Exploring Parallel Merging In GPU Based System

By

Tawsif F. Rahman
ID: 2011-3-60-005
&
Md. Rakib Bahadur
ID: 2011-3-60-025

Supervised By

Dr. Md. Shamim Akhter
Assistant Professor
Department of Computer Science and Engineering
East West University

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelors of Science in Computer Science and Engineering

to the

Department of Computer Science and Engineering
East West University
Dhaka, Bangladesh

Abstract

We present a program that implemented to execute Adaptive merge sort algorithm in parallel on a GPU based system. Parallel implementation is used to get better performance than serial implementation in runtime perspective. Parallel implementation executes independent executable operation in parallel using large number of cores in GPU based system. Results from a parallel implementation of the algorithm is given and compared with its serial implementation on runtime basis. The parallel version is implemented with CUDA platform in a system based on NVIDIA GPU (GTX 650).

Declaration

We hereby declare that, this report was done under CSE497 and has not been submitted elsewhere for requirement of any degree or diploma or for any purpose except for publication

Md. Rakib Bahadur
ID: 2012-3-60-025
Department of Computer Science and Engineering
East West University

Tawsif F. Rahman
ID: 2012-3-60-005
Department of Computer Science and Engineering
East West University

Letter of Acceptance

I here declare that this thesis is from the student's own work and best effort of mine, and all other sources of information used have been acknowledged. This thesis has been submitted with our approval.

_____

Dr. Md. Shamim Akhter **Supervisor**
Assistant Professor
Department of Computer Science and Engineering
East West University

_____

Dr. Mozammel Huq Azad Khan **Chairperson**
Chairperson and Professor
Department of Computer Science and Engineering
East West University

Acknowledgement

We would like to thank and pay our sincere gratitude to our mentor and thesis coordinator Dr. Md. Shamim Akhter for supporting us throughout our endeavor with his extensive knowledge of the related grounds, his tremendous efforts and robustness that compelled us succeeding endure the ordeals and his extraordinary visions that enthralled us to refine our work to the finest point .

We also would like to thank our fellow acquaintance Tarun Das for providing us with a much clearer idea of the requirements that we are expected to achieve.

We thank our beloved family and friends, for supporting us throughout the time of our experiments that allowed us to be able to come up with what we really have today.

List of Figures:

LIST OF TABLES

Chapter 1

Introduction

1.1 Overview

Parallel computation over the GPU is trending now. Compute and process a huge data set in minimal time is always an attraction to a programmer. Reducing time was always one of the major entities in performance improvement. In this Paper we work with an existed algorithm named A**daptive merge sort** and investigate its behavior on GPU parallelism and compare the time consumption to get to a decision about GPU's infrastructure behavior to this algorithm.

Adaptive merge sort is a brand new approach to sort data where the data are first partitioned according to their initial status of either being descending or ascending . Once this partitions of ascended and descended nodes have been organized in a very fragmented manner the algorithm then advances to merge two of consecutive nodes at a time in sorted order . The process prolongs for (height -1) iterations to merge the complete set of data.

Since the algorithm was designed to work only using the resource of CPU , we have decided to implement that same algorithm using the capability of parallel computation of GPU. For such implementation the algorithm required us to be able to alter keratin portions of the code without changing any outcome. The alterations includes –

- Checking every nodes after fragmentation whether are they in ascending or descending order. If any of the nodes prevails anything otherwise than ascending those must be reverted
- In parallel implementations as it is crucial for a randomly picked data to be traced back to its origin a simple frequency distribution has been implied to find the node where the data would most probably be residing. it also serves the purpose of comparing between two consecutive node
- Lessening the number of instruction in the merging function

1.2 Scope and limitation

Scope:

The parallel use of such algorithm might result in-

- Faster calculation of enormous size of data rather than serial implementations,
- lesser memory consumptions,
- rendition of previously introduced algorithms which lacks in the fields of requirements to be accepted as new standard.

Limitations:

- Memory must me calculated beforehand the parallel algorithm takes place in design board
- Memory transfer time can be huge overhead to the total execution time
- Variable handlers must be declared with precise accuracy to suffice the complete execution
- Data that needs to merged might sometimes require to be declared as multiples of 1024
- Number of instructions within one executable block should be as much decreased as possible compared to the serial implementation due to limiting *warpsize*

1.3 Objective:

- To investigate the complete workflow of GPU architecture (Case study 1 & 2)
- Relating the parallel behavior to our redesigned algorithm (Case study 3)
- Saving consumptions of time
- Drawing a standard design so that the oversized data might not result in retaliate in long run by adding overhead to the total execution time

1.4 Preface

In next chapter there are described a concept of GPU architecture and a CUDA introduction along with its hardware co-relation. Chapter 3 contains methodology of our work and Chapter 4 has the case studies and observation we did on **matrix add, matrix multiplication and parallel application of Adaptive merge sort**

Chapter 2

Background Study

Using GPU to compute general computational work is known as GPGPU (General purpose GPU). GPU has hundreds of cores that can execute multiple numbers of instructions in parallel and it is far greater than modern CPUs with 4 or 8 cores. GPU Processing Capacity limited to independent fragments. But these fragments can process in those cores in parallel. So, a programmer should choose those parts of a program that can be fragmented and those fragments are independent from each other. So, it can be said that, GPU can process a multiple times of a same operation on many records in a stream in parallel. A set of records needs similar computation is the Stream. We can say that, the parallelism is provided by Streams.

2.1 GPU based system architecture

The GPU multiprocessors are worked as co-processors for CPU. It more likes an acceleration device for CPU. When CPU invokes a kernel to GPU that kernel executes in parallel number of times in GPUs cores. So, how many tasks a GPU can complete at a time depends on its number of SM and cores per SM. Simple adding more SM can make a device completed more task.

Total no. of SM and SP per SM depends on different architecture and model of device.

The GPU architecture can be described using 3 key words [1]

- Memory(register, shared global)
- Multiprocessors (SM)
- Stream processors (SP) or cores

2.1.1 What is Streaming Multiprocessor (SM)?

Streaming Multiprocessors(SM) are accumulation of multiple independent operators known as core/streaming processor. Upon receiving an execution command from CPU, the GPU SMs are awaken and distributed with equalized workload of “responsibilities” which are so referred as “kernel”.

Having two SMs on the delineated GPU , it is capable of executing both SMs at the same time that is where the parallelism kicks in ; meaning each workload of two different SMs ends at the same given moment ensued by very same initiating moment .

As each kernel is primarily composed of BLOCKS and THREADS, SMs are designed to execute the BLOCKS. Given (i.e.) out of two executable blocks each BLOCKs were accommodated by each SM starts execution at time=0; they are bound to terminate the both operations at the same latter time of time=3.

Though the bigger picture can be convenient enough considering SMs to be responsible for parallelism in GPU it would not be possible without the contribution of streaming processors.

2.1.2 What is a Streaming Processor / CUDA core?

Throughout the course of development of NVIDIA GPUs capacity enhancement marketing policy took a subtle turn of coining a term "core" , that has previously been always implied only to specify CPU configuration .

A CUDA core is actually a rendition of streaming processor, which aside from assisting the parallelism it actually fuels the core mechanism of the desired process (requirement to execute Kernel). Each SMs are allotted with equal number of streaming processors (SPs). The role of SPs can sometimes be undermined by exploiting nature of SMs but it is the SPs that and only that enables the SMs to be exploited with such a high scale.

SPs are capable of handling threads only. If a kernel is thrown into SMs, the SMs would distribute all the instructions residing in the kernel to all available SPs. As the GPU is designated to execute the same kernel depending on the number of iterations that must be executed to suffice the requirement demanded by user , SMs would follow the same rule of distribution , meaning that same kernel would be distributed as threads recurrently to all available SPs (lesser if requirement is set to a lower level)

Considering the kernel would require the SMs to execute the same operations residing kernel for 20 times and the SMs are sufficed with 192 SPs, each set of operations (THREADS) would be distributed to each available SPs.

In short it could be expounded as each SP must execute at least one and more threads in parallel.

Memory will be discussed in detail on next section.

2.1.3 System configuration

Device Specifications:
Processor: Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz (4 CPUs), 3.2GHz
Memory: 8192 MB RAM
Page File: 6004 MB
GPU: NVIDIA GeForce GTX 650

Execution environment:
OS: Windows 10 Pro 64-bit
Compiler: Visual Studio 2013 integrated with CUDA 7.5

GPU build specification:

```
Device: "GeForce GTX 650
Architecture: Kepler
Type: GK 107

GPU Clock rate: 1124 Mhz
(SMX)Streaming Multiprocessor: 2
(Core)Streaming Processor / (Core) Streaming processor per Multiprocessor: 384 / 192
L2 Cache Size: 262144 bytes
Total amount of shared memory per block: 49152 bytes
Maximum number of threads per multiprocessor: 1024
Maximum number of threads per block: 1024
Maximum allocable number of block per multiprocessor: 16
Total number of registers available per block: 65536
Warp size: 32

Total amount of global memory: 1024 MB
Memory Clock rate/Memory Clock rate per SMX: 2500 Mhz/1250 Mhz
Memory Bus Width: 128-bit
Total amount of constant memory: 65536 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes

CUDA Capability Major/Minor version number: 3.0
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
```

In our device specification we can see that, it is designed by Kepler architecture and uses GK107 chip. The Device has clock rate of 1124 MHz. No. of multiprocessors (SM) in device is 2 with 192 cores in each. Warp size of the device is 32.

2.1.4 Memory Hierarchy

We already know about multiprocessors in GPU. Under every multiprocessor there is large number of 32-bit register available. More than 8k registers space for devices with Compute capability 1.0 and 1.1. Compute Capability 1.2 to 1.3 has 16k registers space and 32k and 64k in Compute capability 2.0 or more [2].

Registers: Register memory it the fastest memory among all. Each thread will be assigned one set of register memory and it uses them for fast store and fetching of data which are used by a thread frequently.

Shared memory: Comparatively slower than registers but sharable between threads in a block. Because it in on chip, it has a higher bandwidth than global or local memory. It can be compare as a L1 cache in regular CPU. It shares a 64k memory segment per SM.

Global memory: Global memory resides on the device but off chip from the multiprocessors. Because of that access to global memory is much more costly than accessing shared memory. All threads from any SMs have access to global memory.

Local memory: Local memory is the private memory of threads. Local memory also is off chip and resides on the device. Local memory is specific for different thread. These memories are allocated for thread when kernel thinks there aren't enough registers to hold thread's all data. It is slow as global memory though called as 'local'

Constant memory: 64k constant memory off-chip from multiprocessors and is read only. The Host code writes on constant memory before kernel launch. Then kernel may read these memories. Constant memory access is cached. Each multiprocessor cached an amount of constant memory, so that repeated reading from constant memory can be fast.

2.2 CUDA programming model

2.2.1 Overview

Using GPUs massive parallelism to for supercomputing application is increasingly accepted by programmers to fast up the calculation. [9] NVIDIA provide one of such platform named CUDA (Compute Unified Device Architecture). CUDA programming model will be described in this chapter using a vector add application. Knowledge of C programming is only requirement for understand this chapter.

In this chapter CUDA programming model, Basic function of CUDA and interaction between CPU vs GPU will be discuss. This section introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C.

2.2.2 Block, Thread and Grid

'Thread is the fundamental building block of a parallel program'[1]. CUDA threads have their own program counter and registers. "Global memory"- a memory address space access by all threads and "shared memory" accessed by all threads within a block. Shared memory is more limited size. All threads in a block share the instruction stream and execute instruction in parallel. The number of threads running in parallel varies on device model. Based on our device maximum thread each SM is 1024.

Threads Running under CUDA must be grouped in a block [3]. The Shared memory can be accessed by the threads of the same block. That means for every block CUDA assign different shared memory space. Under every block each thread has a unique ID starts from 0. Blocks are assigned to SMs to execute. Threads can be positioned by one, two or three dimensional. **blockDim.x** returns the Thread numbers in the specific block in X axis and so does **blockDim.y** for Y axis and **blockDim.z** for Z axis. Limitation of Block in a single SM is 16 in our device.

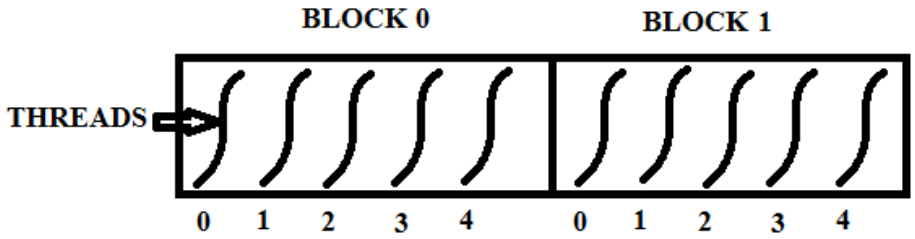

Figure 1 Block and threads

GRID is group of block. By grid we mean total block space that kernel function use to execute. Blocks in grid can also be form in a three dimensional matrix. **gridDim.x**, **blockDim.y, blockDim.z** are three built in variable to define how many blocks there will be in a grid. Let's consider a two dimensional grid, which has block of two dimensional thread.

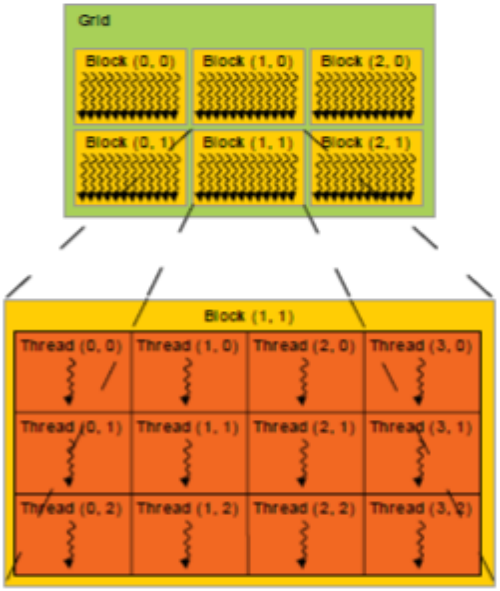

Figure 2 Grid, Blocks and threads (two dimensional) [3]

2.2.3 Kernels

CUDA C allows programmers to define C functions, called Kernel. When a Kernel function is called it execute number of time equal to number of threads specified by programmers.

Kernel function is defined by using **__global__** declaration identifier and the number of thread specified by **"<<<…>>>"***execution configuration syntax.* [1]

Programmers have to specify 3 things to work a kernel function:

1. The dimension of grid (That means how many blocks on each dimension will be used)
2. The dimension of block (This means how many threads on each dimension inside a block)
3. The kernel function

The grid dimension and block dimension have to write between the <<< and >>> syntax separated by a ',' **Comma.** That defines total threads number will be equal to the multiplication of grid dimension and block dimension (which means the kernel function will execute number of threads times in parallel)

A few things may help during write a kernel function:

1. The computation inside kernel must be Independent
2. Kernel function work with device memory

3. Every variable declared inside a kernel are completely different depending on threadID. Example: If thread 0 insert a value in variable X declared inside the kernel function, in thread 1 value of X doesn't carry the value of X in thread 0.
4. To share value between two threads in same block have to use shared memory.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figure 3 Example of a Sample CUDA C program of vector add [3]

2.2.4 Heterogeneous Programming

The CUDA programming model assumes that, the device where the threads execute exists as a coprocessor to the host running the C program. The CUDA kernel executes on the device and the rest of the C code are running on CPU [3].

The CUDA programming model maintains two separate memory space in Dram, One is Host memory and the other is Device memory by the Host and device, respectively. 'Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (described in Programming Interface). This includes device memory allocation and deallocation as well as data transfer between host and device memory.'[1]. See figure 4.

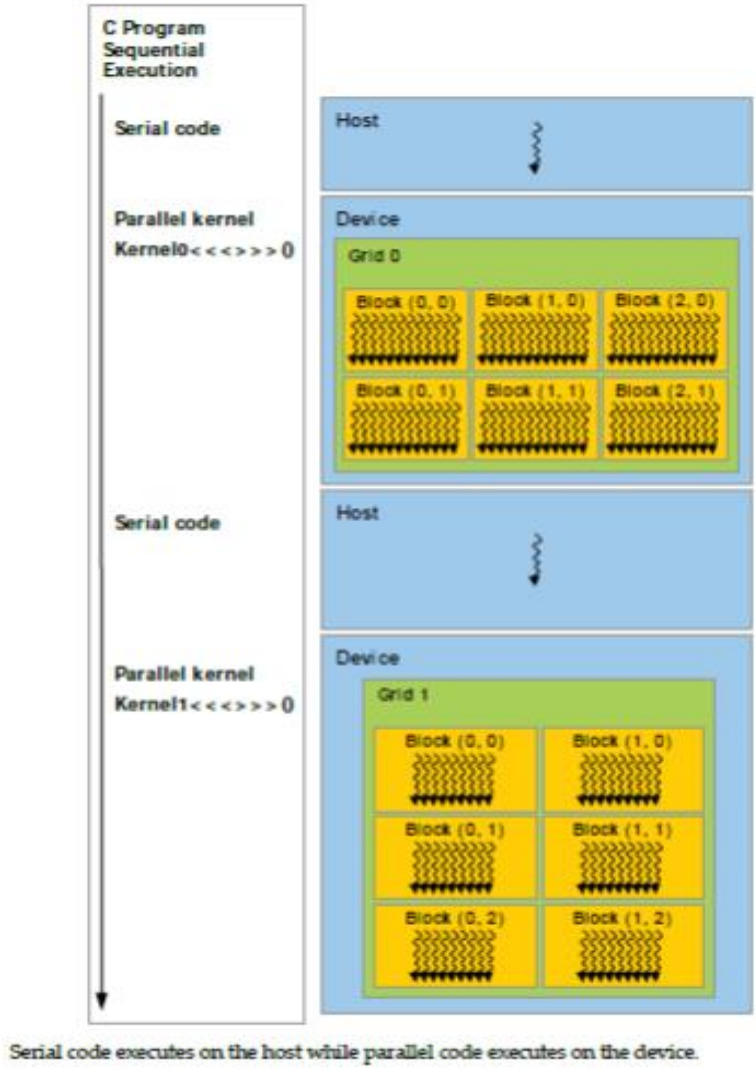

Figure 4 Heterogeneous Programming [3]

2.2.5 Some CUDA syntax and qualifier

To understand completely the next chapters we need to discuss some built in CUDA function and their use. To understand or write a simple CUDA program the code can be divided into two parts:

1. Host Code
2. Device code

Host codes are execute serially on CPU and the device functions that means kernels are execute on GPU.

- **__global__[3]:** It is a qualifier that declares a function as being a kernel. It gives a function such characteristics:
    - Functions that execute on device
    - Can be called from Host function
    - Can be called form device function in device with compute capabilities 3.0 or more

The function must have a **void** return type. A call to a __global__ is asynchronous, that means it returns before the device has completed its execution.

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Figure 5 Kernel definition [3]

- **Dim3:** This type is an integer vector type based on uint3 that is used to specify dimensions.[3] When defining a variable of type dim3, any component left unspecified is initialized to 1. There are two Dim3 type built in variable in CUDA: **gridDim** and **blockDim.**
    - **gridDim:** This variable contains the value of all three dimension of a Grid. **Dim3 gridDim(5,5)** – means the block are formed a 2 dimensional matrix with both length and height 5. Total number of block 10. Inside the braces first value is length and second one is for width. If one of these values are absent the that axis set to 0 by default.

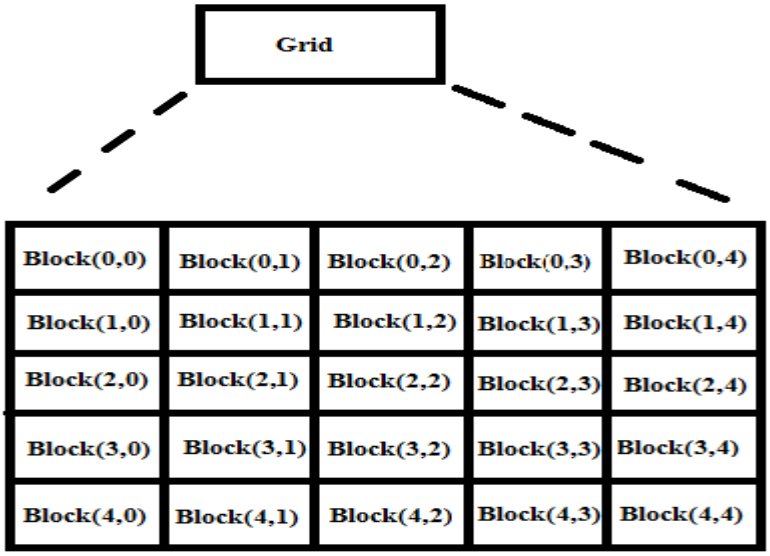

Figure 6 gridDim (5, 5)

- **blockDim:** This variable contains the value of all three dimension of a Block. **Dim3 blockDim(2,2)** – means the threads are formed a 2 dimensional matrix with both length and height 2 inside the block. Total number of threads 4.

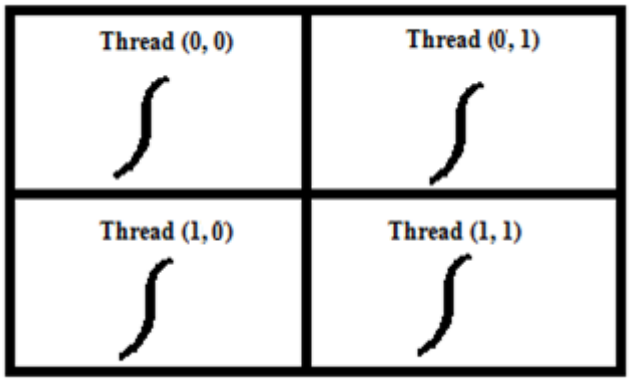

Figure 7 blockDim (2, 2)

- **cudamalloc():** cudamalloc() has been used for typically allocate linear memory. It is used for allocating array for device memory. It is written as follows

  float *d_a;
  cudamalloc(&d_a, size of array);

- **cudafree():** Use to free the memory allocated by cudamalloc(). Call as:

  cudafree(d_a);

- **cudamemcpy():** Copy the data from Device memory to Host memory or Host memory to Device memory.

  cudamemcpy(DeviceMemoryAddress, HostMemoryAddress, size of source,cudamemcpyHostToDevice)

  cudamemcpy(HostMemoryAddress, DeviceMemoryAddress, size of source,cudamemcpyDeviceToHost)

2.3 Related works

The Computational power of GPU is being discovered by the researcher in Time. Many research and new type of application are designed in GPGPU by programmers. Many complicated computational problem has been solved using GPU's computational capability and gain better time performance using GPU's parallelism.

Some outstanding woks on GPGPU are given below:

- Sanyam Mehta , Arindam Misra , Ayush Singhal , Praveen Kumar , Ankush Mittal & Kannappan Palaniappan worked on **Parallel implementation of video surveillance algorithms on GPU architecture in CUDA**. They implemented Gaussian Mixture Model (GMM), Morphological Image operations and Connected Component Labeling (CCL) In CUDA and solve the dependency problem in CCL algorithm [4].
- Yao Zhanga , John Ludd Reckerb , Robert Ulichneyc , Giordano B. Berettab , Ingeborg Tastlb , I-Jong Linb , John D. Owens **worked on A Parallel Error Diffusion Implementation on a GPU** and gain 10 – 30 x speedup over a two-threaded CPU error diffusion implementation with comparable image quality[5]
- **HETEROGENEOUS HIGHLY PARALLEL IMPLEMENTATION OF MATRIX EXPONENTIATION USING GPU** presented by Chittampally Vasanth Raja, Srinivas Balasubramanian, Prakash S Raghavendra. The research is on highly parallel implementation of Matrix Exponentiation and It employs many general GPU optimizations and architectural specific optimizations with 1000X speedup and 44 fold speedup with the naive GPU Kernel.[6]
- Brij Mohan Singh, Rahul Sharma, Ankush Mittal, Debashish Ghosh observe the accuracy and performance characteristics of GPUs on well-known global binarization Otsu's approach for Optical Character Recognition systems and implemented the algorithm in CUDA to make binarization faster for recognition of a large number of document images on GPU.[7]

GPU computing makes parallelization more powerful and makes complex computation possible. For the advantage of GPU computing it is being developed in a large number. We, can say that GPU computing open the doors of supercomputing by remove the limitation of parallel computing using CPU.

2.4 Some merging technique:

In this section we will talk about **merge sort algorithm** invented by **John Von Neumann** in 1945 and its's one modified version **Adaptive merge sort**[8] proposed by **Nenwani Kamlesh, Vanita Mane, Smita Bharne** The algorithm we implemented in our work.

2.4.1 Merge sort [8]

Let's consider the sorting is working on data set of size N. Merge sort woks as follows:

- Divide data set into sub list until get sub list of size 1
- Get N number of sub list
- Merge two nodes into new node; new node number will be exact. Repeat this step Log (N) times

2.4.2 Adaptive merge sort [8]

With data set of size N the steps are:

- Start by finding the sub-lists which are already sorted in required or reverse order

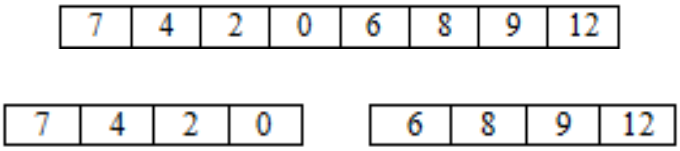

Figure 8 Adaptive merge sort step 1

- If there is any sub-list with elements in reverse order, then reverse the sub-list by exchanging 1st element with last, 2nd element with 2nd last and so on.

Figure 9 Adaptive merge sort step 2

- Keep on merging sub-lists to produce new sub-lists until there is only 1 sub-list remaining.

Figure 10 Adaptive merge sort Final step

Our parallel implemented application works on 3 steps

- Divide in sub list (node) which is already in required or reverse order. Then count the node and keep it in p
- All nodes that are sorted in reverse order will reverse. This operation will execute parallel
- Merging nodes number of times equal to Log (p).

Complexity of time and space [8]

	Required order			Random order			Reversed order		
	Merge sort	**Adaptive merge sort**	**Our implementation**	**Merge sort**	**Adaptive merge sort**	**Our implementation**	**Merge sort**	**Adaptive merge sort**	**Our implementation**
No. of merging step	**Log (n)**	**1**	**1**	**Log (n)**	**Log (n)**	**Log (p)**	**Log (n)**	**1**	**1**
Space complexity	**n**	**2n**	**3n**	**N**	**2n**	**4n**	**n**	**2n**	**3n**
Time complexity	**n log(n)**	**n**	**n**	**n log(n)**	**n log(n)**	**n log(p)**	**n log(n)**	**n**	**n**

Table 1 Time and space complexity comparison

Chapter 3

Methodology

To implement an application in parallel, have to understand the problem at first. Have to fragment the problem in independent section. Then searching those sections, if whether some of them have independent iteration in them or not so we can make those section execute in parallel.

3.1 common parallel patterns [1]

There Number of problem that we can call as parallel patterns. These patterns can be found in much application. We will discuss those in this chapter so that, it'll be easier to find those fragmented section that can be executed as parallel.

3.1.1 Loop-based patterns

We all are familiar with loops. There are different kinds of loop all with an initial and terminating condition which the get even before the loop had been started. In serial implementation loop with dependencies doesn't create any problem. Cause in serial implementation, only one iteration has performed at a time. But in parallel implementation if we The increment of loop can be relate with thread index and If the data dependencies can be handle than we can implement a kernel function instead of **for** loop. So, if we get a loop in our program we will know that could be implemented in parallel.

3.1.2 Fork – join pattern

Fork - join pattern is a huge common pattern found in serial implementation where, there will be a synchronization point and when the program comes at that point where the work can be distributed in multiple processors. The main process than "fork" or spawn in similar processes and execute in parallel. After the processes completion they join again with their processed result and complete the rest of part in serial

3.1.3 Divide and conquer

Divide and conquer is one of the most common strategy to programmers to divide a big problem and conquer those division individually. Divide and conquer are mainly used in recursive application. Merge sort is a better example for divide and conquer pattern. At first it recursively divide the data set into small units the sort them while merged.

3.2 Solving a problem –Our merge

We already know that in Adaptive merge we divide the data set into small node based on their order. It is an iterative process and every iteration depends on previous iteration. That's why we do the portioning part in serial.

In 2nd step of Adaptive merge we reverse those nodes that are in reverse order. It is also a iterative process. In every iteration only one node will be reversed. This iterations and independent so, we do the reversing in parallel by writing a kernel. Where thread 0 will check node 0's order, thread 1 checks whether node 1 are in ascending or descending.

We do the merging in parallel for the same reason we told just now. In every step of iterations only one value finds its new position in the merged array and updates the node information. Every data do the same thing and they don't have any data dependency

In next chapter we will discuss 3 CPU bound program and their parallel implementation in CUDA. The memory passing operation from host to device memory or device to host memory is very time consuming. That means a CUDA application is a I/O bound program. So, it's not a wise decision to implement an I/O bound application implementation in CUDA.

Chapter 4

The Case studies

Before getting into much of details or exposure we have decided to experiment with primitive concepts of CUDA which includes –

- Threads and blocks manipulations
- Consumption of time
- Relative time comparisons between both CPU and GPU

4.1 Case Study 1

We came up with a very basic program to be executed on our GPU system which includes addition of two variables and then returning the final results to CPU.

Objective of the experiment:

- Understanding the basic requirements of CUDA
- Optimizations of those requirements

4.1.1 Implementation of SMs and SPs:

4.1.1.1 Understanding THREADS and BLOCKS:

These being the heart of CUDA parallel programming with a predefined minimalistic approach encourage the programmers to design the kernel with inclusive of all elements required for parallelism.

A sample CUDA code

```
#define gridsize 5
#define blocksize 1024
__global__ void adder(int *d_a,int *d_b,int *d_c)
{
        int a = blockDim.x * blockIdx.x + threadIdx.x;
        d_c[a] = d_a[a] + d_b[a];
}
int main()
{
```

……
adder << <gridsize, blocksize >> >(d_a, d_b, d_c);
……..
}

The function labeled with __global__ type is function that our serial code can invoke for the GPU to execute. The instructions written inside the function are exactly the instructions each thread would execute. As the ***blocksize*** stands for the number of threads allocated in each **block,** the GPU will enumerate and create 5 **blocks** each accommodating 1024 **threads** in a logical partition of GPU.

Once the enumeration is done blocks would be thrown to SMs to execute. Now there are some restriction imposed on the device we were working on due to resource constrains. As each threads are first being allocated in a block and the blocks are then superimposed on the SMs, by predefinition the users are bound to declare the size of block equal or lesser than the number of threads the SPs confined in a single SM are capable to accommodate.

The maximum number of threads each 192 SPs within a SM can manage for our device has been confined within 1024 which means if the user tends to declare ***blocksize*** more than 1024 (i.e. 1025 , 1250 , 2048) the kernel will not simply be executed . If the user declares ***blocksize*** below 1024 or equal which the example preferred to be referenced with the GPU will let the program to run utilizing the defined numbers of **threads** within each block.

What backend execution would take place can be described as each SMs will accommodate 5 (from example) in a row within 5 SMs sitting in a row (if available. GTX 650 only has 2. More explanations forthcoming). And accommodation of block is then followed by each blocks designating its 1024 threads to 192 SPs which resembles the capacity of operating on 1024 threads at time (GTX 650 cannot operate on more than 1024 threads within each SPs)

Bottom line: multiple threads creates a block, multiple threads creates a grid which determines the size of the total number of parallel calculations to be done.

4.1.1.2 Equivalency and Relevance:

Referring from the previous explanations there must arise some question of-

- *"what if the threads in a block (blocksize) being defined tends to be smaller than the available accommodating space within your SMs" ,*
- *" Would the rest of the core/SPs be sitting idle" ,*
- *"How GPU would overcome the burden of calculating blocks within a grid when the blocks in a grid (gridsize) tends to be defined more than the number of available SMs"*

4.1.1.3 Explanations:

How gridsize and blocksize is determined:

From the example the user intended to calculate executing 5120 adding instructions in parallel. As each SM has capability of adding 1024 instructions (instructions per threads) within it gridsize X blocksize would be the determining factor for the iterations it requires to meet the requirement.
gridsize = 5 , blocksize = 1024 creates a space of 5120 threads to be calculated at first .

- *"Should they really be executed on a single stride?"*

The answer to that is **NO.** It is impossible unless our GPU allocating 5120 threads to be executed in a single fly as GTX has the privilege of executing 2048 threads at a time. (Each SM handles 1024 threads, 2 SMs available, 1024 X 2 = 2048 threads)

The GPU will throw any two of the five blocks to the 2 available SMs in the device and they will be calculated with both maintaining precisely the same initiating and ending time for those 2 executing blocks , which explains the parallelism of GPU . Once these two blocks are done with the calculation /execution a set of another two blocks are then thrown into the SMs in a serial manner .When these last two blocks are executed in parallel the remaining block (single block) is thrown to any of the SMs to be calculated.

If each of the sets takes 3.200 micro seconds to be executed (two blocks are executing at the same time)
The total execution time should be somewhere around 9.6 micro seconds. It can be decremented by decreasing the numbers of data are being written to the registers in serial.

- *"what if the threads in a block (blocksize) being defined tends to be smaller than the available accommodating space within your SMs" ,*
- *" Would the rest of the core/SPs be sitting idle" ,*
- *"How GPU would overcome the burden of calculating blocks within a grid when the blocks in a grid (gridsize) tends to be defined more than the number of available SMs"*

#define gridsize 5	#define gridsize 10	#define gridsize 20	#define gridsize 40	#define gridsize 80
#define blocksize 1024	#define blocksize 512	#define blocksize 256	#define blocksize 128	#define blocksize 64

For example if the user defines the blocksize to be 512 / 256 / 128 / 64 threads per each block should the SMs would only allocate a block of 512 / 256 / 128 / 64 threads **[cite]** to a single SM . **NO.**
[cite: gridsize must be 10, 20, 40 and 80 to calculate same number of 5120 data]

Steps of handling that situation:

1. Each SM would first allocate two 512 /256 / 128 / 64 sized blocks in them.
2. The SMs would then calculate how many threads are available within them to be available which must allocate at least one complete block
3. For each situations these SMs would be able to supply with 2 (1024/512), 4 (1024 / 256) , 8 (1024/128) and 16 (1024/64) more spaces consecutively to be allocated .
4. Given that the SMs are each supplied with one block already for those previously situations each SMs will then again be supplied with 1 , 3 , 7 and 15 blocks of predefined sizes.
5. Now consider previous 4 steps being done in parallel rather than the serial nature it has been explained by.
6. Which means for those required blocksize the SMs are filled with 4 , 8 ,16 , 32 (2 , 4 , 8 ,16 each SM) blocks per iteration and following the already explained serial scheduling of blocks after the running iteration
7. Meaning the GPU would iterate 3 times for each of the given situations and consume 9.6 microseconds just like the situation with (5 X 1024) data

4.1.1.4 Time Complexity and Maximum blocks per SM:

Nvidia has already limited 16 Blocks per SM for GTX 650. If the user defines blocksize as 32 threads per blocks the GPU should be able to inject 32 blocks per SM. Due to the shared memory constrain the SMs would not be able to accommodate more than 16 blocks.

For calculation 5120 data defined with blocsksize 32 and gridsize 160 the GPU would push 32 blocks in SMs (16 blocks per SM) accommodating 1024 threads in total (512 threads each SM) which indicates 1024 threads (512 threads) are being sit idle. Which implicates for the same size of data that we have previously worked on if the blocksize is somehow mismatched the program would execute with more iterations (11 for given example) consuming more times (35.2 microseconds)

Best Fit and Worst Fit :

Best Fit:

From our design perspective we have considered granting a set of data to be fit in the best case scenario if and only if the data fits within the block occupying exactly the same number of threads . If 1024 data is declared to be worked upon and perfectly fits within a blocksize of 1024 then this should considered as a **Best Fit** scenario.

Worst Fit:

From the exampled scenario if the data now takes an increment of only one data making it 1025 ,to calculate the data the GPU would require to be iterated for a second Kernel launch which would consume as much time as a complete Kernel iteration .

Our approximations:

Each kernel execution takes exactly the same amount of time due to thread and block level parallelism rather depending on the number of data to be executed on

4.1.2 Experiments and results:

Experiment 1: relevance and connection between blocksize and time consumption

So we have decided to increase the blocksize per sample whether the gridsize would be unchanged.

Gridsize: 1

Blocksize: 32 to 1024

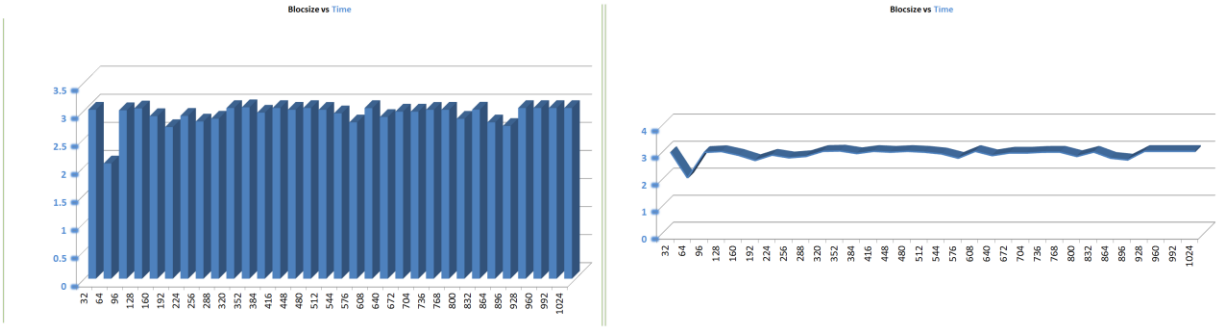

Figure 11 Time graph for Gridsize: 1 & Blocksize: 32 to 1024

Analysis:

A single block took almost the same amount of time regardless the number of data or threads in flight that had been taken in consideration. The initial downtrend of the graph might play a little role of counterproof but in average the time taken to execute each of the blocks keeps a relatively static flow.

Experiment 2: relevance and connection between gridsize and time consumption using the Best Fit and Worst Fit model

Data: 4096 data

Gridsize: 4 to 128

Blocksize: 1024 to 32

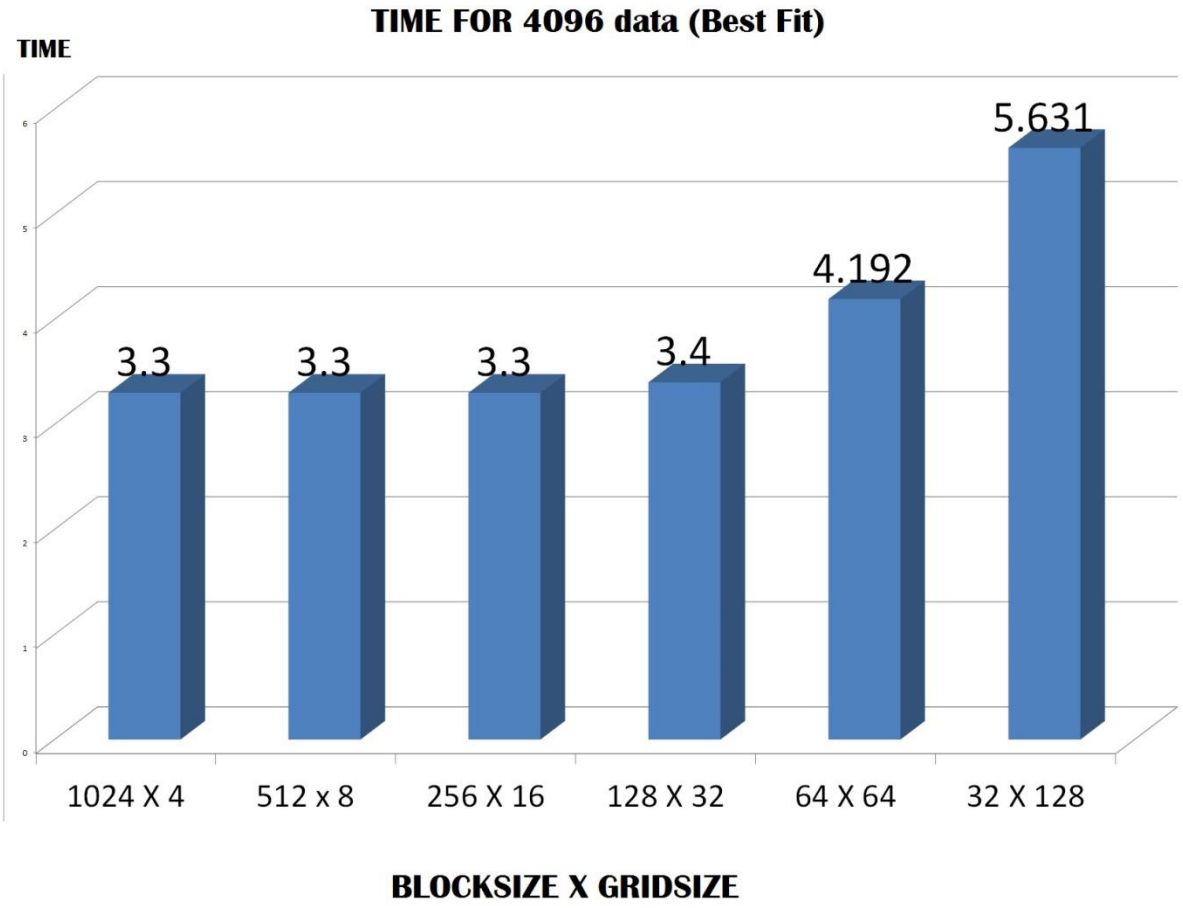

Figure 12 Time Graph for best fit

Analysis:

Regardless of the number of threads and data time remains constant as long as the Kernel executes for the same amount of time (first 4 bar) . 5th and the 6th falls in the categories of over consumed block (more than 16 blocks) that had already been explained in the previous sections.

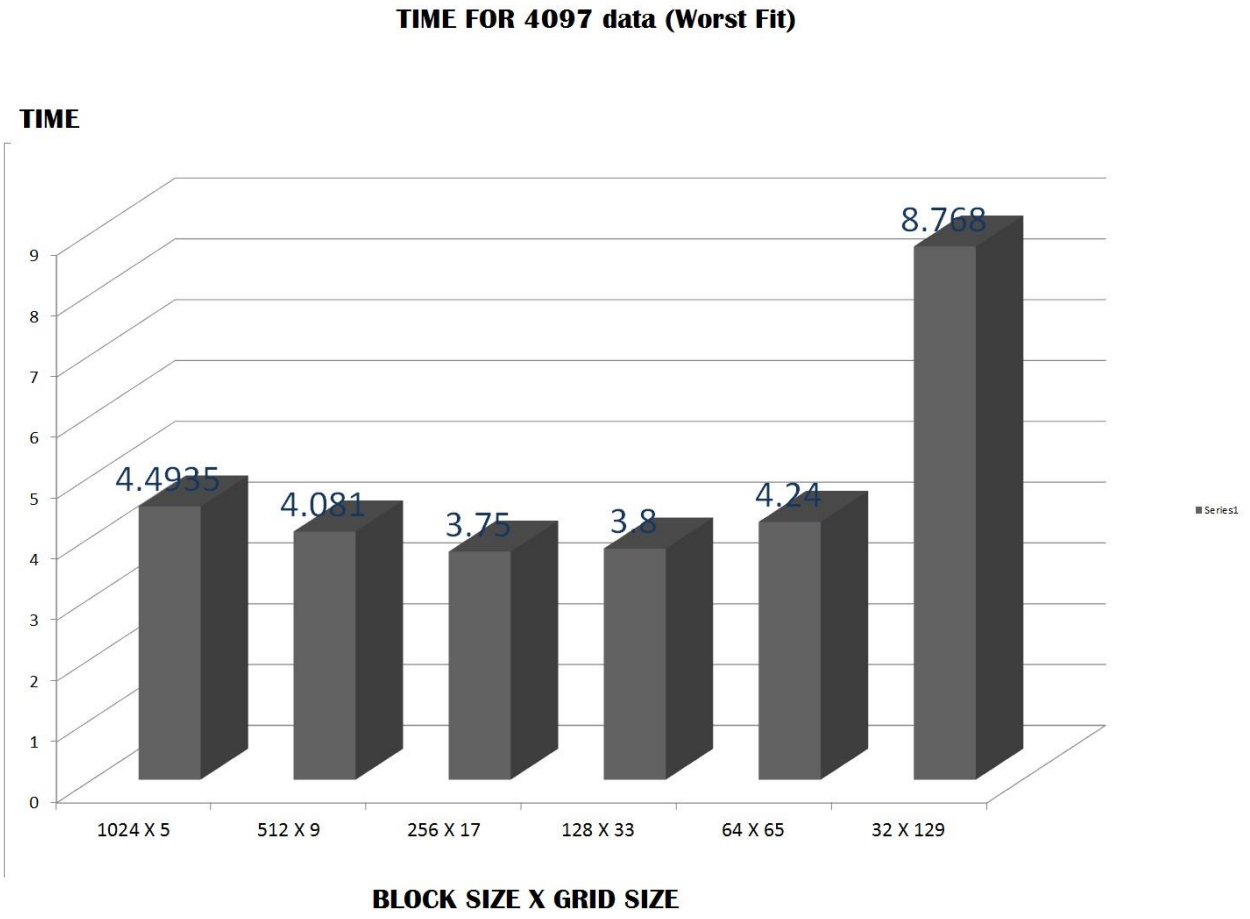

Figure 13 Time Graph for worst fit

Analysis:

To compute single more data the GPU had to iterate its Kernel for one more time adding more time as execution overhead.

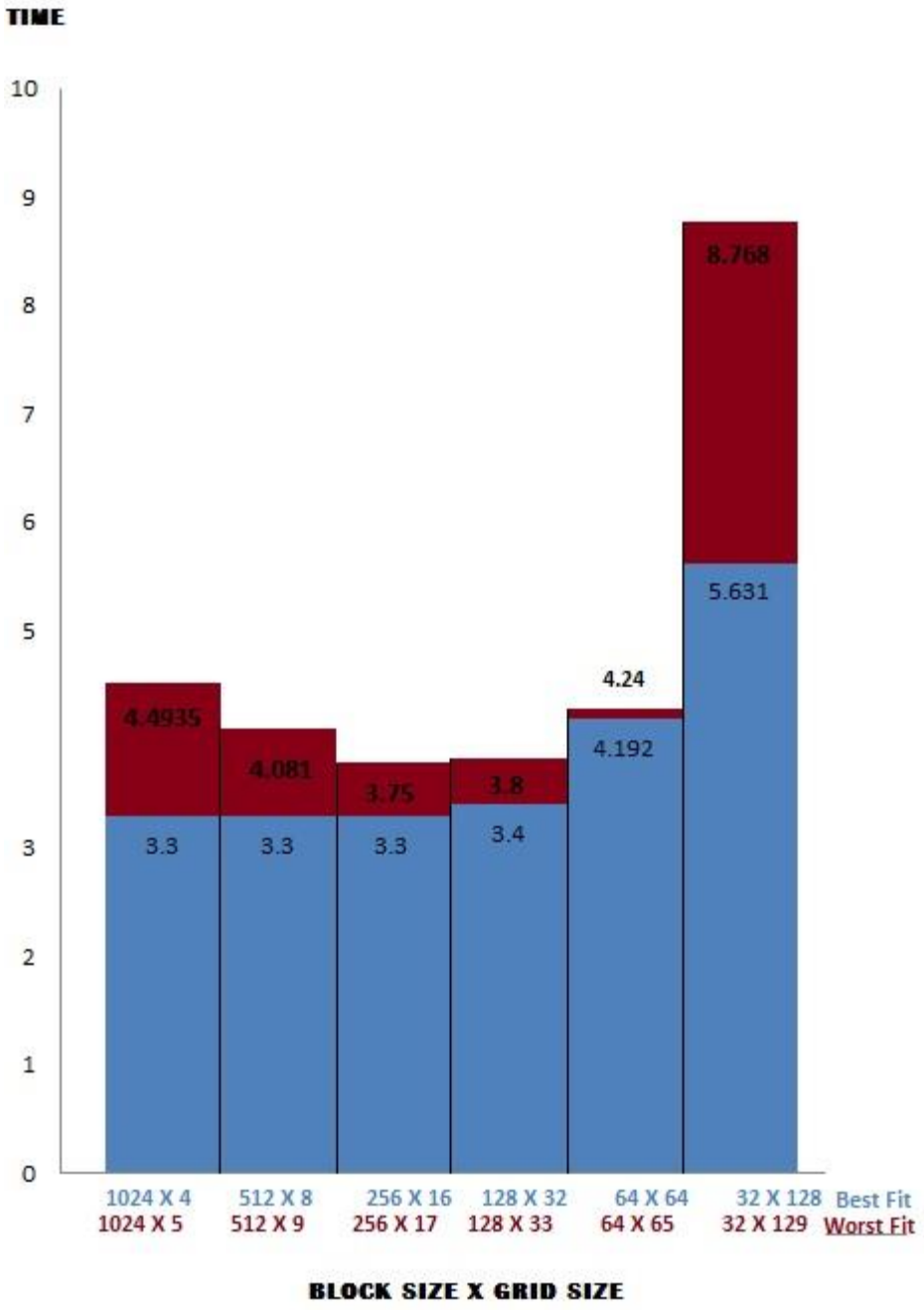

Figure 14 Combined Time Graph for best and worst fit

Findings:

Blocksize and Gridsize must be manipulated with precautions and expectations of them being self-manipulating for the given problem otherwise it can tear down the primary purpose of integration of GPU parallelism.

4.2 Case Study 2: Matrix Multiplication

Knowing CUDA A Little More

To understand CUDA a little more and compare the runtime performance we will discuss in this chapter a sample case, Matrix Multiplication. We will discuss its algorithm, serial implementation, parallel implementation and runtime comparison. We assume that the reader are familiar with matrix.

4.2.1 Matrix Multiplication

To multiply two matrices the condition that must be followed is: 'The column of 1st matrix must be equal with the row of 2nd matrix'. The dimension of the result matrix will be (column of the 1st matrix × row of the 2nd matrix). That means, if we take two matrix A[2][3] & B[3][4] and multiply then the result matrix will be C[2][4].

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \end{bmatrix} \times \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \end{bmatrix}$$

Figure 15 Matrix multiplication

The equation to calculate the value of each position of C is:

$C_{0,0} = A_{0,0} \times B_{0,0} + A_{0,1} \times B_{1,0} + A_{0,2} \times B_{2,0}$ or,

$C_{1,2} = A_{1,0} \times B_{0,2} + A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}$

4.2.2 Serial implementation

In serial application the program determines every value in result array one at a time. We can describe the procedure as follows:

```
for i = 0 to < row of 1st matrix
begin
  for j = 0 to < column of 2nd matrix
  begin
    for k = 0 to < row of 2nd matrix
    begin
      sum = sum + matrix1[i][k]*matrix2[k][j];
       end

       result_matrix[i][j] = sum;
       sum = 0;
     end
   end
```

This part calculates the value of each position of the result matrix. As we can see to calculate the result matrix it needs 24 times to calculate the value of **sum** according to the example given above**.**

It maybe seems very little number to a beginner level programmer and also took a tiny period of time to execute on the modern processor, but when we multiply matrices with hundreds of rows and columns then the time cost will be visible to us. Time consumption of matrix multiplication is mention in next section.

We present some data of time to calculate to multiply two matrices. To avoid complexity we are using square matrices (matrix with same height and width)

Size of matrices	Time (sec)
200 (A[200][200] & B[200][200])	0.037
300	0.152
400	0.408
500	0.871
600	1.555
700	2.688
800	5.469
900	9.871
1000	13.658

Table 2 Average calculation time of matrix multiplication

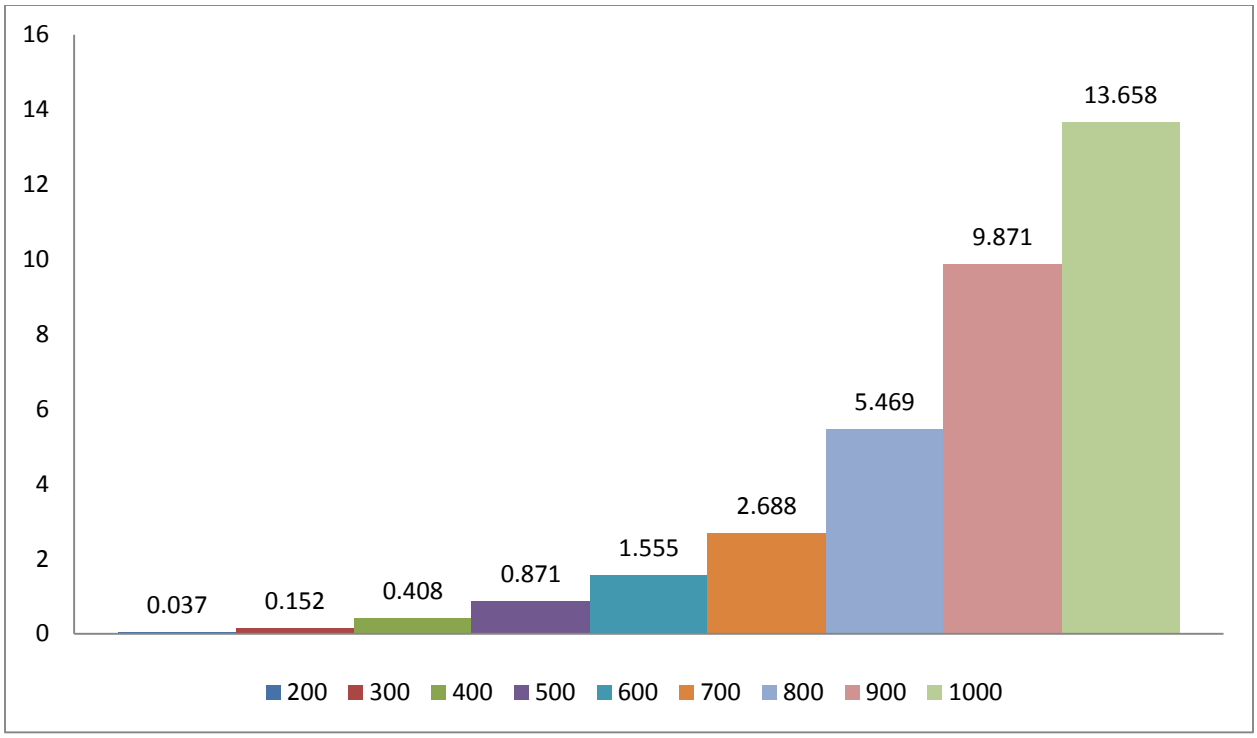

Figure 16 Calculation time (sec) vs matrix size

So, we can see a drastic change of time in such a small increment of array size. That's where the concept of parallel execution comes.

4.2.3 Parallel implementation

Before understand the parallel application let's recall the concept of two dimensional thread and block at first. We already know that every thread has an ID known as **threadIDx** start from 0 at every new block. To know position of a thread from the beginning of first block we can use the following equation:

Position_of_thread = blockIDx × blockDim + threadIDx

Here,

- blockIDx is the block ID which contain that active thread
- blockDim is the number of thread in a block
- threadIDx is ID of that active thread

So the idea is, value of every position of result matrix will be calculated in one different thread. That means, value of $C_{0,0}$ will be calculated on **threadIDx**(0,0) and $C_{1,2}$ on **thradIDx**(1,2),

Whether they belong on the same block or different. Which is completely depends on the value of **blockDim** defined by programmer.

So, the Kernel function will be like:

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
        // Each thread computes one element of C
        // by accumulating results into Cvalue
        float Cvalue = 0.0;
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        if(row > A.height || col > B.width) return;
        for (int e = 0; e < A.width; ++e)
                Cvalue += A[row][col+e] * B[row+e] [ col];
        C.elements[row][col] = Cvalue;
}
```

After compiling matrix multiplication program in CUDA based on different Block Size (number of thread in a block) the outputs are (all measurement of time are in **µs**)

Matrix Size	block 32	block 16	block 8	block 4
200	3080.054	2746.244	3742.556	12178.075
400	45405.638	42524.052	67428.965	232724.521
600	75183.699	84427.212	105658.493	362601.307
800	157396.11	160046.887	249656.568	858890.919
1000	327179.877	318410.165	486874.411	1677327.788

Table 3 Matrix multiplication – kernel execution time

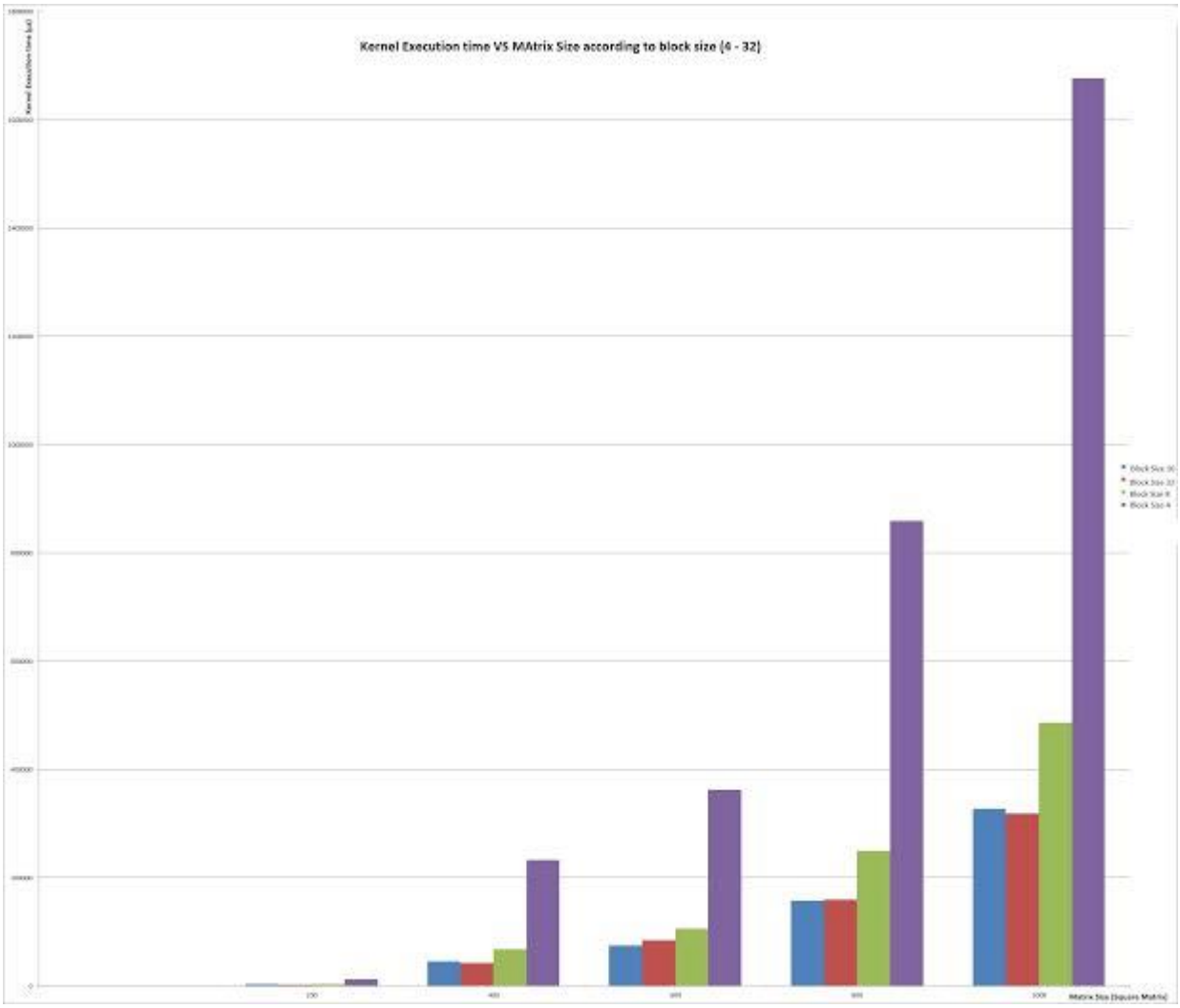

Figure 17 kernel execution time (µs) vs. matrix size based on various Block Sizes

Analysis:

As we can see, we are getting noticeable decrement of time on same size matrix calculation between serial and parallel implementation. A question can be arise here, why in 200 size square matrix multiplications there are no difference between 32 and 4 Block size but have huge difference on 1000 size square matrix multiplication? This answer would be found in section X.

Findings:

Matrix multiplication is a specific type of application where the charisma of CUDA really shines. To avoid multi-faceted iterations that might prove to be a huge deal of time consumption, it is a good idea to implement GPU parallelism rather than relying upon CPU's serial implementations. We have considered not including memory transfer time not to be included as it does not serve the purpose of our pursuit and

left it out of the complete equation. But if the user intends to compute the time inclusive of memory transfer time use of __shared__ memory can be good example for such problem.

4.3 Case Study 3: Adaptive Merge sort algorithm in GPU

Serial and parallel implementation

4.3.1 Overview

The algorithm we work with is the modified version of adaptive merge sort. Any raw data set contains some natural order or sequence among them. Even in the most disordered situation at least two elements have an ordered sequence, may be increasing or decreasing. At first we are going to find those natural ordered sequence and mark them using **flag** based on ascending or descending means divide them into smaller **sub list** or **node.** After that every consecutive pair of one merge and create a new node in sorted order.

4.3.2. Index based sorting in serial

Key word that had been used:

- high_node
- low_node
- total merge

Every time merge operation occurs, it happens between two consecutive pair of node, like node0 with node1 or node4 with node5. Here, for this example low_nodes are 0 and 4 and high_nodes are 1 and 5.

A one-time merge is happened between two consecutive node from 0 to N nodes is termed here as total merge. And this total merge will be occurs ($\log_2$ (Number of nodes)) times, which is as height.

4.3.2.1 Program Flowchart

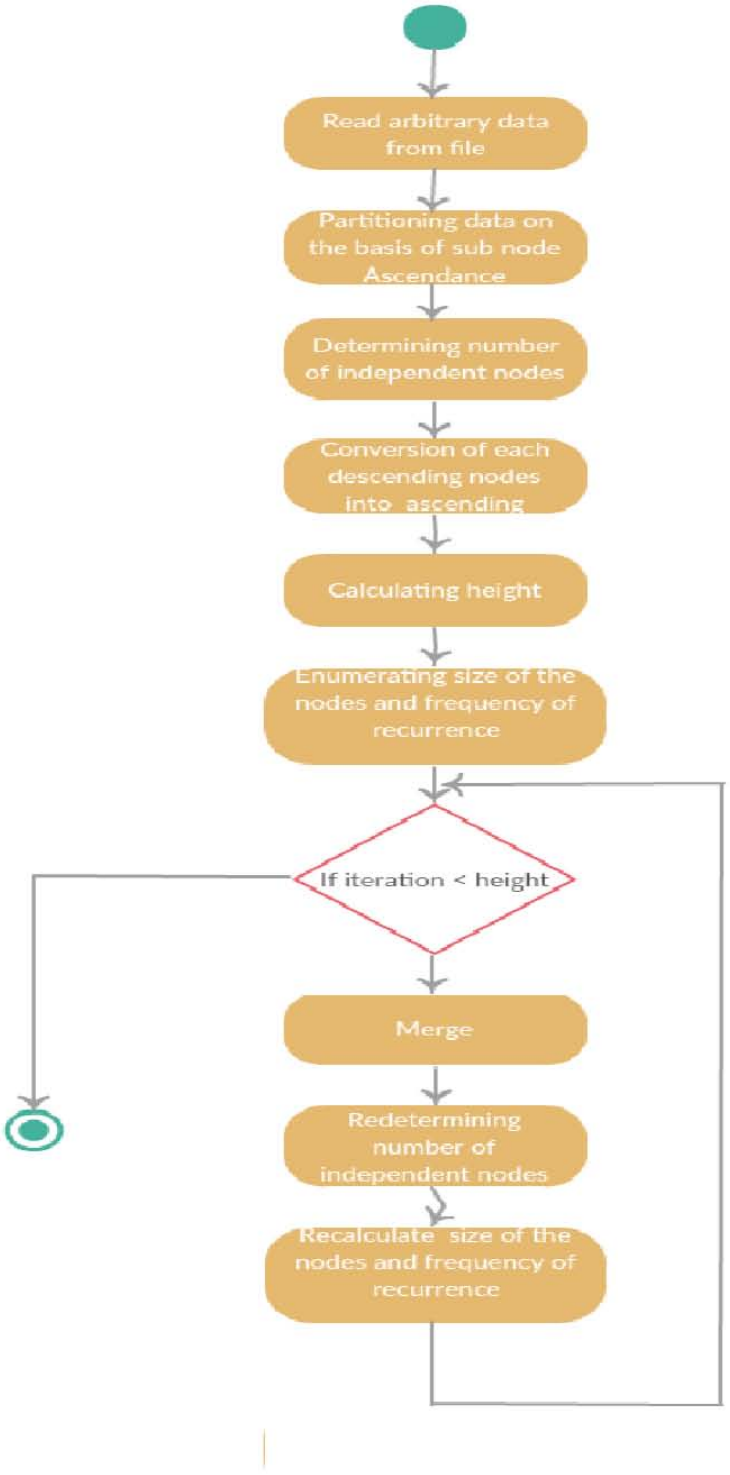

Figure 18 Flow chart of serial implementation

The Flow chart is described in the next

4.3.2.2 Description of flow chart

Read arbitrary from File:
At first it read the unsorted data set and store it in a array. This array will be processed and hold the final sorted array. To understand the total process let's consider a data set with 20 values in it. Data set is

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
num[index]	20	45	8	8	4	4	32	51	76	7	3	7	91	64	89	0	31	19	15	0

Table 4 array example of our implementation in serial of adaptive merge sort

Partitioning the data into node

The partition function divide the array into small nodes according to ascending or descending order and find out the start and ending index of every node along with assign a 0 for ascending or 1 for descending. This function returns the total no. of node and use array "num[]" as parameter. We use three array name "start_ind[node], end_ind[node], as_ds[node]" to store the starting index, ending index and flag respectively.
After processing we will get nodes like following:

start_ind[0] = 0 start_ind[1] = 2 start_ind[2] = 6 start_ind[3] = 9
end_ind[0] = 1 end_ind[1] = 5 end_ind[2] = 8 end_ind[3] = 10
as_ds[0] = 0 as_ds[1] = 1 as_ds[2] = 0 as_ds[3] = 1

start_ind[4] = 11 start_ind[5] = 13 start_ind[6] = 15 start_ind[7] = 17
end_ind[4] = 12 end_ind[5] = 14 end_ind[6] = 16 end_ind[7] = 19
as_ds[4] = 0 as_ds[5] = 0 as_ds[6] = 0 as_ds[7] = 1

index_value = 8 [# of node is 8]

Determining number of independent node

Number of total node counted here.

index_value = 8 [# of node is 8]

Conversion of each descending nodes to ascending

Those nodes that have **as_ds** value '1' are identifying as descending ordered. In this part all nodes with descending order convert into ascending order and change their ascending value into 0 along with their starting or ending index unchanged.

For node [1]:

start_ind[1] = 2
end_ind[1] = 5
as_ds[1] = 0

So, the new array will be

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
num[index]	20	45	4	4	8	8	32	51	76	3	7	7	91	64	89	0	31	0	15	19

Table 5 array example of our implementation in serial of adaptive merge sort

Calculating height

The function calculates the height of the binary tree than could be made by nodes.

Height = 3

Enumerating the sizes of the nodes and frequency of recurrence

The function also will calculate the different sizes of nodes and the frequency of their appearance and to store that information it use two array name **size** and **freq** respectively.

Index	0	1	2
size[index]	2	4	3

Table 6 array of size

Index	0	1	2
freq[index]	5	1	2

Table 7 array of frequency

Merging

In merging section only one data of the data set is processed per iteration. The function '**merge**' we define works in 3 steps

a. **Find out probable node:** When we work with one of the data in data set it finds the node no. which contains that data. If the node index is even then that node is low_node. So, high_node will be (low_node + 1). Values of **size** and **freq** will use to calculate approximate average size of nodes and use it to find out the node index in which the processing data is on.

b. **Position searching:** merging are occurs between a low and high node. If the node that contains the value is low then it will check with all data in high node until it finds bigger value than itself and fetch the index of that data. Than it calculate the new position of that data in merged node using this equation:

   New position = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]);

c. **Update new values:** After sorting it will update the "start_ind[]", "end_ind[]" with start index and end index of new merged node. Every merge node will have starting index equal to starting index of low node and ending index equal to ending index of high node. New merged node will be indexed as the ½ of index of low node.

After that, section 5.2.1.4, 5.2.1.6 and 5.2.1.7 repeats for (height -1) times.

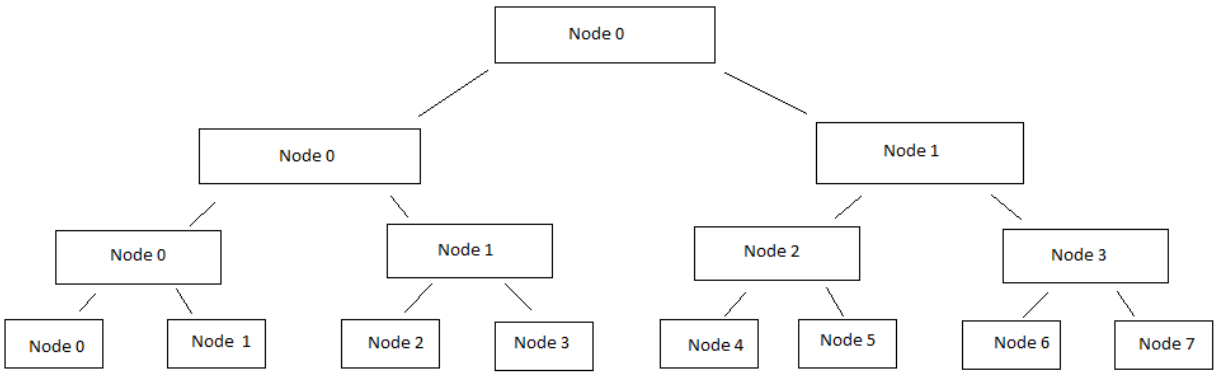

Figure 19 merging 8 nodes represented by binary tree

4.3.3 Merging Procedure

Merge (current index)

Begin

Value position percentage = (current index * 100) / size of data;

Approx. node numbers = size of data / average node size;

Probable node = (approx. node numbers*value position percentage) / 100;

While (1)

Begin

If current index is between start and end index of probable node

Then Break

Else if for any other situation

Then decrease or increase value of probable node until current index become in between start and end index of probable node, then Break

End

If probable node is even and total nodes number is even and probable nodes isn't the last node

Then check, how many data in next node are less than or equal the data in current index and add this number with its own position in probable node then insert the data in that position. Create a new node with starting index equal to starting index of probable node and ending index equal to ending index of next node. New node will be indexed as the ½ of index of probable node.

If probable node is even and total nodes number is odd and probable nodes is the last node

Then, create a new node with starting index equal to starting index of probable node and ending index equal to ending index of same node. New node will be indexed as the ½ of index of probable node.

Else

Then check, how many data in previous node are less than the data in current index and add this number with its own position in probable node then insert the data in that position. Create a new node with starting index equal to starting index of probable node and ending index equal to ending index of next node. New node will be indexed as the ½ of index of probable node.

End

4.3.4 Parallel implementation

In parallel implementation is very similar with the serial one. Two procedures from serial implementation are executing in parallel on parallel implementation. Those processes are

- Conversion of each descending nodes to ascending
  In serial application we have to convert every node which is in descending order one per iteration. Which can be equal to (size of data/2) in worst case scenario. But if we check all nodes in different threads and run in parallel we can check and convert multiple numbers of nodes in a single time period, 4096 nodes to be exact on the program we have written.
- Merging
  We transform this part into kernel function as for the same reason we've done the previous one. Merge function must be invoke times equal to data size according to our design.

Now we will observe parallel implementation using an example. Let's consider a data set

INDEX	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
VALUE	17	51	64	94	17	17	18	5	4	2	1	0	18	15	14	64	18	17	5	0

Figure 20 Sample Data set of parallel implementation of adaptive merge sort

Green and red markings indicate start and ending index.

4.3.4.1 Step by Step workflow

After completing the partitioning

This partitioning is exactly same as portioning on serial implementation

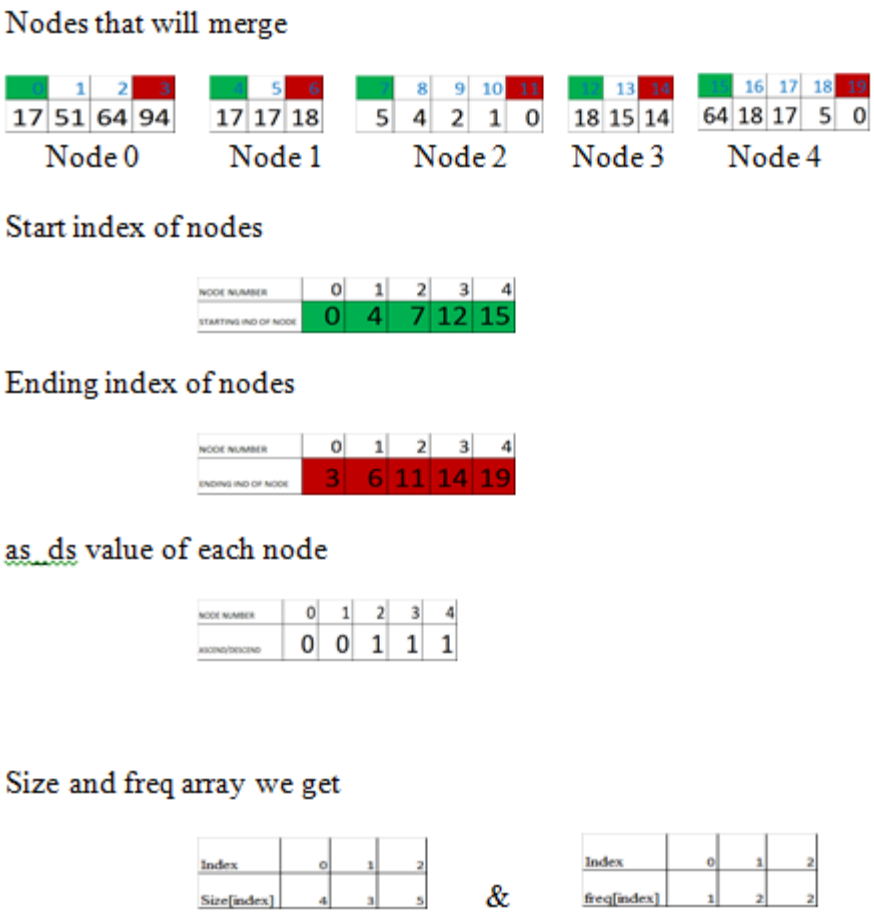

Figure 21 Overall status of all arrays after partition

Convert all descending nodes to ascending

This kernel will execute in GPU and will work with 5 threads only thread with ID 2, 3, 4 will execute the function.

__global__ void merge_myway_ascend(int *d_num, int *d_start_ind, int *d_end_ind, int *d_as_ds, int *swapper)

And invoking kernel function in main
merge_myway_ascend << <1, nonodes >> >(d_num, d_start_ind, d_end_ind, d_as_ds, swapper);

And the outputs are

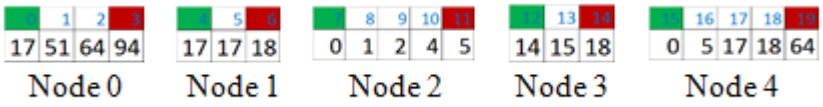

Figure 22 Nodes after conversion

Height = 3

Parameter needs to invoke the merge kernel

The following values need to pass as parameters to invoke a kernel
Size of data = 20
Aproxnode size = ∑(size[i]*freq[i]) / ∑ freq[i] = 4

Invoke merge kernel

On thread 0:

Data [0] = 17, probable node = 0 & 0 is a low_node 1 is high_node

17 will check in Node 1 that, at which index the data is bigger than 17, which is 1

New position of 18 is

New position = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]);

New position = 0 + (6 - 4) + (0 - 0) = 2

The new merged node will be indexed as ½ of low which is 0

Start index of new node 0 will be **start_ind[low] =** 0

Ending index of new node will be **end _ind[high]** = 6

Let's observe thread 6:

Data [6] = 18, probable node = 1 & 1 is a high_node 0 is low_node

18 will check in Node 0 that, at which index the data is bigger or equal to 18, which is 3

New position of 17 is

New position = start_ind[low] + (fetched index - start_ind[low]) + (index of itself in main dataset - start_ind[high]);

New position = 0 + (1 - 0) + (6 - 4) = 3

Start index of new node 0 will be **start_ind[low]** = 0

Ending index of new node will be **end _ind[high]** = 6

After first level merging

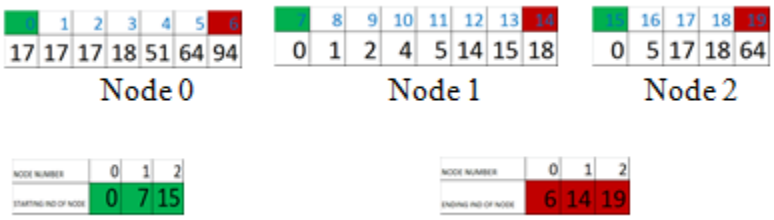

Figure 23 Main data set and Start and end INDEX of node after 1st level merging

After 2nd level merging

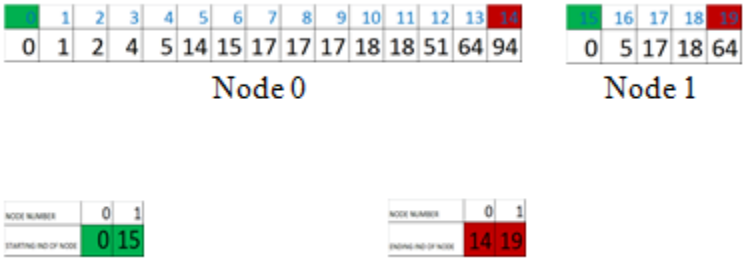

Figure 24 Main data set and Start and end INDEX of node after 2nd level merging

After final merging

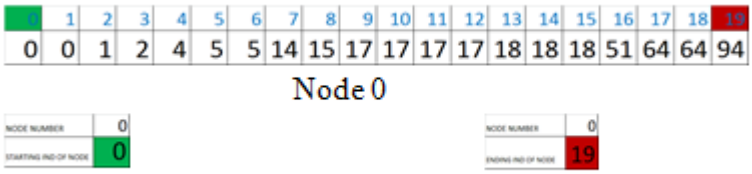

Figure 25 Main data set and Start and end INDEX of node after 2nd level merging

Total number of level needed = 3 = height

4.3.5 Time Analysis

The GPU and CPU performance will be compared in this section. We took data of time consumption to merge the nodes for both CPU and GPU. In figure 4.16 and table 4.6, we can see the CPU execution time is far more than GPU at 1024 block size

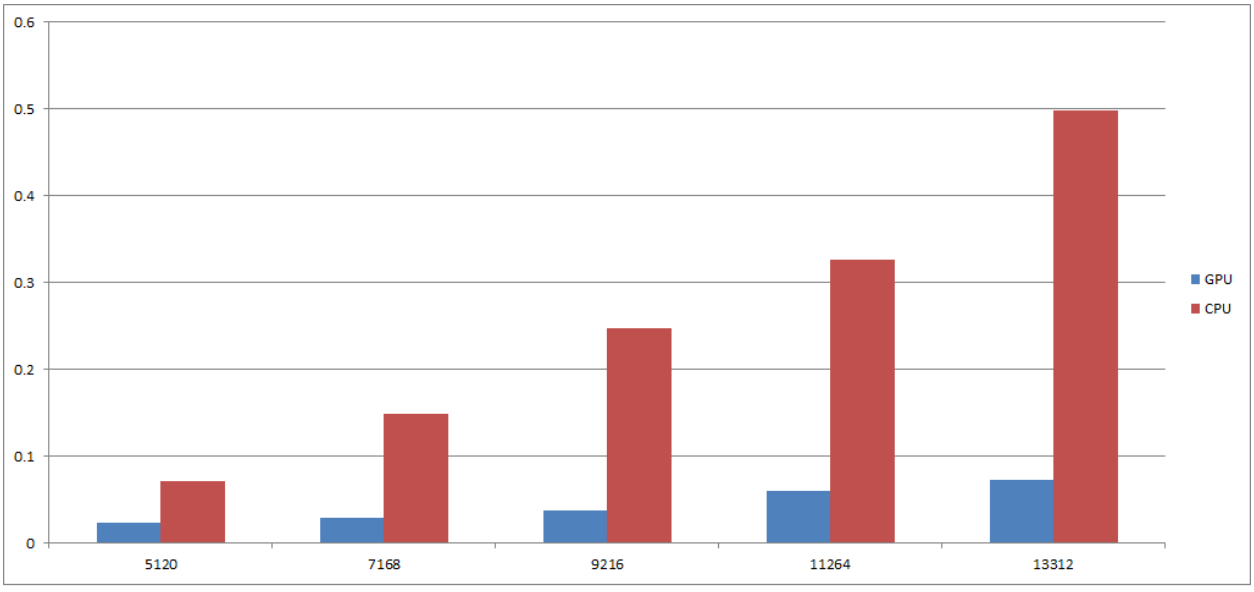

Figure 26 Time comparison graph Between CPU and GPU

DATA	GPU Time	CPU Time
5120	0.023636	0.072
7168	0.029439	0.149
9216	0.038158	0.248
11264	0.06062	0.327
13312	0.072206	0.498

Table 8 CPU Vs GPU execution time (1024 blocksize)

Optimum blocksize detection:

In this section, we fixed the data size in 5120 and run the kernel in different block size. Our goal is to find the optimum Block size to get the best performance. We can see, 128 thread per block is the most optimum block size.

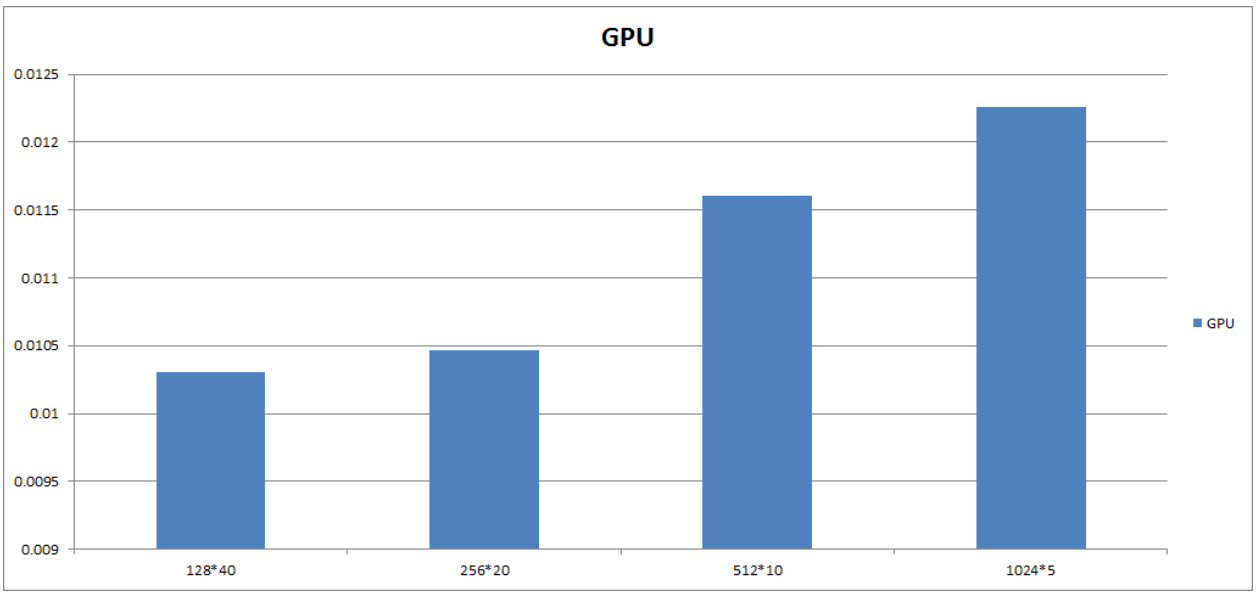

Figure 27 Time comparison graph Between different CUDA block

BLOCKSIZE*GRIDSIZE	GPU Time
128*40	0.010305
256*20	0.010469
512*10	0.011609
1024*5	0.012256

Table 9 block size vs. time of execution

Now we have come to our decision about optimal size of Block. We have again done the 1st experiment and observe the result

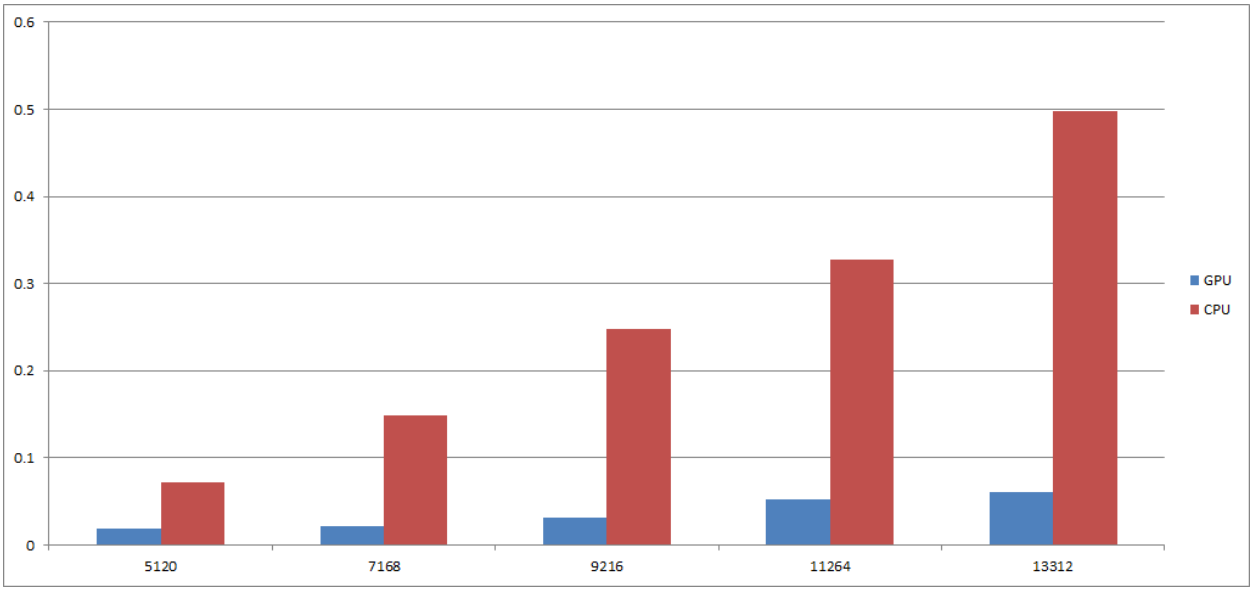

Figure 28 Time comparison graph Between CPU and GPU (128 block size)

DATA	GPU Time	CPU Time
5120	0.019048	0.072
7168	0.022364	0.149
9216	0.031158	0.248
11264	0.052972	0.327
13312	0.060832	0.498

Table 10 CPU Vs GPU execution time (block size 128)

Chapter 5

Conclusion

5.1 Conclusion

Towards the end of our experiment we have been stalled with some recurring problem with our approach. We have been limited to experiment with 13312 values at a time. This problem can be referred to memory issues we have briefly explained in the previous sections. To overhaul such problem the programmer must limit the number of declared variables, use shared memories, and change the allocation of variables after each iterations depending on requirements.

While designing the algorithm one must make sure of keeping the variables declared within 65536 numbers of variables. Due to decision parameter we were not able to alter the maximum capacity of our program to run on by using shared memory.

If the programmer intends to execute two Kernel in parallel then the only way to do so includes installing two device (GPU).

Apart from those unexpected memory issues that we have experienced the execution time seemed to have a drastic improvement. From that perspective it can be stated that for adaptive merge sort it is a better solution to be implied in GPU rather than in CPU. And drawing from that experience that we have gathered we can also ensure that merging techniques are much more of calculation dependent which can only be done with effectiveness and lesser consumption of time if done in parallel .So we hope and believe that more merging techniques would evoke in the near future using the power of GPU and which for sure would become a smarter choice for the sorting techniques analysts.

5.2 Future work

Concentrating on memory usage and reducing I/O time can make the application more efficient.

Appendix:

Code for parallel implementation of Adaptive merge sort in CUDA

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <math.h>
#include <time.h>
#define numsize 20

int maxnodesize = 0;
int num[numsize];
int start_ind[numsize / 2];
int end_ind[numsize / 2];
int as_ds[numsize / 2];
int size[200], freq[200];


int ind = 0;


void readnumbers()

{
	int i = 0;
	int var;
	FILE *file = fopen("numbers.txt", "r");
	fscanf(file, "%d", &var);
	while (!feof(file))
	{
		num[i] = var;
		i++;
		fscanf(file, "%d", &var);

	}
```

```
	fclose(file);
}


int partition(int *num)
{
	int nodesize;
	int temp = num[0];
	int i = 1; int ind = 0;
	int start = 0, end = 0, flag = 404;
	while (1)
	{
		while (1)
		{

			if (temp <= num[i] && i < numsize)
			{
				temp = num[i];
				flag = 0;
				end = i;
				i++;
				continue;
			}
			break;
		}
		if (!flag)
		{
			start_ind[ind] = start;
			end_ind[ind] = end;
			as_ds[ind] = flag;
			ind++;

			nodesize = (end - start) + 1;

			start = end = i;
			temp = num[i++];


			printf("\nnodesize:%d", nodesize);
			int a = 0;
			while (a < 200)
			{
				if (size[a] == nodesize)
```

```
                {
                        freq[a] = freq[a] + 1;
                        break;
                }
                else if (size[a] == 0)
                {
                        size[a] = nodesize;
                        freq[a] = freq[a] + 1;
                        break;
                }
                else a++;
        }
}

while (1)
{
        if (temp >= num[i] && i < numsize)
        {
                temp = num[i];
                flag = 1;
                end = i;
                i++;
                continue;
        }
        break;
}
if (flag)
{
        start_ind[ind] = start;
        end_ind[ind] = end;
        as_ds[ind] = flag;
        ind++;

        nodesize = (end - start) + 1;

        start = end = i;
        temp = num[i++];


        printf("\nnodesize:%d", nodesize);
        int a = 0;
        while (a < 200)
        {
                if (size[a] == nodesize)
```

```
                                {
                                        freq[a] = freq[a] + 1;
                                        break;
                                }
                                else if (size[a] == 0)
                                {
                                        size[a] = nodesize;
                                        freq[a] = freq[a] + 1;
                                        break;
                                }
                                else a++;
                        }

                }
                if (end == numsize) { printf("indexvalue : %d", ind); break; }
        }

        return ind;
}

int approxnodesize()
{
        int sumsizefreq = 0, sumfreq = 0;
        for (int i = 0; size[i] != 0; i++)
        {
                if (size[i] > maxnodesize) { maxnodesize = size[i]; }
                //printf("\nsize[%d]:%d\tfreq[%d]:%d", i, size[i], i, freq[i]);
                sumsizefreq = sumsizefreq + (size[i] * freq[i]);
                sumfreq = sumfreq + freq[i];
        }
        //rintf("\n%d : %f", approxnodesize);
        int approxnodesize = (sumsizefreq / sumfreq);
        printf("\napproxnode : %d", approxnodesize);
        return approxnodesize;

}

int height(int nodenumbers)
{
        int node = nodenumbers;
        int height = 0;
        while (1)
        {
                node = node / 2 + node % 2;
```

```
            height++;
            if (node == 1) break;

        }
        printf("\n height : %d\n", height);

        return height;
}

__global__ void merge_myway_ascend(int *d_num, int *d_start_ind, int *d_end_ind, int *d_as_ds,
int *swapper)
{
        int position_of_thread = blockIdx.x * blockDim.x + threadIdx.x;
        int low = position_of_thread * 2;
        int high = low + 1;
        int nodelowstart = d_start_ind[low];
        int nodelowend = d_end_ind[low];
        int nodehighstart = d_start_ind[high];
        int nodehighend = d_end_ind[high];
        //      int swapper[500];
        if (d_as_ds[low] == 1)
        {
                int z = nodelowstart;
                int n = nodelowstart;
                int s = nodelowend;

                while (s >= nodelowstart)
                {
                        swapper[z] = d_num[s];
                        z++; s--;
                }
                z = nodelowstart;

                while (n <= nodelowend)
                {
                        d_num[n] = swapper[z];
                        n++; z++;
                }
                d_as_ds[low] = 0;
        }

        if (d_as_ds[high] == 1)
        {
                int z = nodehighstart;
```

```
                int m = nodehighstart;
                int r = nodehighend;

                while (r >= nodehighstart)
                {
                        swapper[z] = d_num[r];
                        z++; r--;
                }
                z = nodehighstart;

                while (m <= nodehighend)
                {
                        d_num[m] = swapper[z];
                        m++; z++;
                }
                d_as_ds[high] = 0;
        }
}


__global__ void merge(int approxnodesize, int numsizee, int *d_num, int *d_start_ind, int *d_end_ind, int *store, int nonodes, int odd, int *new_d_start_ind, int *new_d_end_ind)
{
        //printf("\nblock:%d\tthread:%d\ti:%d\td_num[%d]:%d,blockIdx.x,threadIdx.x,i,i,d_num[i]");
        int low, high;
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int valuepositionpercentage = (i * 100) / numsizee;
        int approxnodenumbers = numsizee / approxnodesize;
        int probablenode = (approxnodenumbers*valuepositionpercentage) / 100;
        //int new_d_start_ind[500], new_d_end_ind[500];
        while (1)
        {



        //printf("\nblock:%d\tthread:%d\ti:%d\td_num[%d]:%d,blockIdx.x,threadIdx.x,i,i,d_num[i]");
                if (i >= d_start_ind[probablenode] && i <= d_end_ind[probablenode])
                {
                        break;
                }
                else if (d_start_ind[probablenode] == 0 && d_end_ind[probablenode] == 0)
                {
```

```
                probablenode--;
                break;
        }
        else if (i < d_start_ind[probablenode] && i < d_end_ind[probablenode])
        {
                while (1)
                {
                        probablenode--;
                        if (i >= d_start_ind[probablenode] && i <=
d_end_ind[probablenode]){ break; }
                }
        }//break;
        else if (i > d_start_ind[probablenode] && i > d_end_ind[probablenode])
        {
                while (1)
                {
                        probablenode++;
                        if (i >= d_start_ind[probablenode] && i <=
d_end_ind[probablenode]){ break; }
                }

        }break;
}

printf("\nblock:%d\tthread:%d\ti:%d\td_num[%d]:%d", blockIdx.x, threadIdx.x, i, i,
d_num[i]);

if (probablenode % 2 == 0)
{
        if (odd == 1 && probablenode == nonodes - 1)
        {
                printf("\nstore[%d]:%d\td_num[%d]:%d", i, store[i], i, d_num[i]);
                store[i] = d_num[i];
                new_d_start_ind[probablenode / 2] = d_start_ind[probablenode];
                new_d_end_ind[probablenode / 2] = d_end_ind[probablenode];

        }



        else
        {

                low = probablenode;
```

```
                        high = probablenode + 1;
                        int p = d_start_ind[high];
                        while (p <= d_end_ind[high])
                        {
                                if (d_num[i] >= d_num[p]) p++;
                                else break;
                        }
                        int newposition = d_start_ind[low] + (p - d_start_ind[high]) + (i -
d_start_ind[low]);
                        __syncthreads();
                        //printf("\nblock:%d\tthread:%d\ti:%d\td_num[%d]:%d",       blockIdx.x,
threadIdx.x, i, i, d_num[i]);
                        store[newposition] = d_num[i];
                        new_d_start_ind[low / 2] = d_start_ind[low];
                        new_d_end_ind[low / 2] = d_end_ind[high];
                }
        }
        else
        {
                high = probablenode;
                low = probablenode - 1;
                int p = d_start_ind[low];
                while (p <= d_end_ind[low])
                {
                        if (d_num[i] > d_num[p]) p++;
                        else if (d_num[i] == d_num[p]) break;
                        else break;
                }
                int newposition = d_start_ind[low] + (p - d_start_ind[low]) + (i - d_start_ind[high]);
                __syncthreads();
                store[newposition] = d_num[i];
                new_d_start_ind[low / 2] = d_start_ind[low];
                new_d_end_ind[low / 2] = d_end_ind[high];

        }
        printf("\nblock:%d\tthread:%d\ti:%d\td_num[%d]:%d\tstore[%d]:%d",       blockIdx.x,
threadIdx.x, i, i, d_num[i], i, store[i]);

}

void afterkernelnodesize(int nonodes_passingtodevice, int *host_new_end_ind,int
*host_new_start_ind)
{
```

```
	for (int loop = 0; loop < nonodes_passingtodevice; loop++)
	{
		int j = 0;
		int afterkernelnodesize = (host_new_end_ind[loop] - host_new_start_ind[loop]) + 1;
		while (1)
		{
			if (afterkernelnodesize == size[j])
			{
				freq[j] = freq[j] + 1;
				break;
			}
			else if (size[j] == 0)
			{
				size[j] = afterkernelnodesize;
				freq[j] = freq[j] + 1;
				break;
			}
			else if (size[j] != afterkernelnodesize && size[j] != 0)
				j++;

		}


	}

	//int newappnodesize = approxnodesize();
	//return newappnodesize;
}
int main()
{
	dim3 blockIdx;
	dim3 threadIdx;
	dim3 blockDim;

	int i = 0; int nonodes;
	int host_new_start_ind[200], host_new_end_ind[200];
	int odd;
	//int size[200] = { 0 }, freq[200] = {0};
	readnumbers();

	while (i < numsize)
	{
		printf("num[%d] : %d \t", i, num[i]);
		i++;
```

```
        }
        nonodes = partition(num);

        int h = height(nonodes);
        printf("\nnodes: %d\n", nonodes);
        printf("\nheight: %d\n", h);
        for (int i = 0; i <= nonodes; i++)
        {
                printf("\nstart[%d]:%d\t", i, start_ind[i]);
                printf("end[%d]:%d\t", i, end_ind[i]);
                printf("as_ds[%d]:%d\t", i, as_ds[i]);

        }


        //printf("\n\napp nodesize:%d", y);
        int *d_num, *d_start_ind, *d_end_ind, *d_as_ds, *swapper, *store, *new_d_start_ind, *new_d_end_ind;

        cudaMalloc(&d_num, numsize*sizeof(int));
        cudaMalloc(&d_start_ind, numsize / 2 * sizeof(int));
        cudaMalloc(&d_end_ind, numsize / 2 * sizeof(int));
        cudaMalloc(&d_as_ds, numsize / 2 * sizeof(int));
        cudaMalloc(&swapper, maxnodesize * sizeof(int)* 500);
        cudaMalloc(&store, numsize*sizeof(int));
        cudaMalloc(&new_d_start_ind, numsize / 2 * sizeof(int));
        cudaMalloc(&new_d_end_ind, numsize / 2 * sizeof(int));

        cudaMemcpy(d_num, &num, numsize*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_start_ind, &start_ind, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_end_ind, &end_ind, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_as_ds, &as_ds, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);


        merge_myway_ascend << <1, nonodes >> >(d_num, d_start_ind, d_end_ind, d_as_ds, swapper);

        cudaMemcpy(&num, d_num, numsize*sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(&start_ind, d_start_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(&end_ind, d_end_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(&as_ds, d_as_ds, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
```

```
for (int i = 0; i <= nonodes; i++)
{
        printf("\nAscendstart[%d]:%d\t", i, start_ind[i]);
        printf("Ascendend[%d]:%d\t", i, end_ind[i]);
        printf("Ascendas_ds[%d]:%d\t", i, as_ds[i]);

}
int u = 0;
while (u < numsize)
{
        printf("num[%d] : %d \t", u, num[u]);
        u++;
}


cudaMemcpy(d_num, &num, numsize*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_start_ind, &start_ind, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_end_ind, &end_ind, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_as_ds, &as_ds, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);

int approxnodeSize = approxnodesize();
printf("\nbefore karnel app node size :%d", approxnodeSize);

int n = numsize;
int nonodes_passingtodevice = nonodes;

for (int ll = 0; ll <= h; ll++)
{

        if (nonodes_passingtodevice % 2 == 0)
        {
                odd = 0;
        }
        else
        {
                odd = 1;
        }


                merge <<< 1, n >>>(approxnodeSize, n, d_num, d_start_ind, d_end_ind, store, nonodes_passingtodevice, odd, new_d_start_ind, new_d_end_ind);
```

```
		nonodes_passingtodevice = ((nonodes_passingtodevice / 2) + (nonodes_passingtodevice % 2));

		printf("\nNODES:%d", nonodes_passingtodevice);


		cudaMemcpy(&host_new_start_ind, new_d_start_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost); //rework
		cudaMemcpy(&host_new_end_ind, new_d_end_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost); //rework
		cudaMemcpy(&num, store, numsize*sizeof(int), cudaMemcpyDeviceToHost);

		cudaMemcpy(d_num, &num, numsize*sizeof(int), cudaMemcpyHostToDevice);
		cudaMemcpy(d_start_ind, &host_new_start_ind, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);
		cudaMemcpy(d_end_ind, &host_new_end_ind, numsize / 2 * sizeof(int), cudaMemcpyHostToDevice);


		size[200] = { 0 }; freq[200] = { 0 };
		afterkernelnodesize(nonodes_passingtodevice, host_new_end_ind, host_new_start_ind);
		int approxnodeSize = approxnodesize();
		printf("\ninkarnel app node size :%d", approxnodeSize);
	}

	/*
	//int newsize[20] = { 0 }, newfreq[20] = {0}, afterkernelnodesize;




	//cudaMemcpy(&num, store, numsize*sizeof(int), cudaMemcpyDeviceToHost);

	cudaMemcpy(&num, store, numsize*sizeof(int), cudaMemcpyDeviceToHost);
	int v = 0;
	while (v <numsize)
	{
		printf("\nafter 1st merge-->num[%d] : %d \t", v, num[v]);
		v++;
	}
	//cudaMemcpy(&start_ind, d_start_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
```

```
	//cudaMemcpy(&end_ind, d_end_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
	//cudaMemcpy(&as_ds, d_as_ds, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
	//cudaMemcpy(&start_ind, d_start_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
	//cudaMemcpy(&end_ind, d_end_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);



	//cudaMemcpy(&start_ind, d_start_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);
	//cudaMemcpy(&start_ind, d_start_ind, numsize / 2 * sizeof(int), cudaMemcpyDeviceToHost);


	for (int loop = 0; loop < nonodes_passingtodevice; loop++)
	{
	printf("\nafter 1st merge-->newsize[%d]:%d", loop, newsize[loop]);
	printf("\nafter 1st merge-->newfreq[%d]:%d", loop, newfreq[loop]);
	}

	printf("\nafter 1st merge-->newapproxnodesize:%d",newappnodesize);

	printf("\nafter 1st merge--> nodes:%d", nonodes_passingtodevice);
	printf("\nafter 1st merge-->odd=%d", odd);
	for (int i = 0; i < 10; i++)
	{
		printf("\nafter 1st merge -->start[%d]:%d\tend[%d]:%d", i, host_new_start_ind[i], i, host_new_end_ind[i]);
	}



		merge << <1, n >> >(newappnodesize, n, d_num, d_start_ind, d_end_ind, store, nonodes_passingtodevice, odd, new_d_start_ind, new_d_end_ind);
		*/
	cudaMemcpy(&num, store, numsize*sizeof(int), cudaMemcpyDeviceToHost);
	int vv = 0;
	while (vv <numsize)
	{
		printf("\n num[%d] : %d \t", vv, num[vv]);
```

```
            vv++;
        }

        cudaFree(d_num);
        cudaFree(d_start_ind);
        cudaFree(d_end_ind);
        cudaFree(d_as_ds);
        cudaFree(swapper);
        cudaFree(store);
        cudaFree(new_d_start_ind);
        cudaFree(new_d_end_ind);
        //cudaFree(p);

        return 0;
}
```

Reference

[1] CUDA Programming. A Developer's Guide to Parallel Computing with GPUs
By Shane Cook

[2] Matrix Multiplication with CUDA | A basic introduction to the CUDA programming model
By Robert Hochberg

[3] CUDA C PROGRAMMING GUIDE By Nvidia corporation

[4] Parallel Implementation of Video Surveillance Algorithms on GPU Architecture using CUDA by Sanyam Mehta‡ , Arindam Misra‡ , Ayush Singhal‡ , Praveen Kumar† , Ankush Mittal‡ , Kannappan Palaniappan

[5] A Parallel Error Diffusion Implementation on a GPU by Yao Zhanga , John Ludd Reckerb , Robert Ulichneyc , Giordano B. Berettab , Ingeborg Tastlb , I-Jong Linb , John D. Owensa

[6] HETEROGENEOUS HIGHLY PARALLEL IMPLEMENTATION OF MATRIX EXPONENTIATION USING GPU by Chittampally Vasanth Raja, Srinivas Balasubramanian, Prakash S Raghavendra By

[7] Parallel Implementation of Otsu's Binarization Approach on GPU By Brij Mohan Singh, Ankush Mittal, Debashish Ghosh

[8] Adaptive Merge Sort. By Nenwani Kamlesh, Vanita Mane, Smita Bharne

Additional References

[9] Practical Applications for CUDA (http://supercomputingblog.com/cuda/practical-applicationsfor-cuda/)

[10] Matthew Guidry, Charles McClendon, "Parallel Programming with CUDA".

[11] http://www.top500.org/blog/2009/05/20/top_trends_high_performance_computing
Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012 120

[12] Danilo De Donno et al., "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD," IEEE Antennas and Propagation Magazine, June 2010

[13] Miguel C´ardenas-Montes," Accelerating Particle Swarm Algorithm with GPGPU," 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2011

[14] http://supercomputingblog.com/cuda/practical-applications-for-cuda
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter35.htm

[15] http://www.idav.ucdavis.edu/