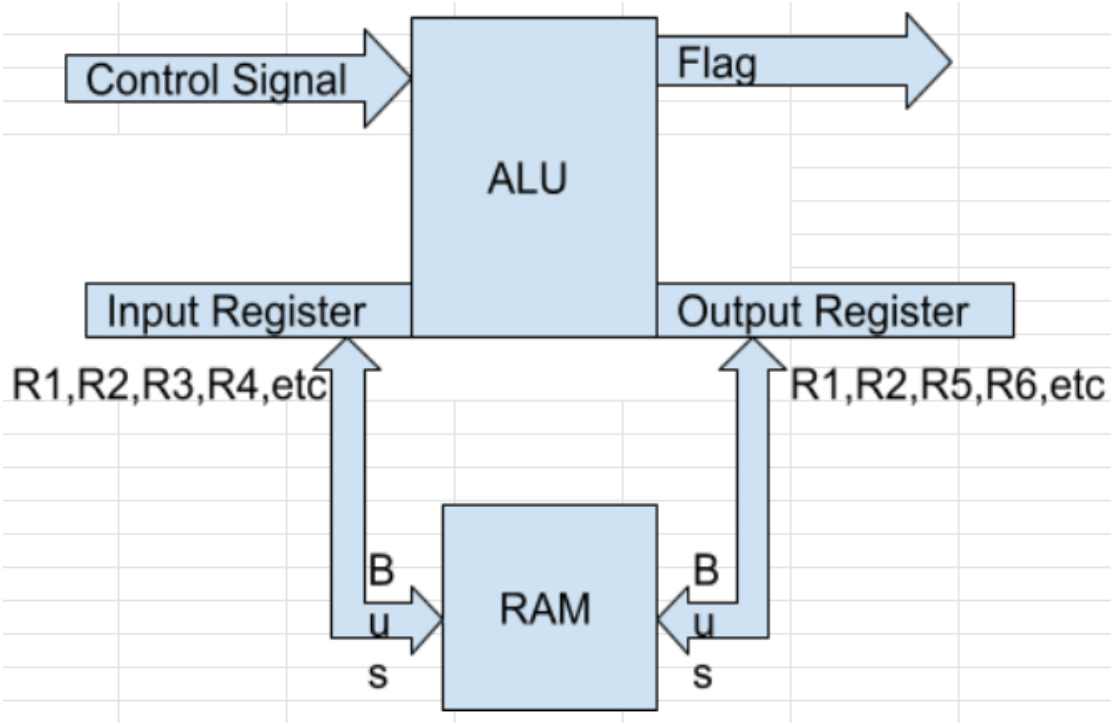


Figure 3.1: Program Execution Flow

Normally GPU computation happens in different phases viz. instruction fetch/decode, warp scheduling, register read, operand collection, streaming processor(integer instruction/floatingpoint instruction) operation, special function unit operation, and load/store unit operation.In this paper using nvidia480 (discussed in section 1.1) based GPU and gpgpu-sim (discussed in section 2.1) for simulation purpose. Figure 1.3, During Instruction fetched, from the memory store the instruction starting address into the program counter (PC) then address passed to MAR(memory address register) using MAR the

content (opcode and operand) read from the memory and stored it into MDR(memory data register)and later passed into IR (instruction register) if operand are data then directly passed to the accumulator else return back to MDR to store it.Now, the Decode process happened where the opcode are decoded based on the required weather they are store/load/alu operation, the register read the data from the MDR and help ALU to perform the operation, after the ALU operation performed the data sent back to the register to perform the write-back operation.During the above mentioned operation the register play an important role because it appears two to three times during the completion of the single instruction in the GPU or CPU pipeline.Throughout this process the signature that is the nature and content of the register changes frequently and this is the key idea dealing in this paper that how this changes impact on the power and performance of the GPU while computing.

3.2 Experimental Setup

Performing the experiment, GPU NVIDIA-480 architecture which support CUDA programming and Google-colab notebook with gpu enabled platform has been used. Two different 2D square matrices (dimension may same or different possible) were used to demonstrate how the proposed algorithm can speed up matrix multiplication computation with 10 x 10,16 x 16, 20 x 20, 48 x 48, 1024 x 1024, 2048 x 2048, and 4096 x 4096, 30 x 2048 The sizes of the two matrices to be multiplied have been investigated using 100, 256, 400, 2304, 250000,1048576,4194304, 16777216 60720 thread respectively. Each of the matrices has been divided into different block size viz. 16, 32 and 64 respectively.Each block formed by warp (collection of 32 thread).Lets 48 x 48 matrix and block size is 16 then diving(48, 16)= 4 i.e., 4 blocks required for the computation means 4 * 256= 1024 total thread or division(1024, 32)= 32 total warp needed. Not necessary all thread are active at a time due to memory latency, that's why we need to do proper scheduling of warp and we are going to schedule the warp based on the register behaviour change.

The dataset we had used is Rodinia dataset: It is a collection of parallel programs which is helpful in heterogeneous computation viz. BFS is graph traversal, kmeans is dense linear algebra, Huffman is finite state machine, Hotspot is Structured domain, etc.

Secondary , for matrix computation data is taken as random input whose range is [0,1024].

3.3 Procedure and Techniques

Proposed algorithm:

Step:1

int sumofBitDifferences(int ar[], int n)

result i=0

for:i (0 to 32) //traversing all bit

// count iḱh set bit

counter k=0

for:j (0 to n)

checking if ar[j] (1<<i) counter++

answer += (counter * (n - counter) * 2)

return answer

Step:2

Warp formation: pick(minHammmingDistance(R1,R2,...Rm)

Step:3

warp scheduling

Step:4

Instrcutin fetch/decode, execute, memory and write-back task happened.

Step:5

Repeat step 1 to step 4 till the end of the program

Step:6

End of the program.

Bitwise Anding: if ((arr[j] (1 <<i)))count++ step is taking T(n)=O(n)

Explanation:

1→ 0 0 1

3→ 0 1 1

6→ 1 1 0

1st iteration:- 2(setbit) * 1(unset bit) * 2= 4

2nd iteration:-2(setbit) * 1(unset bit) * 2= 4

3nd iteration:-1(setbit) * 2(unset bit) * 2= 4

total= 12

Input Signature of thread 1 - IST1:

R1 (32 bit): 10001111111. . . . . 0000, R2, : 01100011111000111. . . ..0, R3,..., Rn : 0000001111 1111

Output Signature of thread: OST1:

R1 (32 bit): 01100011111000111. . . ..0, R2, : 01100011111000011. . . ..0, R3,..., Rn: 011011111000111. 011

IST and OST form nxn matix and where each element of matrix will give hamming distance between the (i,j) pair where i is ith input signature of the thread and j is the jth output signature of the thread.Plotting graph and taking the minimum Hamming distance pair (it is bit wise toggle) for the warp scheduling.

CHAPTER 4

Results and Discussions

Table 4.1 inferring Graph 4.1 which shown that as we increase the block size the computation time will decreases gradually that is taking 40% less time if we increase block size from 16 to 64. Block size 16 and 32 taking same computation time.

mxnxk	bs(16)	bs(32)	bs(64)
10x10x10	0.2	0.236	0.143
16x16x16	0.182	0.172	0.164
20x20x20	0.176	0.174	0.15
48x48x48	0.16	0.184	0.145
30x2024x20	0.346	0.412	0.204
1024x1024x1024	7.172	6.778	1.427
2048x2048x2048	50.854	46.581	4.474
4096x4096x4096	284.83	263.667	166.678

Table 4.1: Parallel

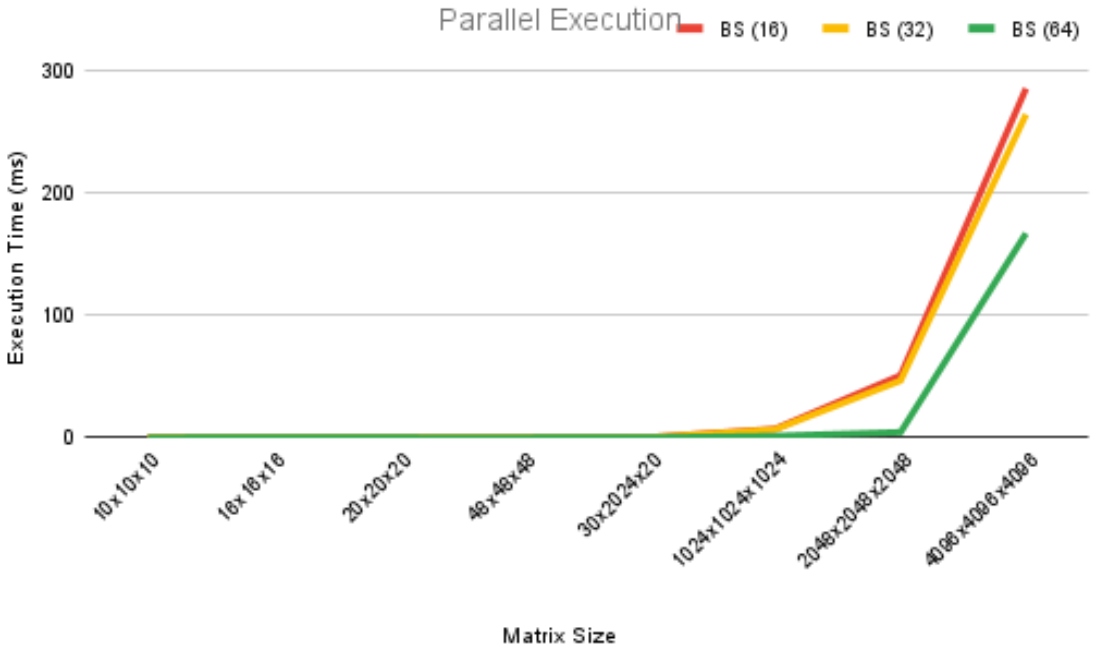


Figure 4.1: parallel execution

Table 4.2 inferring Graph 4.2 which shown that as we increase the block size the computation time will increases or decreases alternatively. Block size 32 performed well in view of computing time.

mxnxk	bs(16)	bs(32)	bs(64)
10x10x10	0.004	0.004	0.004
16x16x16	0.012	0.014	0.012
20x20x20	0.024	0.022	0.024
48x48x48	0.29	0.348	0.417
30x2024x20	3.733	1.98	3.925
1024x1024x1024	6906.182	7127.26	7301.486
2048x2048x2048	76327.421	75924.375	79832.91
4096x4096x4096	2009658	1990100.25	2077223.75

Table 4.2: Baseline

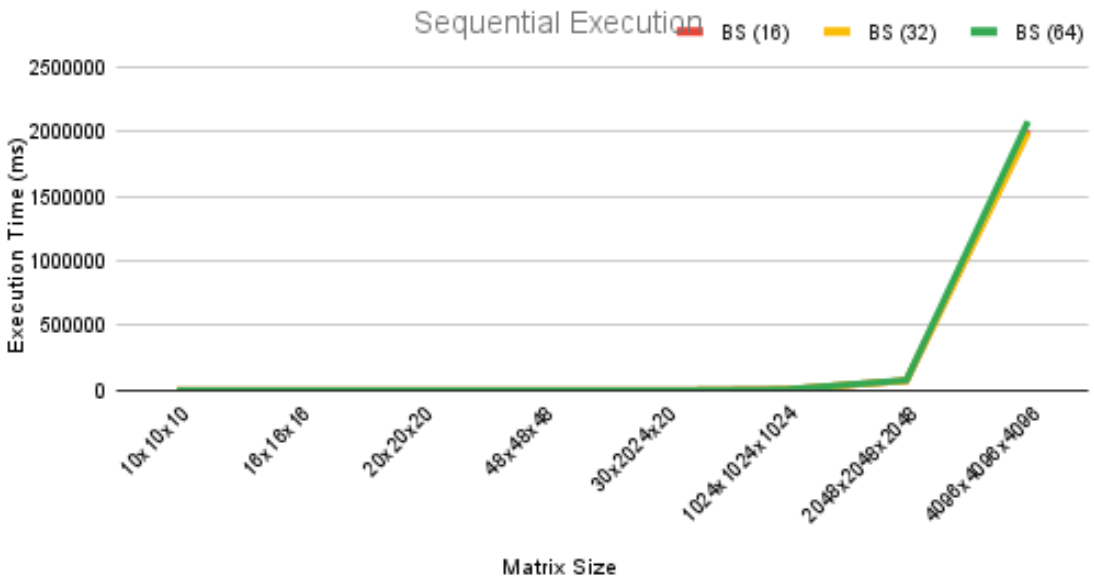


Figure 4.2: baseline execution

Table 4.3 inferring Graph 4.3 which shown that as we increase the block size, beginning speedup increases for all possible block size but later the 32 block size performed well because lesser number of threads are seating ideal due to lesser memory request as compared to the 64 block size computation, so the speedup decreases as we increases the block size more then 32.

Table 4.4 inferring Graph 4.4 which shown the relationship between the baseline and modified GPU which is clearly reflecting as we increase the matrix size the computation time increase gradually but baseline taking too much time as compared with mod. GPU.

mxnxk	bs(16)	bs(32)	bs(64)
10x10x10	0.02	0.01694915254	0.02797202797
16x16x16	0.06593406593	0.08139534884	0.07317073171
20x20x20	0.1363636364	0.1264367816	0.16
48x48x48	1.8125	1.891304348	2.875862069
30x2024x20	10.78901734	4.805825243	19.24019608
1024x1024x1024	962.9366983	1051.528474	5116.668535
2048x2048x2048	1500.912829	1629.943003	17843.74385
4096x4096x4096	7055.640206	7547.779017	12462.49505

Table 4.3: Speedup

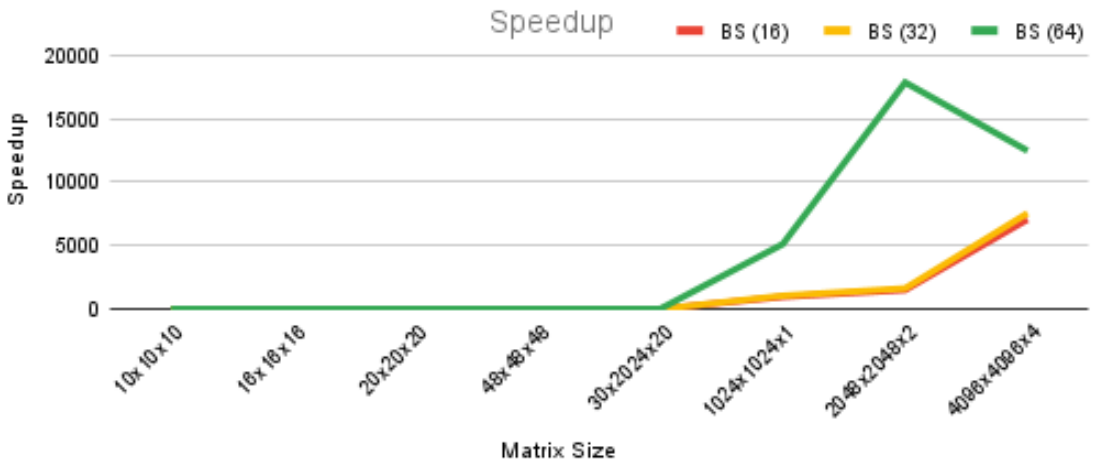


Figure 4.3: speedup curve: gpu vs baseline

mxnxk	bs(16)	bs(16)
10x10x10	0.2	0.004
16x16x16	0.182	0.012
20x20x20	0.176	0.024
48x48x48	0.16	0.29
30x2024x20	0.346	3.733
1024x1024x1024	7.172	6906.182
2048x2048x2048	50.854	76327.421
4096x4096x4096	284.83	2009658

Table 4.4: Block-size based comparision between Baseline and GPU

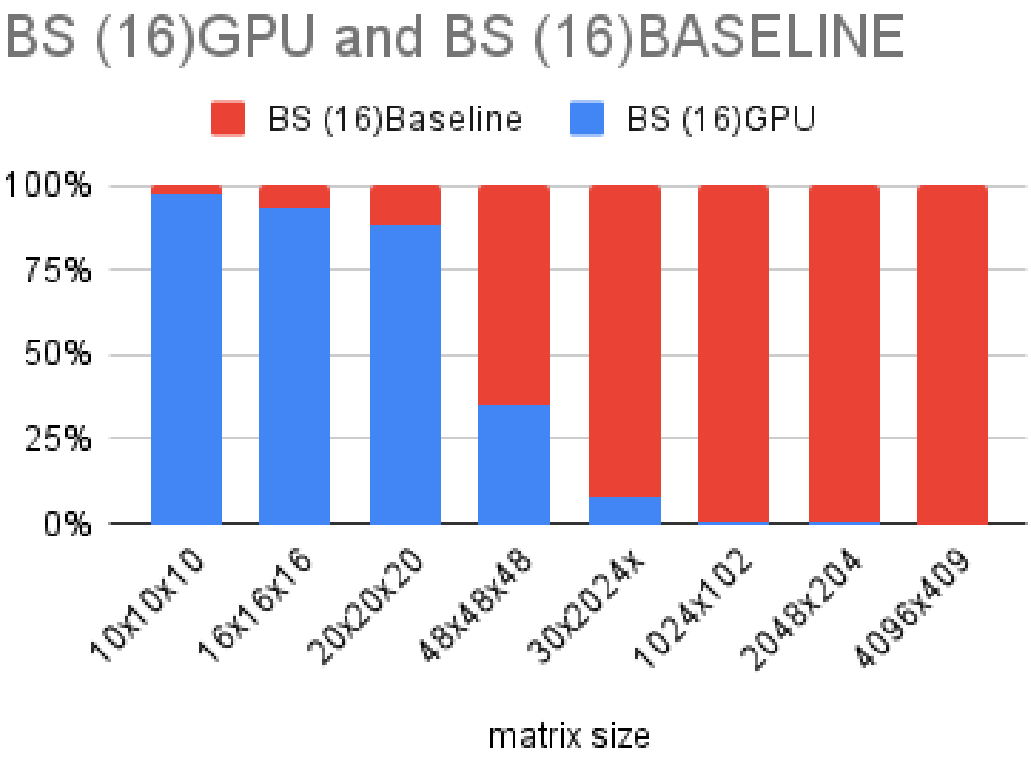


Figure 4.4: block size based comparision: gpu vs baseline

CHAPTER 5

SUMMARY AND CONCLUSION

Which will be further used for scheduling; the result shows that the performance of the GPU will be increased by 40% for the most of the input data-set which is not a great but is a good number. Power analysis is missing here due to Our proposed methods proof can be given after testing the work on GPGPU simulator, which, unfortunately, we could not complete in the given time frame. There is a lot of scopes here to understand the register behavior, which will help in forming warp, and scheduling them more wisely will improve the performance and power of GPGPU for intensive load computing.

REFERENCES

1. **M. Abdel-Majeed**, **D. Wong**, and **M. Annavaram**, Warped gates: Gating aware scheduling and power gating for gpgpus. *In 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013*a*.

2. **M. Abdel-Majeed**, **D. Wong**, and **M. Annavaram**, Warped gates: Gating aware scheduling and power gating for gpgpus. *In 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013*b*.

3. **H. Aghilinasab**, **M. Sadrosadati**, **M. H. Samavatian**, and **H. Sarbazi-Azad**, Reducing power consumption of gpgpus through instruction reordering. *In Proceedings of the 2016 international symposium on low power electronics and design*. 2016.

4. **A. Bakhoda**, **G. L. Yuan**, **W. W. Fung**, **H. Wong**, and **T. M. Aamodt**, Analyzing cuda workloads using a detailed gpu simulator. *In 2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009.

5. **W. W. Fung**, **I. Sham**, **G. Yuan**, and **T. M. Aamodt**, Dynamic warp formation and scheduling for efficient gpu control flow. *In 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007.

6. **E. Ipek**, **M. Kirman**, **N. Kirman**, and **J. F. Martinez**, Core fusion: accommodating software diversity in chip multiprocessors. *In Proceedings of the 34th annual international symposium on Computer architecture*. 2007.

7. **A. Jog**, **O. Kayiran**, **N. Chidambaram Nachiappan**, **A. K. Mishra**, **M. T. Kandemir**, **O. Mutlu**, **R. Iyer**, and **C. R. Das** (2013). Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance. *ACM SIGPLAN Notices*, **48**(4), 395–406.

8. **O. Kayiran**, **A. Jog**, **A. Pattnaik**, **R. Ausavarungnirun**, **X. Tang**, **M. T. Kandemir**, **G. H. Loh**, **O. Mutlu**, and **C. R. Das**, μc-states: Fine-grained gpu datapath power management. *In 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2016*a*.

9. **O. Kayiran**, **A. Jog**, **A. Pattnaik**, **R. Ausavarungnirun**, **X. Tang**, **M. T. Kandemir**, **G. H. Loh**, **O. Mutlu**, and **C. R. Das**, μc-states: Fine-grained gpu datapath power management. *In 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2016*b*.

10. **M. Khairy**, **Z. Shen**, **T. M. Aamodt**, and **T. G. Rogers**, Accel-sim: An extensible simulation framework for validated gpu modeling. *In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

11. **J. Leng**, **T. Hetherington**, **A. ElTantawy**, **S. Gilani**, **N. S. Kim**, **T. M. Aamodt**, and **V. J. Reddi** (2013). Gpuwattch: Enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, **41**(3), 487–498.

12. **Y. Park**, **J. J. K. Park**, **H. Park**, and **S. Mahlke**, Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability. *In 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012.