Optimization Techniques for Speedup in a Parallel Algorithm

Supervisor: Dr. Jia Uddin

Fairuz Faria	13201050
Tahmid Tahsan Obin	13201057
Shah Md. Nasir Rahat	13241006
Tanzim Islam Chowdhury	14301074

Department of Computer Science and Engineering
BRAC University

26th December 2017

Declaration

We hereby declare that this thesis is based on our own work and results we obtained. All other resources used have been acknowledged with proper reference, which can be found at the end of the paper in the reference section. This thesis, neither in part nor in whole has been submitted to any other University or Institution for the award of any degree or diploma.

Signature of Supervisor:

_____

Dr. Jia Uddin

Signature of Authors:

_____

Fairuz Faria

_____

Tahmid Tahsan Obin

_____

Shah Md. Nasir Rahat

_____

Tanzim Islam Chowdhury

Acknowledgement

First of all, we would like to express our deepest gratitude to almighty Allah for giving us the ability to start and successfully finish the thesis work.

Secondly, we would like to express our sincere gratitude to our advisor DR. Jia Uddin for the continuous support in our research, for his guidance, motivation, patience and colossal knowledge. Without his unparalleled support, it would have been an immense tough work to conduct.

Finally, we would like to express our sincere gratefulness to our beloved families and friends for their love and care. We are also thankful to them for helping us directly or indirectly to complete our thesis. Furthermore, we would also like to acknowledge the numerous assistance we received from diverse online and offline research works.

Table of Contents

Chapter 3

Methods and Implementation Detail

Chapter 4

Experimental Result

Chapter 5

Conclusion

List of Figures

List of Tables

List of Abbreviations

1D	One Dimension
2D	Two Dimension
3D	Three Dimension
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CMP	Chip Multiprocessors
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
GPGPU	General Purpose Compute Unified Device Architecture
GPU	Graphics Processing Unit
PCI	Peripheral Component Interconnect
RLE	Run-Length Encoding
SMM	Maxwell Streaming Multiprocessor
SMX	Streaming Multiprocessor

ABSTRACT

For our thesis we study about conditions of a good parallel algorithm which greatly increases efficiency in a program, and show that it is possible to implement Lossless Data Compression using the Run Length Encoding algorithm in parallel architecture. Lossless compression is when the original data that was compressed will not get lost after the data is being decompressed, hence without any loss of data we hope to accomplish a massive reduction in execution time by applying parallelism to this algorithm. Many compression algorithms are typically executed in CPU architectures. In our work, we mainly focused on utilizing the GPU for parallelism in data compression. Hence an implementation of Run Length Encoding algorithm is used by the help of NVIDIA GPUs Compute Unified Device Architecture (CUDA) Framework. CUDA has successfully popularized GPU computing, and General Purpose Compute Unified Device Architecture applications are now used in various systems. The CUDA programming model provides a simple interface to program on GPUs. A GPU becomes an affordable solution for accelerating a slow process. This algorithm is convenient for the manipulation of a large data set as oppose to a small one as this technique can increase the file size greatly. Furthermore, this paper also presents the efficiency in power consumption of the GPU being used compared to a CPU implementation. Lastly, we observed notable reduction in both execution time and power consumption.

Chapter 1
Introduction

Integration of increasing number of processing cores on chip multiprocessors (CMPs) is currently considered a trend in computer architecture design. Increasing the CPU clock frequency does no longer adds a significant performance enhancement due to much increased power dissipation from the CPU. GPUs are an example of many core architectures. GPUs are evolved to carry out general purpose computation presenting large amount of simple in-order processing elements. The parallel GPUs are increasingly used for acceleration of inherently data-parallel functions, such as motion estimation algorithms and image transforms.

Manipulation of huge data efficiently in terms of execution time and power consumption is becoming imperative in computer science. To manage huge data sets, there are different compression algorithms for different data sets like audio, video, image, texts etc. Using efficient algorithms alongside parallel computing can produce a huge speedup of the execution time and thus reducing the power consumption of the compression techniques. The way a parallel solutions work is simply by the distribution of tasks among different processors, which then process the designated data set using standard serial compression algorithms. Faster and efficient compression allows for much better performance in memory space management or simply data processing in a slower network bandwidth.

In this paper we present an implementation of Run Length Encoding algorithm on NVIDIA GPUs, which is a data compression technique. It is more of a serialized algorithm having number of conditional branches which is not efficient enough for GPU threads. Our work focused on implementing RLE in parallel on NVIDIA GeForce GTX 1050 Ti GPU using CUDA framework to observe significant speedup in execution time and reduction in power consumption compared to a CPU implementation.

1.1 Contribution Summary

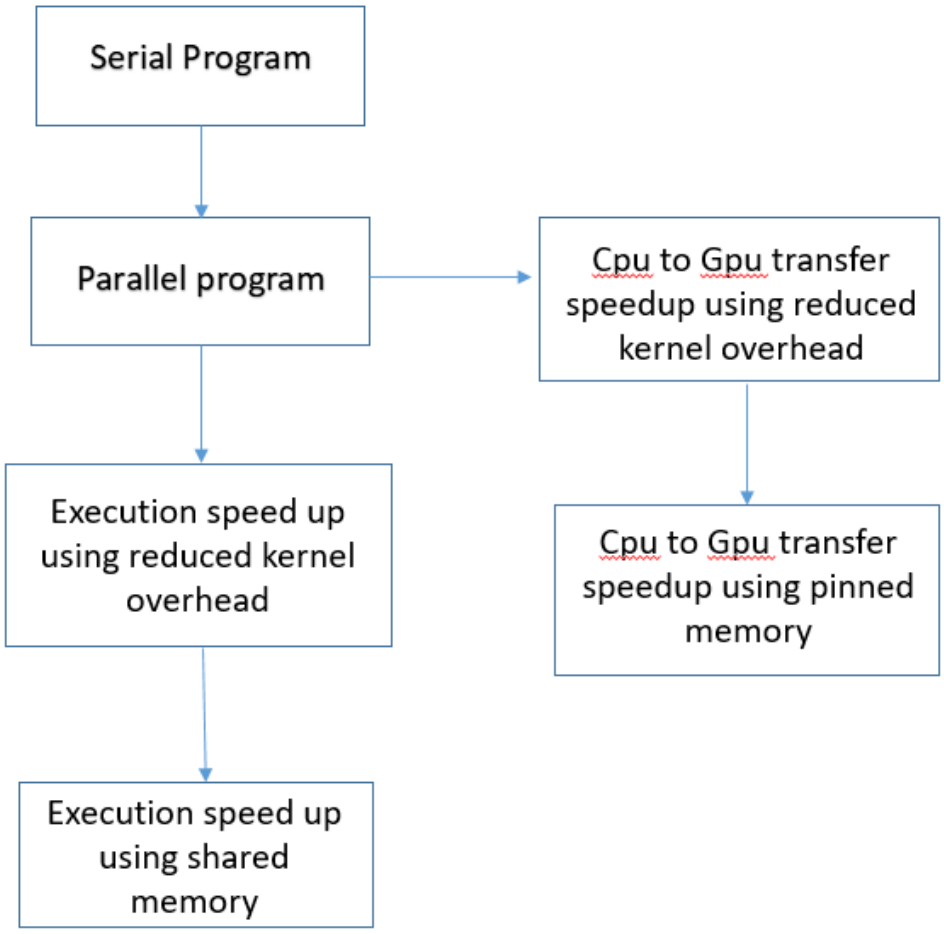
In our thesis work, we have used CUDA toolkit and CUDA C programming language. Among many lossless data compression techniques we have chosen Run-Length Encoding (RLE)

algorithm. We implemented methods like shared memory, kernel overhead, pinned memory optimization in RLE to optimize the execution time.

1.2 Motivation

The importance of data transfer is growing day by day with increasing demand for sharing data online. This importance in turn placed priority for a faster more efficient compression technique such as Run-Length Encoding algorithm, which is an algorithm for data compression. But efficiency is the main priority here in order to save network bandwidth consumption. In our paper we understood the importance of an efficient more optimized version of compression algorithm and to achieve that we need to apply parallelism followed by some optimization techniques on the previous algorithm to run more efficiently with less execution time than before. Hence those optimizations could save a lot of bandwidth consumption if this was to be implemented in the future as an application for online compression for file transfers.

1.3 Methodology



1.4 Thesis Orientation

The rest of the thesis is organized as follows:
Chapter 2 includes the background information of Graphics Processing Unit (GPU), CUDA.

Chapter 3 represents the methods and implementation details for RLE in parallel along with other optimization techniques.

Chapter 4 shows the experimental result and comparison.

Chapter 5 is the conclusion of the thesis and also states the future research directions.

Chapter 2

GPU

2.1 Background Information of GPU

GPUs (Graphics Processing Unit) are computer processors, an electrical component specialized for fast processing of graphical data or generating images in a frame buffer to be displayed in various display devices. GPUs are located on plug-in cards, in a chipset on the motherboard or in the same chip as the CPU. In the past, there were no dedicated processing units for graphical data, rather all graphics operations and calculations were processed by the CPU alone. As software demands increased and graphics became more important (especially in video games), a need arose for a separate processor to render graphics. On August 31, 1999, NVIDIA introduced the first commercially available GPU for a desktop computer, called the GeForce 256 [30]. Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.

2.2 GPU Architecture

Heterogeneous architectures enable throughput or latency optimization. Central Processing Units (CPU) and computational accelerators, for example, Graphics Processing Units (GPU) combines to form the heterogeneous architecture. Workstations of many engineering or scientific labs have supercomputers which uses heterogeneous architecture. This leads to CPU being used for other works and users able to do large parallel computing capabilities. But mapping the computational algorithm to heterogeneous architecture makes of a programming challenge and difficulty in using the GPU efficiently. In the last ten years, parallel global optimization of a GPU improved their performances significantly. Many intensive solutions are achieved due to the high performance of modern GPUs and parallel processors [2]. Number of cores or scalar processors makes up a multiprocessor and many streaming multiprocessors makes up the NVIDIA GPU. There are different NVIDIA GPU architectures available such as Kepler, Fermi Tesla etc.

2.2.1 Till NVIDIA G70

Until the latest NVIDIA G70 GPU and their previous generations of architecture came to the market, vertex and pixel shading were handled in multiple dedicated units. Array was used, where the top eight shaders were used for vertex processing and the processing of pixel was done in the middle of the array with twenty four shaders. The pixel shaders had to sit idle until data came from the vertex shaders which is a loss. For this reason NVIDIA shifted their vision to new architecture [4].

2.2.2 G80

The idleness of the shaders were solved when NVIDIA developed their new G80 architecture. It dynamically allocates the number of pixel and vertex shaders as per the current application required thus removing the idle issue of hardware. The GPU architecture of G80 was the first architecture that integrated one hundred and twenty eight processing elements between eight cores of shaders [3]. Pipeline was a major part of the previous architectures but this was no longer a part of the G80 architecture. Besides, before a pixel is presented to the frame buffer, the pipeline loopback into itself. In the earlier versions, each shaders type had few cores allocated for them but this is not the case for G80. All execution units get hold of the shaders due to the prioritization controlled by the scheduler. This improves the performance as the hardware can increase the number of vertex shaders along the cores. CUDA (Compute unified Device Architecture) was introduced by this architecture and the first C-based development environment for GPU's. Another improvement came in the name Tesla [4].

2.2.3 Tesla Architecture

The first microarchitecture made by NVIDIA by implementing unified shader model is Tesla. It brought notable changes in GPU functionality and compatibility. For instance, separate functional units like pixel shaders to homogeneous collection of universal floating point processors, known as stream processors. Tesla is used in GeForce 8 Series, GeForce 9 Series and others. Tesla was the first GPU which had unified shader with 128 processing elements which distributed in 8 shader core [5].

2.2.4 Fermi Architecture

NVIDIA developed a GPU microarchitecture which is a successor to the Tesla microarchitecture, named Fermi. Since the original G80, the Fermi architecture is one of the most enormous step forward towards GPU architecture. To create the World's First Computational GPU, which needed a completely new approach, NVIDIA made improvements on different areas of the Fermi architectures like True Cache Hierarchy, Faster Context Switching, ECC support, Double Precision Performance, Faster Atomic Operations and More Shared Memory. NVIDIA also collected vast user feedback on GPU computing since the introduction of G80 and GT200. The first Fermi based GPU was implemented with 3.0 billion transistors and features up to 512 CUDA cores. 512 CUDA cores are organized in 16 SMs of 32 cores each for executing floating point or integer instructions. Femi has six 64-bit memory partitions for a 384-bit memory interface which supports up to a total of 6GB of GDDR5 DRAM memory. PCI express, a host interface which connects the GPU to the CPU. Thread blocks are distributed to SM thread schedulers by Giga Thread global scheduler [29].

2.2.5 Kepler Architecture

After Fermi, NVIDIA presented the Kepler GPU microarchitecture. Power efficiency was the main focus for NVIDIA's engineering team. GeForce arrangement from 600 to 700 and some of 800 arrangement utilized Kepler design and all are manufactured in 28nm. Efficiency, programmability and performance was the main priority enforced by NVIDIA on Kepler architecture on the other hand its predecessors focused on increasing performance on compute and tessellation. Unified GPU clock, simplification in static schedule instruction and prominent importance on performance per watt helped Kepler to achieve such efficiency. The key to Kepler high performance is the addition of cores instead of shader clock on previous versions. Cores are more power-friendly. Two Kepler core use 90% of power of one Fermi core but unified GPU clock scheme diminishes 50% power consumption [7].

Later on Maxwell engineering replaced the Kepler architecture.

2.2.6 Maxwell Architecture

Maxwell is the next GPU microarchitecture in the development chain of NVIDIA after Kepler microarchitecture. Newer versions of Maxwell architecture use new chip code unlike the older

versions used chip of Kepler. These chips gave consumers few features as NVIDIA focused more on the power efficiency of GPU. In Kepler the L2 cache was 256 KiB whereas on Maxwell, it is 2MiB, which decreases the want for more memory bandwidth. Therefore, reducing the memory bus from 192 bit in Kepler to 128 bit in Maxwell; a further power saving. In Maxwell, the streaming multiprocessors were redesigned, partitioned and named SMM as of the previous layouts from Kepler. Despite the texture units and FP64 CUDA cores still shared, the structure of the warp scheduler was the same as on Kepler. On the other hand most execution unit layout were divided to allow each warp schedulers in an SMM control one set of 32 FP32 CUDA cores, one set of 8 load/store 15 units and one set of 8 special function units. In comparison to Kepler, where each SMX has four schedulers that schedule to a shared pool of execution units [8].

2.2.7 Pascal Architecture

After Maxwell architecture, Pascal architecture is the newest addition to the chain of NVIDIA GPU architecture, introduced in April 2016 with the GP100 chips. The name is derived from the 17th century, French mathematician, physicist Blaise Pascal. In Pascal , 64 CUDA cores are used in the SM (Streaming multiprocessor), whereas there were 128 cores in Maxwell, 192 cores in Kepler, 32 cores in Fermi and 8 cores in Tesla. The GP100 SM is divided in two processing blocks where each block contains 32 single precision CUDA Cores, a wrap scheduler, an instruction buffer, two dispatch units and two texture mapping units [9]. Most importantly it supports CUDA Compute Capability 6.0.

2.2.8 Titan Architecture

The newest and most advanced NVIDIA GPU till date is the Titan V Volta. It is based on the GV100 GPU architecture which features a total of 5120 CUDA cores along with 320 texture units. On contrary, there are 640 Tensor Cores inside the Volta GPU also. This is the exact same amount of cores featured on the Tesla V100. These arrangements are made for maximizing the performance of deep learning, for algorithms which are related to artificial intelligence and clocked to 1200 MHz base and 1455 MHz boost and comes in a 250W package. Despite the heavy specs it requires only one 8 and 6 pin power connector to boot [27].

2.3 CUDA Overview

CUDA(Compute Unified Device Architecture) which is a NVIDIA GPU architecture that is in GPU card made for programming general purpose computation on parallel GPU architectures, launched in 2007 [31]. It has positioned itself as a whole new meaning for general purpose computing with GPUs. CUDA uses extension of C++ known as CUDA C for programming purpose. CUDA provides advantage of huge computational power to the programmer and it is famous among them since it gives a lot of freedom to work on. Depending on GPU models, CUDA has many cores which works side by side. Here cores can communicate and also they can exchange information with each other so that, running multithreaded application there is no need for streaming computing in GPU [32]. As previously mentioned CUDA uses C programming language, as GPU possesses a large amount of threads and each can execute the same code in parallel but here all the threads are executed using same code but different data. CUDA programs consist of one or two parts that is 16 executed either on host (CPU) or device (GPU). Depending on how much parallelism there is, it is better to execute codes in CPU if it involves little parallelism but GPU becomes a much better alternative when the code involves huge parallelism. This is due to transfers or copy of data from host to device or device to host which have a negative impact on both execution time and power consumption, hence simple codes are not meant for parallel execution as it may take longer due to these data transfers but executing complex codes in a GPU are a far better option than executing on the CPU as it overcomes data copy speed and hence boosts performance.

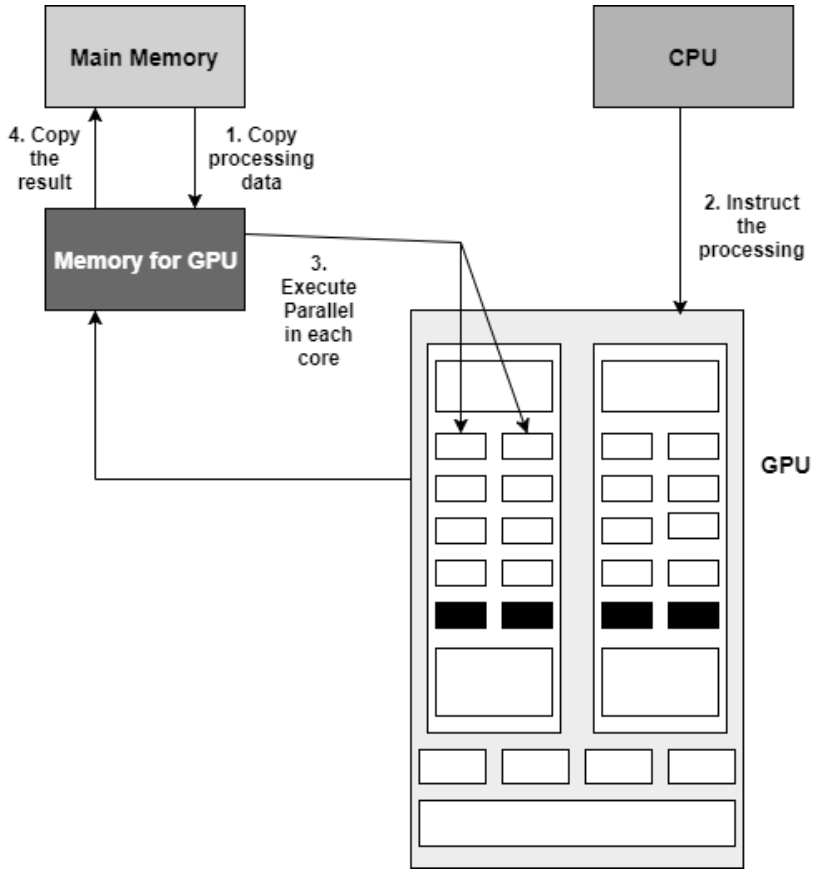


Figure 1: Processing flow on CUDA

In Figure 1 [26], we can see the processing flow on CUDA. Every algorithm has two parts, data dependent and data independent. Data dependent part is not sent to GPU as it takes time to copy from CPU to GPU which is not beneficial. On the other hand data independent part is sent to GPU for parallelism which makes the algorithm efficient.

2.3.1 Units of CUDA

In this part we are going to discuss about those basic units of CUDA, kernel, thread, block, grid that it uses to execute and run a program.

2.3.1.1 Kernel

In CUDA CPU and its memory is known as host and GPU and its memory refers to device. Code run on host can manage both memory on host and device and launches kernel, special function, which are executed on device. A kernel is called to execute parallel functions and defined by using _global_ declaration specifier. For a given kernel call, a specific kernel is being executed by how many number of CUDA threads is specified using a new <<<...>>> execution configuration syntax. A particular thread ID is assigned to each thread that executes the kernel that is usable through the included threadIdx variable [17]. Shown in Figure 2

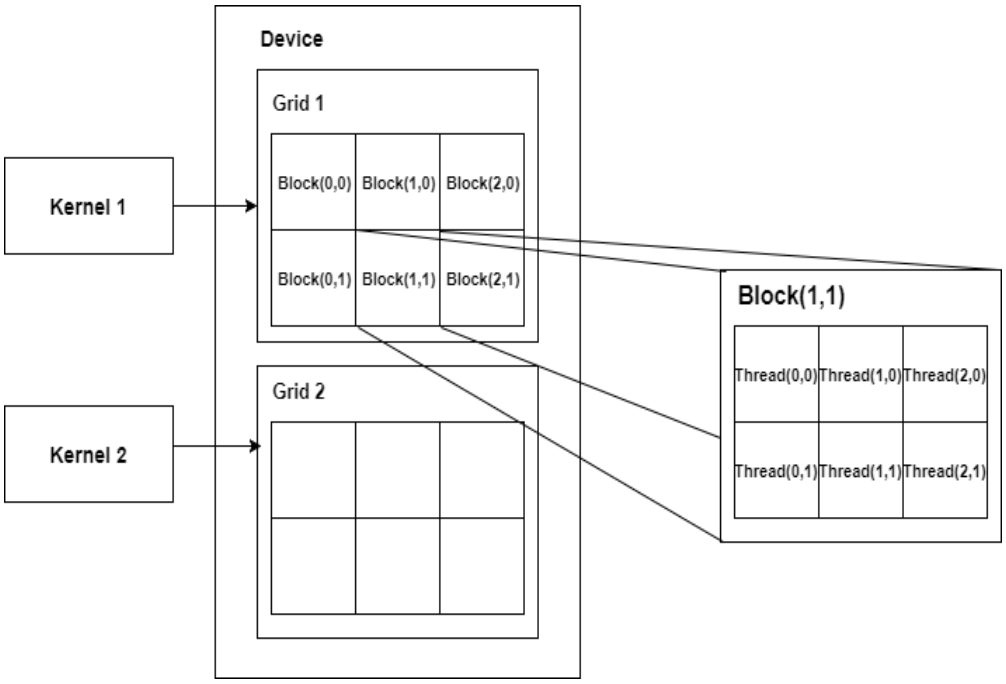


Figure 2: Kernel

2.3.1.2 Grid

A group of threads all running the same kernel is called a grid. Threads here are not synchronized. One grid is able to make every call to CUDA from CPU. Starting a grid on CPU is a synchronous operation but multiple grids can run at once. For maximum efficiency, several

grids are being used by GPUs as grids cannot be shared between GPUs on multi-GPU systems [23], which is shown in Figure 3.

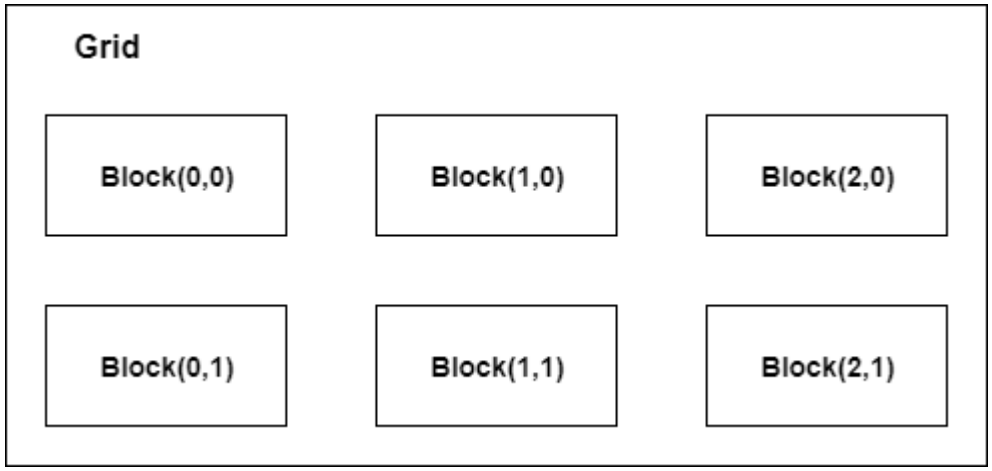


Figure 3: 1D Grid

2.3.1.3 Block

Blocks make up a grid. Each block is a logical unit containing a number of coordinating threads, a certain amount of shared memory. Blocks are not shared between multiprocessors just as grids are not shared between GPUs. All blocks in a grid use the same program. To identify the current block, a built in variable "blockIdx" can be used. Block IDs can be 1D or 2D (based on grid dimension). Usually there are 65,535 blocks in a GPU [23]. Shown in Figure 4

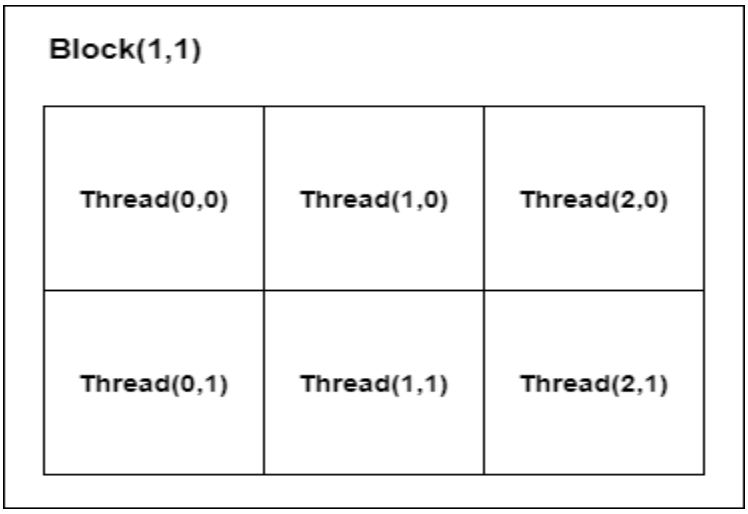


Figure 4: 1D Block

2.3.1.4 Thread

Blocks are composed of threads. Individual cores of the multiprocessors runs all the threads, but unlike grids and blocks, they are not restricted to a single core. Like blocks, each thread has an ID (threadIdx). Based on block dimension, thread IDs can be 1D, 2D or 3D. The thread id is relative to the block it is in. Threads have a certain amount of register memory. Usually there can be 512 threads per block [23].

2.3.2 Memory Units in CUDA

2.3.2.1 Global Memory

This memory is used for both read and write but is slow on copying. It needs sequential and aligned 16 byte read/writes to be fast. This memory is also known as device memory [23].

2.3.2.2 Shared Memory

This memory is common for all threads in a block and those blocks can use this memory for read and write operation. Its size is smaller than global memory [23].

2.3.2.3 Constant Memory

It is slow but with cache and read only memory. Constants and kernel arguments are stored here [23].

2.3.2.4 Texture Memory

It is a read only memory and its cache is optimized for 2D spatial access pattern [23].

2.3.2.5 Local Memory

It is generally used for whatever does not fit into register but it is slow and does not have cache. Allows automatic coalesced reads and writes [23].

2.3.2.6 Registers

This is the fastest memory among all. One set of register memory is given to each thread and it uses them for fast storage and retrieval of data like counters, which are frequently used by a thread [23]. Shown in Figure 5.

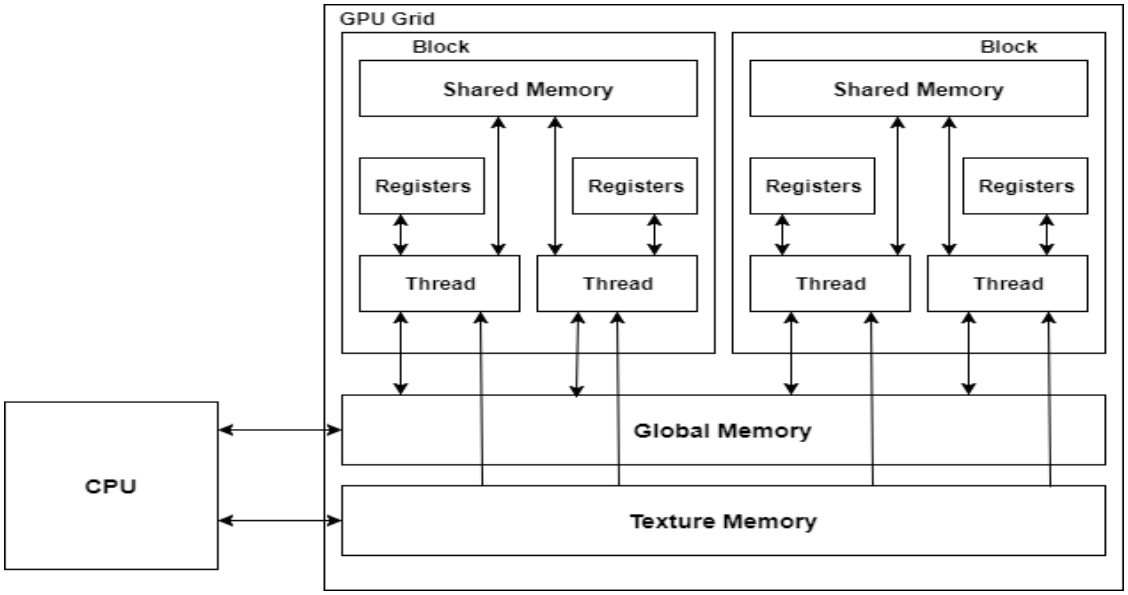


Figure 5: Memory Model of CUDA

2.4 Difference between CPU and GPU

Processing of tasks by CPU and GPU is different from one another, as there are few CPU cores that are responsible for sequential serial processing whereas the GPU supports thousands of tiny and much more efficient cores which can carry out numerous concurrent processes. Thus the massively parallel architecture enables GPU to carry out functions in a much faster and more efficient way than CPU [15]. There are more differences between CPU and GPU which are given in the Table 1.

Table 1: Differences between CPU and GPU [23]

CPU	GPU
1. Hides memory latency via hierarchy of caches	1. Memory latency not hidden by large cache
2. Really fast caches (great for data reuse)	2. Lots of math units
3. High performance on a single thread of execution	3. High throughput on parallel tasks
4. CPUs are great for task parallelism	4. GPUs are great for data parallelism
5. Lots of different processes/threads	5. Run a program on each fragment/vertex
6. Fine branching granularity	6. Fast access to onboard memory
7. CPU optimized for high performance on sequential codes (caches and branch prediction)	7. GPU optimized for higher arithmetic intensity for parallel nature (Floating point operations)
8. Most die area used for memory cache	8. Most die area used for ALUs

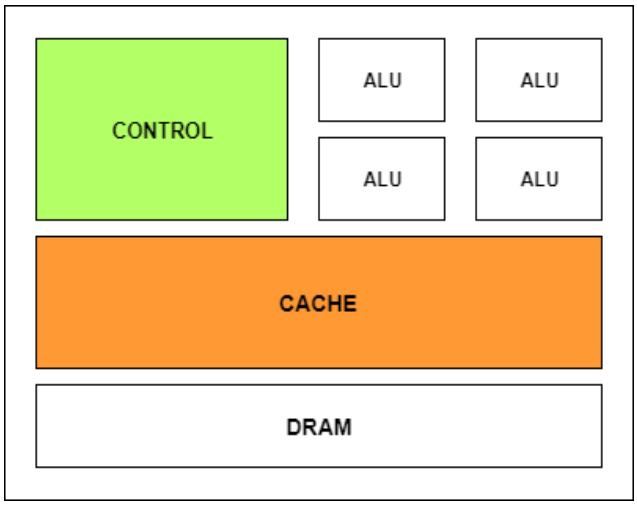


Figure 6: CPU architecture

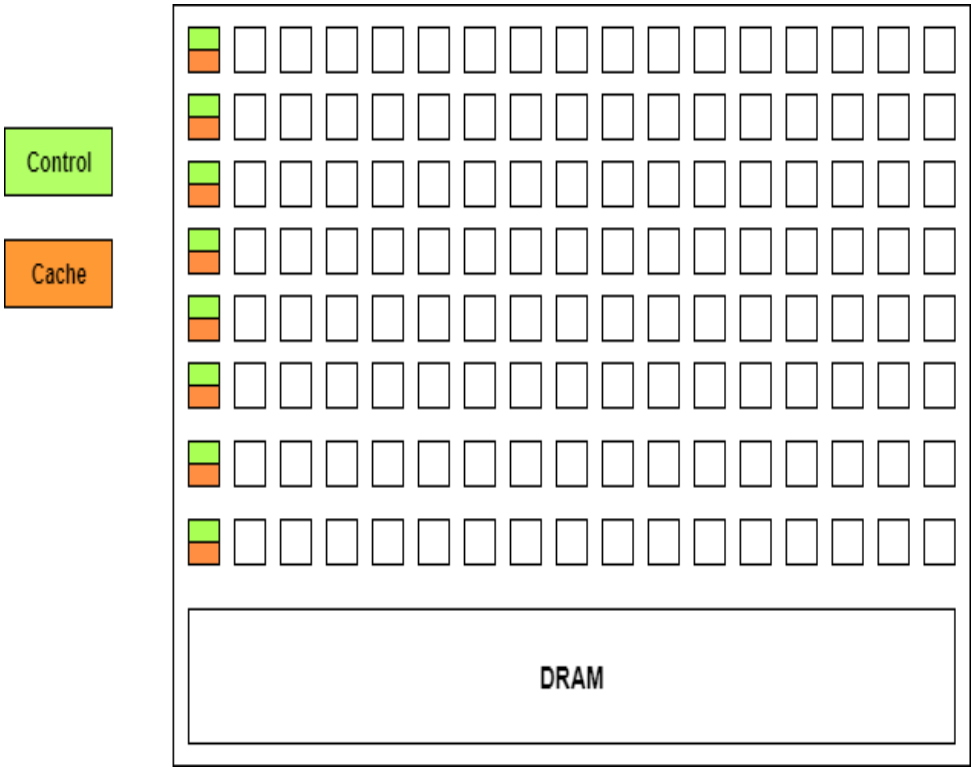


Figure 7: GPU architecture

Chapter 3
Methods and Implementation Detail

3.1 Run Length Encoding Algorithm

Run-length encoding (RLE) is a very straightforward lossless data compression algorithm where runs of data which is sequence of same data value exists in consecutive data elements stored as a single data value and count, as opposed to the original runs. This technique becomes more convenient where the data set contains many such runs for example, in simple graphics images animations, icons and line drawings. Its usefulness or the performance of the program deteriorates with files that don't have many runs as it will lead to an increase in the file size substantially [28].

3.2 How RLE Works

Run Length Encoding works by replacing the sequences of same string of characters. It replaces the sequences of repeating string or same data values within a file, called a *run*, to an encoded two bytes. The first byte is used for the representation of the number of characters in the run, called the *run count*. An encoded run in custom may contain 1 to 128 or 256 characters. However, the run count is the number of characters minus one which is a value in the range of 0 to 127 or 255. Lastly, the second byte determines the value of the character in the run in the range of 0 to 255, known as the *run value*.

A simple example can be as follows: Suppose a string has the following characters; TTTTTTTTEEEEEESSSSiiiiiSSSSSSSSS.

After running RLE compression, the above string will be represented as 8T6E4S5i9S, called *RLE packet*. As a result, the 32-byte string would be compressed to ten (10) bytes of data. Every time there is a change in data set, there is change in the run characters accordingly [12][27]. The flowchart of RLE is given in Figure 8.

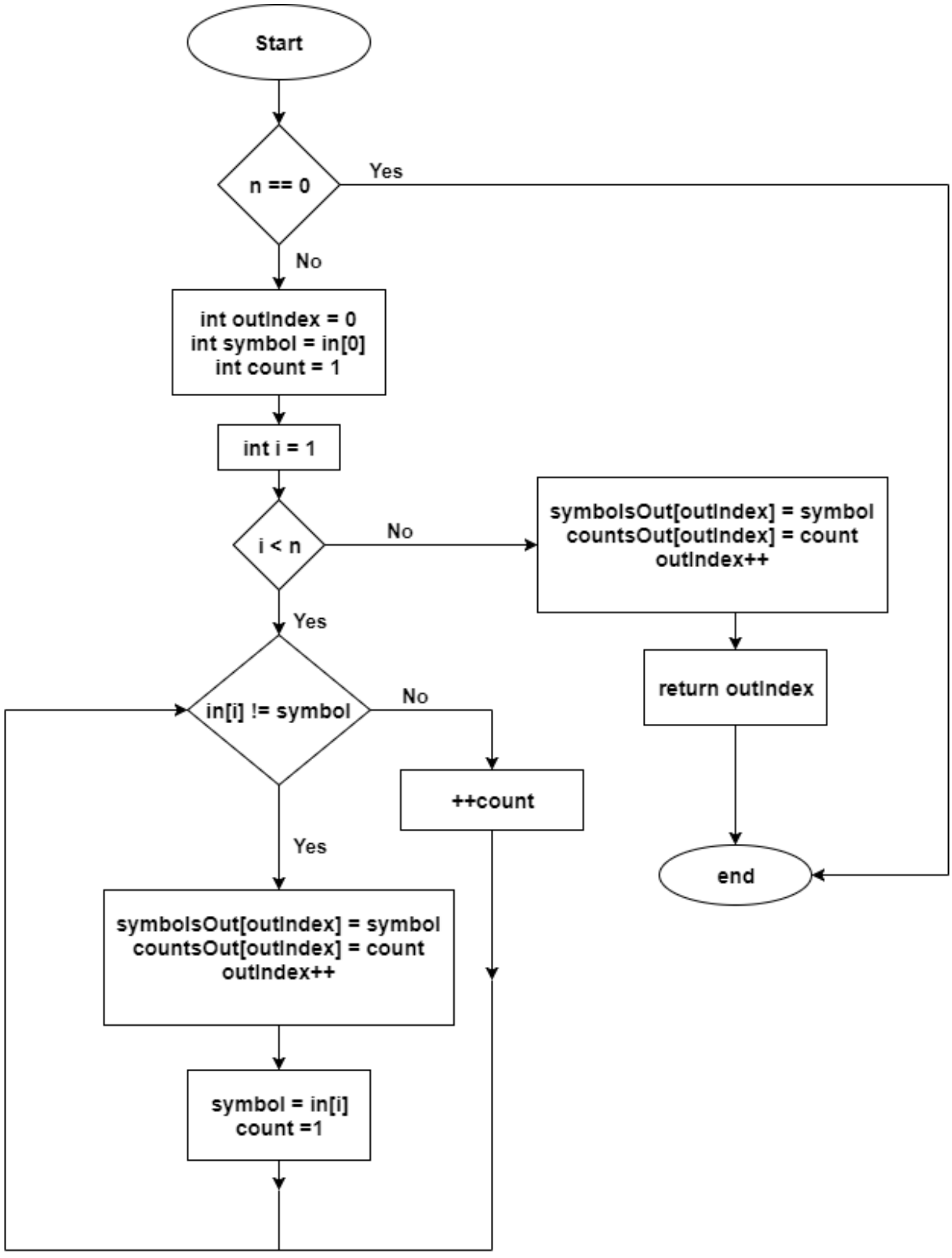


Figure 8: Flowchart of RLE (Run Length Encoding) Algorithm

The attributes in Figure 8 are as follows:

int* in - pointer for input array.

int n - size of the input array.

int* symbolsOut - pointer for symbols out array

int* countsOut - pointer for counts out array.

int outIndex - the last index for symbols out array.
int symbol - simple integer for comparing elements of the input array.
int count - keep count of the number of elements appearing [11].

3.3 Optimization Techniques

In the CUDA implementation of Run-Length algorithm, where we have taken the serial code and applied parallelism with techniques. To launch the kernels, Hemi library is being used where it will automatically choose grid and thread block size for the specific graphics card. Then the execution time efficiency of the parallel implementation is observed over the serial execution time [11]. For further optimization few of the techniques are used which are being discussed below.

3.3.1 Pinned Memory Optimization

In parallel computing, data must be transferred from the host (system) memory to the device (GPU) memory. After the data is processed, the results transfer back to the host from the device. Such back and forth transfers takes place many times and affects negatively on the execution time and the overall efficiency of the code.

Data allocations in CPU (host) are pageable by default. When transfer of data from CPU (host) to GPU (device) is invoked, the GPU cannot access the data directly from pageable host memory rather the CUDA driver assigns the host array to the pageable memory then the host data is copied to the pinned array then finally the transfer of the host data takes place from the pinned memory to the DRAM of the device as shown in Figure 9 [13].

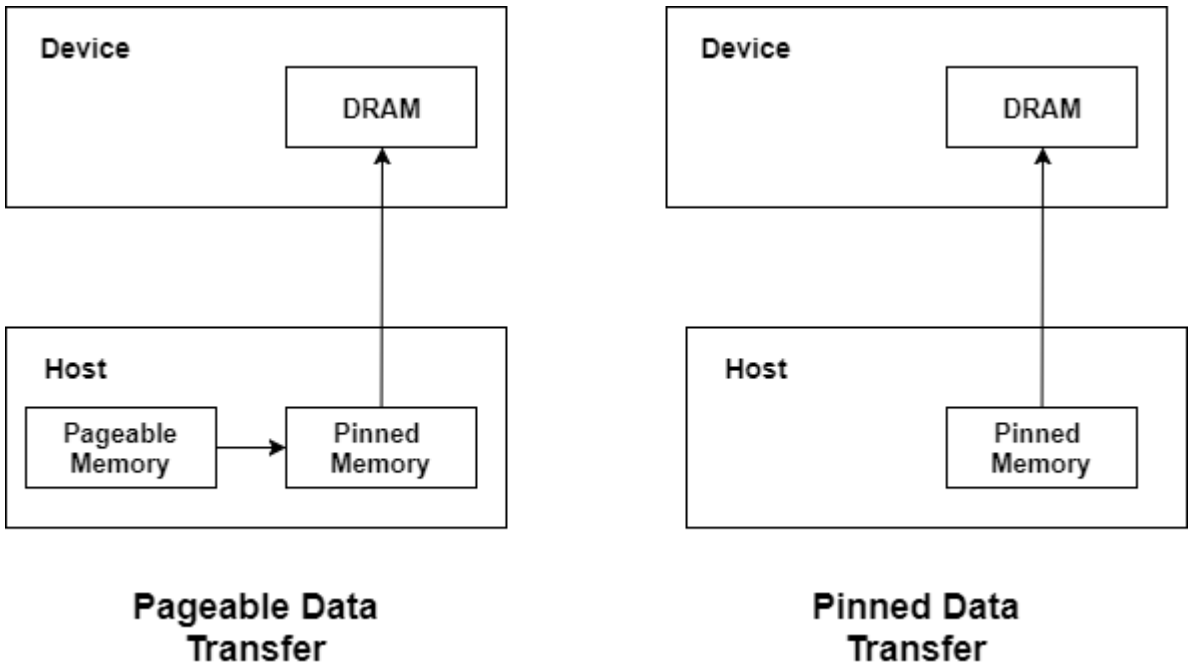


Figure 9: Pageable and Pinned Data Transfer

As shown in the Figure 9, it is possible to avoid data transfer between the pageable memory to the pinned memory by assigning the host array directly to the pinned memory. This reduces a significant data transfer time which ultimately improves the throughput. In CUDA C/C++, assigning pinned host memory is done using cudaMallocHost() or cudaHostAlloc(), and deallocate it with cudaFreeHost() [13].

We used the keywords cudaMallocHost() to allocate our input array with pointer h_in and then used cudamemcpy to copy our input array with pointer h_in to the array allocated on the GPU with pointer g_in. After that we reviewed the copying time using nvcc profiling tool.

3.3.2 Reduced Kernel Overhead

Cost in terms of execution time is severe and also a negative impact on performance is due to the overhead launching of kernels where the kernels are launched many times by the CUDA application. To minimize the impact of this overhead, the primary focus should be given on increasing the amount of work performed in each kernel call and minimizing the total number of

kernel calls. Subsequent kernel calls heavily improves the performance because kernel invocations are asynchronous in the recent CUDA API versions. Multiple kernel calls will be batched in the CUDA driver if developers can call kernels numerous times without intervening synchronization such as in memory transfers.

For our parallel algorithm we had 3 kernels in total. First we tried merging two and then all three. For small datasets the merge didn't seem that effective but after increasing it by a significant amount we achieved our desirable results [18].

3.3.3 Shared Memory

One way to make CUDA codes more optimized is to use the shared memory feature. As shared memory is located on chip it is much faster for accessing data. All threads in a thread block shares the shared memory as it provides a way for the threads to cooperate. Using shared memory is much faster compared to global or local memory as it is on chip, in fact the latency of shared memory is hundred times lower than uncached global memory latency given that no bank conflicts exists between the threads. All threads in a thread block are able to access the same shared memory because shared memory is assigned per thread block. Data from shared memory is loaded from global memory by threads in the same block so that other threads can have an access to the stored data [20].

While using shared memory we definitely benefited a lot but were limited to a data set of only 10,000 since it is cached memory. Since our testing GPU was a normal one 1050 ti we were not able to exceed this limit. But however if we had the latest hardware like the titan x we could have made great optimization with shared memory since its cache size can hold a lot more.

Chapter 4

Experimental Result

4.1 PC Configurations

To analyze the performance of CUDA implementation, we used a GeForce GTX 1050TI card with CUDA version 9 installed on a machine with Intel(R) Core(TM) i3-540 CPU running at 3.30GHz. The CPU implementation of RLE is also tested on the same PC.

4.2 Datasets

For our experiment we were able to use 3 sets only since trying to transfer data which is more than 10^8 results in memory error as it is too much to allocate for our for our Device memory.

- First set = 8 elements
- Second set = 10,000,000 elements
- Third set = 409,999,999 elements

4.3 Execution Speedup

For benchmarking, data allocation and uploading was done beforehand. This ensures that we will only be testing the actual performance of the algorithm on the GPU, and not the transfer operation performance from the CPU to the GPU, but we will get into that later.

Software used were Visual Studio for compilation and profiling was done with command line nvcc profiler.

At first we converted our serial program to parallel program. For parallel we used hemi library which allocated thread size and block according to the GPU's architecture hence much better than manual kernel allocation.

Below are the experimental results of converting the serial algorithm to our parallel algorithm shown in Table 2.

(Times faster was calculated by dividing the serial time result by the parallel time result.)

(Test results are in milliseconds)

Table 2: Comparison of CPU execution time and GPU execution time of RLE

	First-set	Second set	Third Set
Serial	1 millisecond	49 milliseconds	2791 milliseconds
Parallel	ms	0.88485 millisecond	39.476 milliseconds
Times faster	5.6	55.37	70

Observation of these results shows that parallel implementation of the serial algorithm helped speedup execution time by a significant amount. Also higher the dataset in value, more the speedup achieved. A bar chart illustration is given Figure 10, Figure 11 and Figure 12.

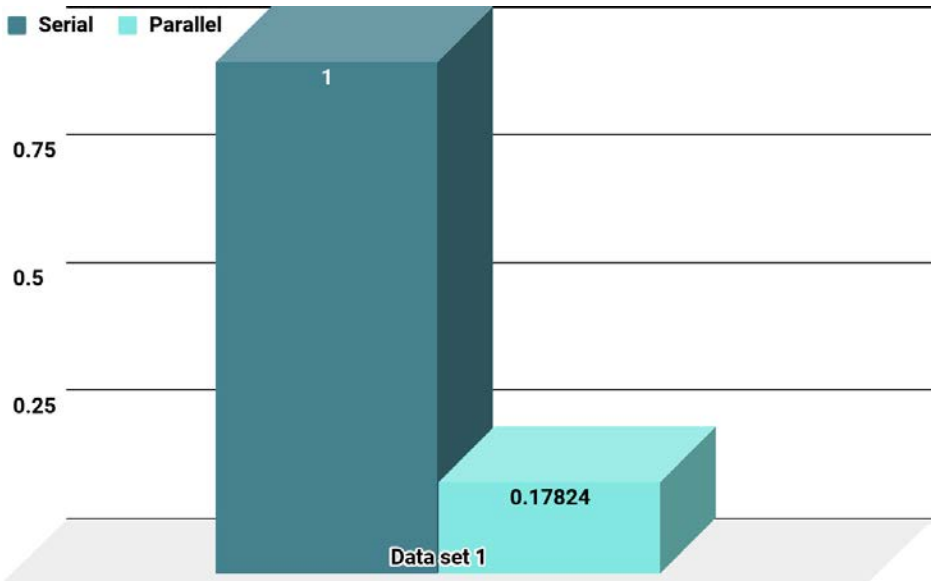


Figure 10: CPU and GPU execution time for dataset 1

Here in Figure 10 we can see that our implementation was only 5 times faster, reason being accessing 8 elements of a single array is not a CPU intensive task hence we did not achieve much benefit of running it in parallel.

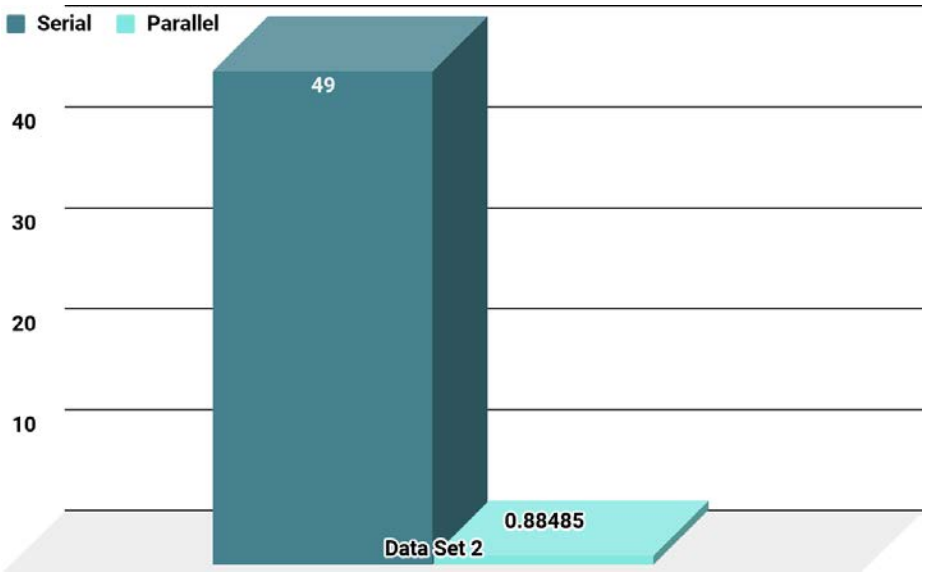


Figure 11: CPU and GPU execution time for dataset 2

In Figure 11 for Dataset 2 of 10^7 elements, we see a much better increase. Since accessing 10^7 elements by a single cpu thread takes quite some time as it access each element one by one hence

going through the loop 10^7 times. Here parallelism really works since all of those elements are each being accessed by thousands of threads separately and so an increase of around 55 times was achieved.

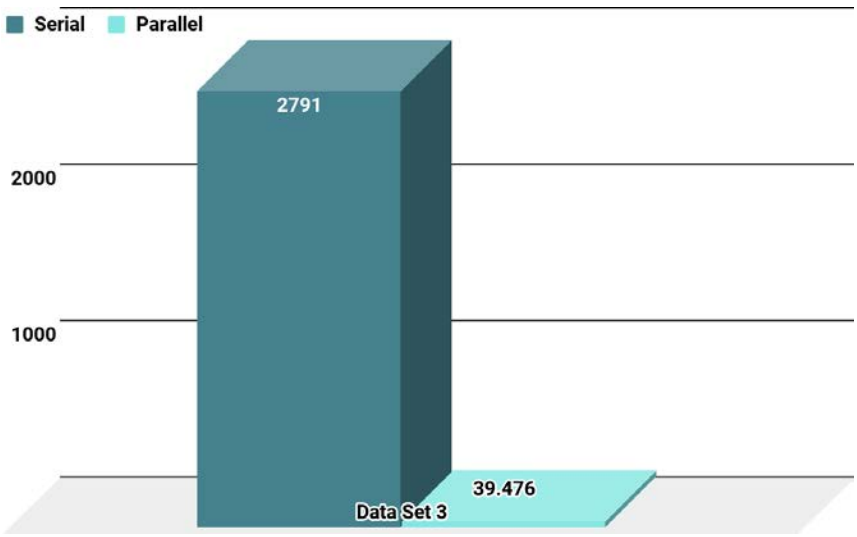


Figure 12: CPU and GPU execution time for dataset 3

As seen in the charts in Figure 10 and 11, the larger our dataset, more intense processing needed for the CPU but a lot easier for the GPU due to its multiprocessor each having thousands of threads running in parallel. So in Figure 12 , using dataset 3 where there are 4 * 10^8 elements approximately, we see even a better increase in performance that dataset 2 which is around 70 times.

We were quite happy seeing the results as we achieved significant increase in performance reducing the execution time. But we wanted to further optimize our parallel algorithm and so we did that by the use of these two techniques.

1) Reduced kernel overhead

2) Shared memory

4.3.1 Reduced kernel overhead

Using kernel overhead reduction technique on our already implemented parallel program we were able to achieve these results shown below in Table 3:

(Times faster was calculated by dividing the serial time result by the parallel time result).

(Test results are in milliseconds.)

Table 3: Execution speedup using reduced kernel overhead technique

Dataset	Original results (ms)	After optimization (ms)	Times faster
First set	0.17824	0.16840	1.0584
Second set	0.88485	0.08642	10.2389
Third set	39.476	2.8866	13

A bar chart illustration of the table above is given below in Figure 13, Figure 14 and Figure 15.

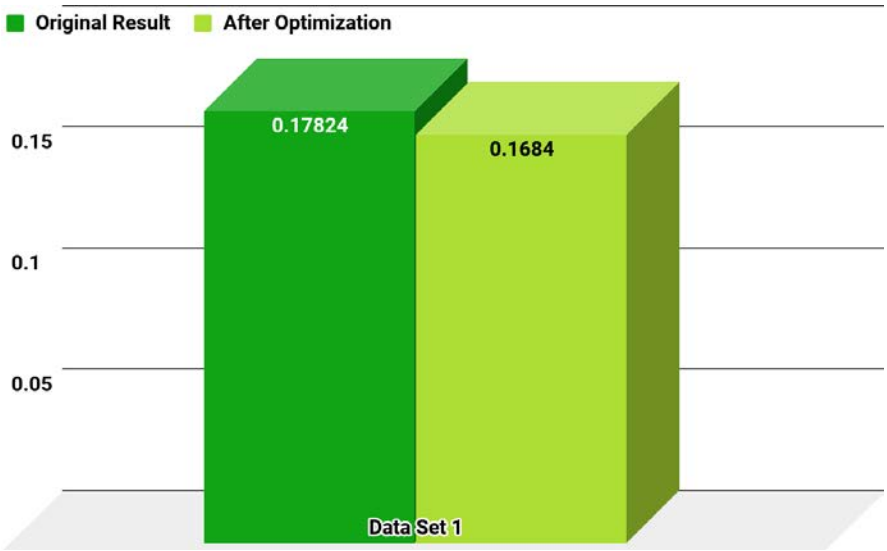


Figure 13: Execution speedup using reduced kernel overhead for dataset 1

As seen previously, smaller dataset leads to very low amounts of speedup .So even after optimizing it with kernel overhead , our speedup was only about 1.05 times faster in Figure 13.

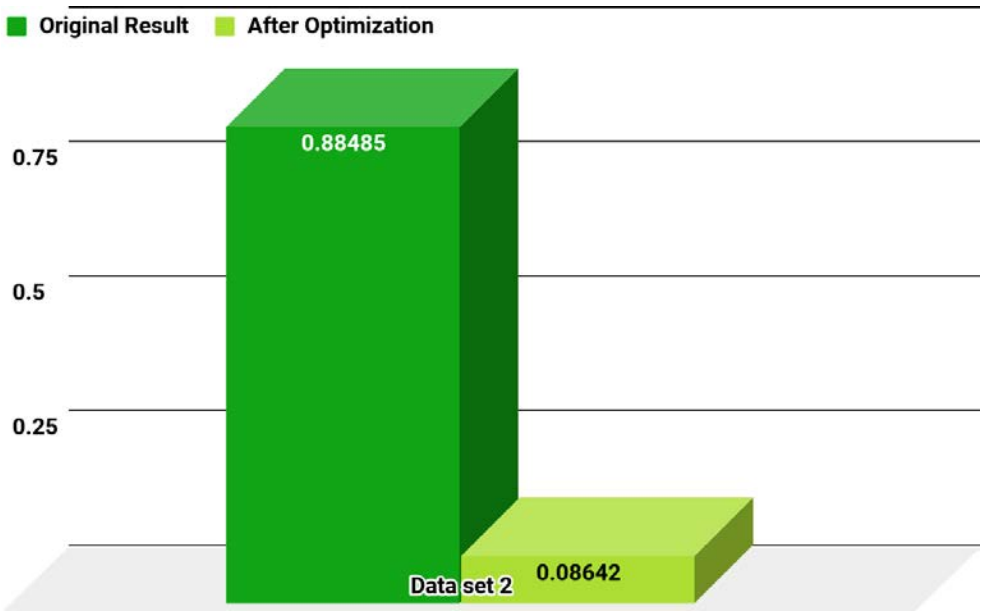


Figure 14: Execution speedup using reduced kernel overhead technique for dataset 2.

In Figure 14 , it seems using overhead did optimize the code further, reason being calling each kernel over and over again is a costly operation , hence after merging all 3 of our kernels and using dataset 2 our parallel program was faster by 10 times.

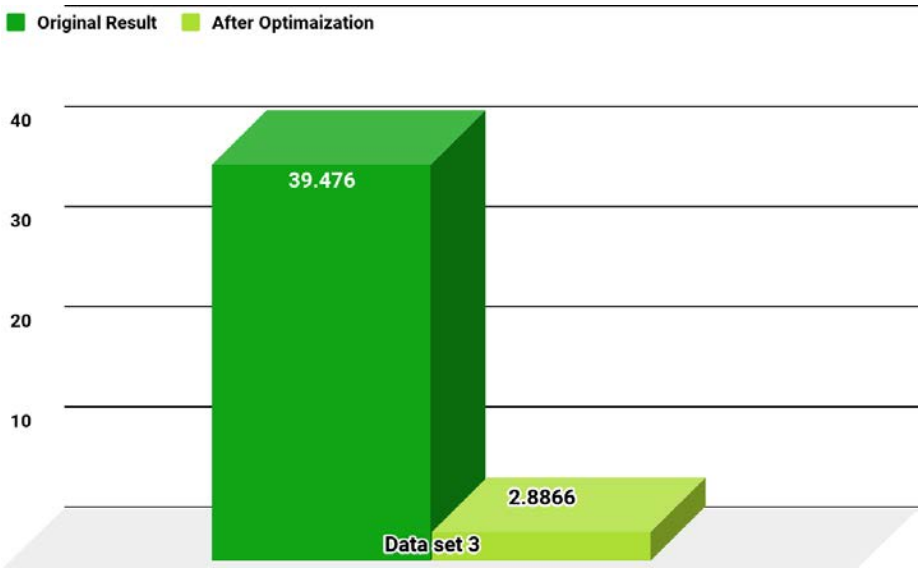


Figure 15: Execution speedup using reduced kernel overhead technique for dataset 3

In Figure 15, we can see it being around 13 times faster which is a bit faster than results obtained in Figure 14, which is about right since all overhead does is merge kernels lowering kernel calls

and that's how it saves time, not by deploying extra threads. Hence the speedup is not anything like 50 or 60 times faster rather its a few times faster.

Therefore performance increase using just Kernel overhead reduction and normal parallel optimization is shown in Table 4.

Table 4: Combined performance increase using Kernel overhead reduction and normal parallel optimization.

Dataset	Times faster
First set	5.6104
Second set	566
Third set	966.88

A remarkable increase in speedup is observed in Table 4 after implementation of both normal parallel optimization and Kernel overhead reduction.

4.3.2 Shared Memory

For shared memory we used the first data set of 8 and a new data set of 10,000 since our GPU did not have enough memory to allocate more than that.Shared memory was implemented after implementation of both normal parallel optimization and Kernel overhead reduction.

Shown in Table 5.

Table 5: Shared memory implementation

	First Dataset	Set of 10,000
Not shared	0.168400	0.011776
Shared	0.006112	0.001312
Times Faster	16.866987	8.9560

A bar chart illustration of the table above is given below in Figure 16 and Figure 17.

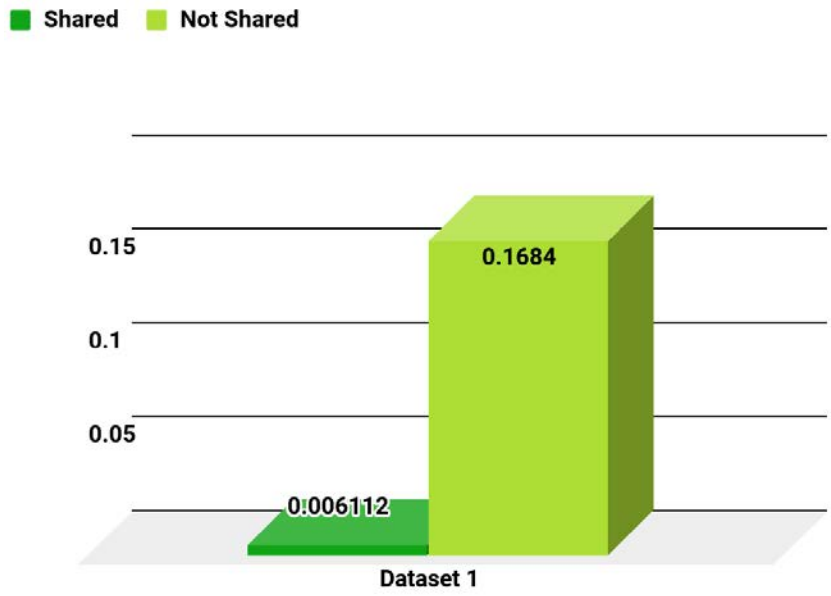


Figure 16: Speedup comparison of shared memory for dataset 1

As seen in Figure 16 , using shared memory with the First dataset gave us quite a big increase in performance. The reason for that is global memory is a lot slower than shared memory and loading an array of only 8 elements into shared memory is not an intensive task for the GPU.

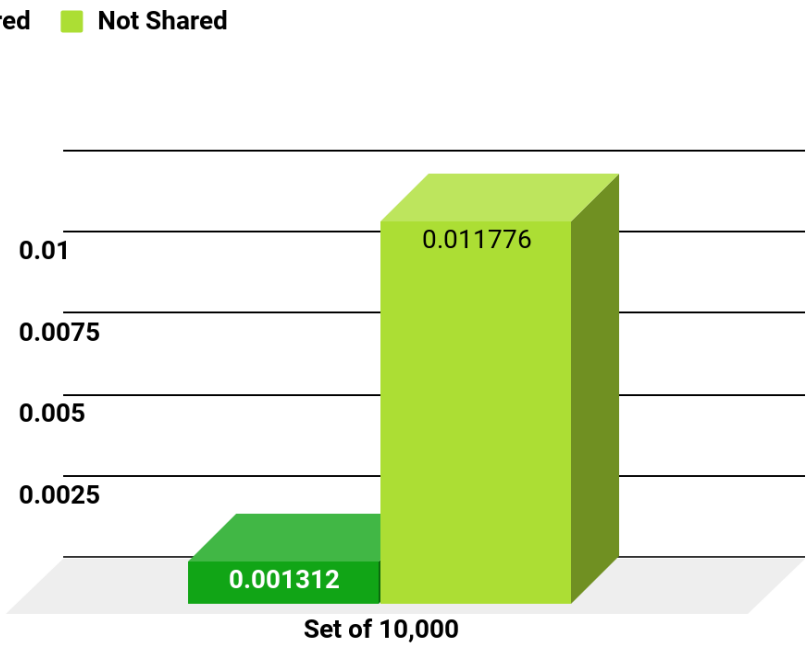


Figure 17: Speedup comparison of shared memory for set of 10,000

In Figure 17 we can see that for a dataset of around 10^4 elements it's only 8.9 times faster, much lower than the smaller dataset of 8 elements. This is due to loading an array of 10^4 into shared memory takes a lot more time than loading an array of 8 elements only.

The GPU used here is only capable of storing dataset of 10^4 elements in its shared memory, but if we had a GPU like Titan X, a much more powerful device than the NVIDIA GeForce GTX 1050TI, we could have included our original data sets 2 and 3. For a flagship GPU like Titax X, our implementation of shared memory on data set 1 would have been at least twice that of our card, so it would be 32 times faster, and our second dataset would be at least 16 times faster. Keeping this fact in mind we can say that our data sets 2 and 3 will atleast get a performance boost of 8.9 times.

Therefore in conclusion , after taking the serial program and converting it to parallel and then further optimizing it with the two techniques , Kernel reduced overhead and shared memory, our final performance gain for all three data sets are shown in Table 6 below :

Table 6: Final results for execution speedup

First set	94.6305
Second set	5082.68
Third set	8605.23

4.4 CPU to GPU Transfer Time

Now, we wanted to analyze our CPU to GPU transfer operations which are costly and then optimizing them by using

1) Reduced kernel overhead

2) Pinned memory optimization.

4.4.1 Reduced Kernel Overhead

Using kernel overhead reduction on data set 3, the results achieved are shown in Table 7.

Table 7: Data Transfer from CPU to GPU using reduced kernel overhead technique

	Normal	Reduced Overhead	Times faster
Data Transfer From CPU to GPU	327.76 milliseconds	260.77 milliseconds	1.25689

A bar chart illustration of the table above is given below in Figure 18.

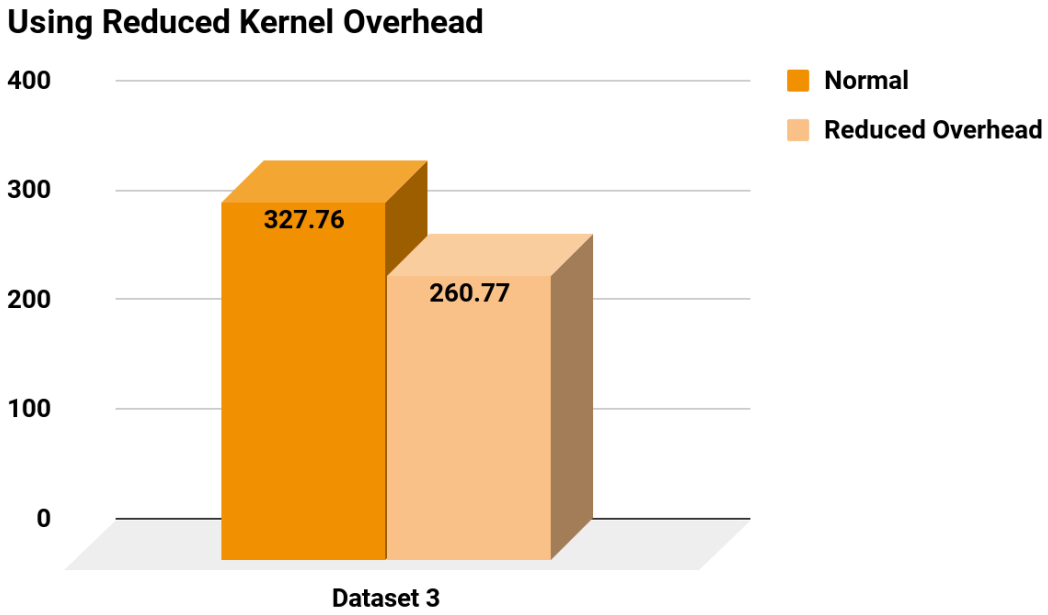


Figure 18: Bar chart of data transfer from CPU to GPU using reduced kernel overhead technique

In Figure 18, we can see that we achieved a boost of about 1.25 times. This is because when our 3 kernels merged to only one , our CPU to GPU copy (cudamemcpy()) only got called once whereas for 3 kernels it had to be called 3 times. It was not such a big increase since amount of elements to be copied did not change, just the kernel calling cost of cudamemcpy() had decreased.

4.4.2 Pinned Memory Optimization

Using pinned memory on set three we achieved

Table 8: Data Transfer from CPU to GPU using pinned memory optimization technique.

	non-pinned	pinned	Times faster
Data Transfer From CPU to GPU	304.54milliseconds	127.93milliseconds	2.38052

A bar chart illustration of the table above is given below in Figure 19.

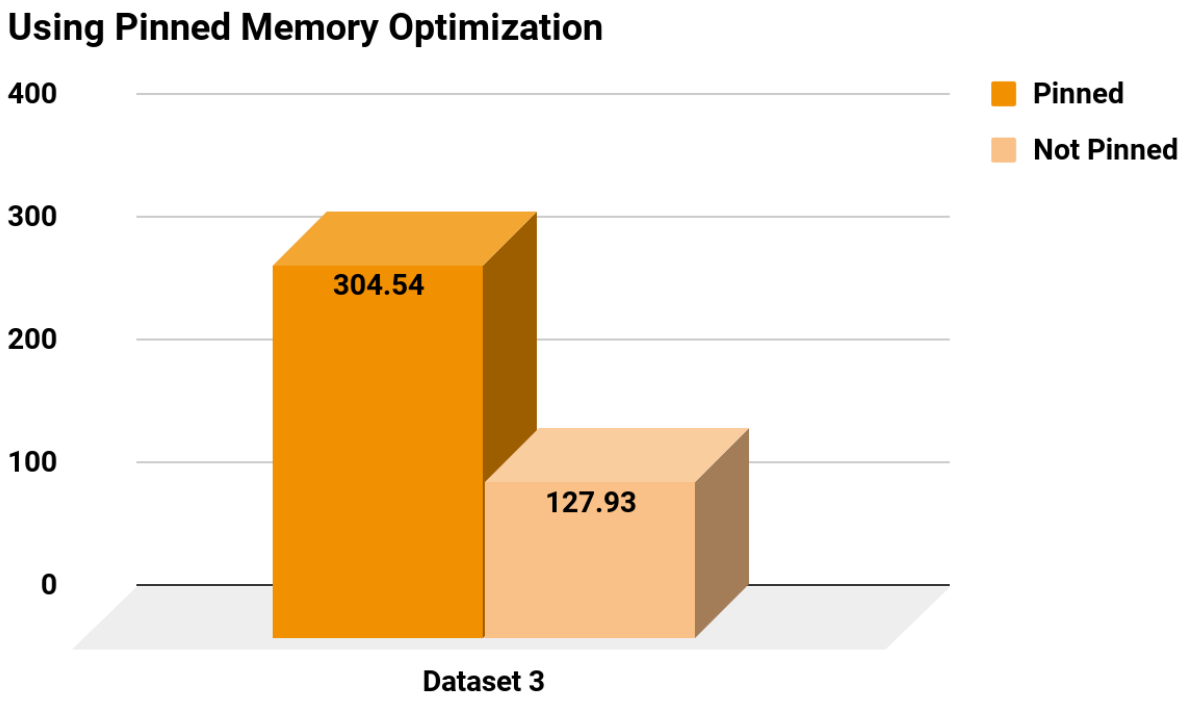


Figure 19: Data transfer from CPU to GPU using pinned memory optimization technique

As seen in Figure 19, pinned memory helped us decrease copy time by about 2.38 times. It's not a big increase since the same amounts of copy is still being done. Only difference is the copy time of elements from pageable memory to pinned memory which is no longer in the equation.

So, for CPU to GPU data transfer by using both kernel overhead reduction and pinned memory optimization the total performance gained is shown in Table 9.

Table 9: Final results for CPU to GPU transfer time

Times Faster	Seconds saved in milliseconds
2.99205	236.61

Chapter 5

Conclusion and Future Works

5.1 Conclusion

In this paper we have shown how a serial implementation of Run Length Encoding algorithm can be made much more efficient through reducing the execution time by using parallel implementation and then using various optimization techniques on the parallel implementation.

We have seen as the data set gets larger our parallel program shows more increase in performance. Hence concluding the fact that more the data, faster the parallel program will be compared to the serial one. Several datasets were used to show and compare the execution time. A gradual increment of speedup is observed with the increasing number of data. Our main focus was to execute a parallel implementation of a serial code and observe/compare its execution time and how using optimization techniques affects the overall performance of an algorithm. We were quite happy seeing the results as we achieved significant increase in performance.

5.2 Future Work

Run Length Encoding algorithm is a lossless data compression algorithm which comes in handy in any online media transfer such as in social media sites which needs compression of its data to lower its network bandwidth consumption. In order to be more efficient in bandwidth consumption, the parallel implementation with the optimization techniques used will play a crucial role. In future this knowledge could help us a lot due to growing demand for faster and reliable transfer of media data.

References

[1] X. Tian, R. Xu, Y. Yan, S. Chandrasekaran, D. Eachempati, and B. Chapman, "Compiler transformation of nested loops for general purpose GPUs," Concurrency and Computation: Practice and Experience, vol. 28, no. 2, pp. 537–556, Aug. 2015.

[2] K. Barkalov and V. Gergel, "Parallel global optimization on GPU," Journal of Global Optimization, vol. 66, no. 1, pp. 3–20, Feb. 2016.

[3] Wiggins, N. Schultz, and P. Schmidt, "Nvidia's gpgpu architecture: Fermi and cuda," Oregon State University, Tech. Rep., 2010.

[4] NVIDIA, "Nvidia launches the world's first graphics processing unit: Geforce 256," August 1999.

[5] NVIDIA (2016). NVIDIA Tesla P100. Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[6] NVIDIA (2009). NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Retrieved from https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[7] NVIDIA (2012). NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Retrieved from https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[8] Smith, Rayan (2014, September 18). The NVIDIA GeForce GTX 980: Review: Maxwell Mark 2. Retrieved from

https://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/2

[9] Whitepaper, "NVIDIA TESLA P100," 2006. Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

.

[10] L. Nyland and S. Jones, Inside Kepler, NVIDIA Corporation.

[11] Implementing Run-length encoding in CUDA. (n.d.). Retrieved from https://erkaman.github.io/posts/cuda_rle.html.

[12] Fileformat.info. Run-Length Encoding (RLE). [online] Available at: http://www.fileformat.info/mirror/egff/ch09_03.htm.

[13] How to Optimize Data Transfers in CUDA C/C. (2017, June 20). Retrieved from https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[14] RLE compression. (n.d.). Retrieved from https://www.prepressure.com/library/compression-algorithm/rle

[15] 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.

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

[17] "NVIDIA - Kernel,". [Online]. Available: https://docs.nvidia.com/cuda/cuda-cprogramming-guide/#kernels. Accessed: Nov. 19, 2016.

[18]. Optimization Techniques. Retrieved from http://www.cs.virginia.edu/~mwb7w/cuda_support/optimization_techniques.html.

[19] Reduced kernel overhead.Retrieved from http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html.

[20] Using Shared Memory in CUDA C/C. (2017, August 17). Retrieved from https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

[21] Retrieved from https://devtalk.nvidia.com/default/topic/463301/cuda-programming-and-performance/using-async-memcopy-without-using-cudamallochost-cudahostalloc-/

[22] How to Optimize Data Transfers in CUDA C/C. (2017, June 20). Retrieved from https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[23] J. Ghorpade, "GPGPU processing in CUDA architecture," Advanced Computing: An International Journal, vol. 3, no. 1, pp. 105–120, Jan. 2012.

[24] Using Shared Memory in CUDA C/C. (2017, August 17). Retrieved from https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

[25] C. Cooper(2011), Efficient shared memory use, GPU computing with CUDA. Retrieved from https://www.bu.edu/pasi/files/2011/07/Lecture31.pdf.

[26] Nambiar, P. P., V., S., & Sowbarnika, V. (1970, January 01). GPU Acceleration Using CUDA Framework. Retrieved from http://www.rroij.com/open-access/gpu-acceleration-using-cuda-framework.php?aid=51399.

[27] Mujtaba, H. (2017, December 09). First NVIDIA Titan V Volta Graphics Card Gaming Benchmark Revealed. Retrieved from https://wccftech.com/nvidia-titan-v-volta-gaming-benchmarks/

[28] Stoimen's web log. (n.d.). Retrieved December 21, 2017. Retrieved from http://www.stoimen.com/blog/2012/01/09/computer-algorithms-data-compression-with-run-length-encoding/

[29] Whitepaper, “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi” 2009. Retrieved from https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[30] The World's First GPU, GeForce 256. Retrieved from http://www.nvidia.com/page/geforce256.html

[31] CUDA Zone. Retrieved from https://developer.nvidia.com/cuda-zone

[32] An Easy Introduction to CUDA C and C. (2017, August 17). Retrieved from https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/