Application Development for multicore Processor

Submitted in partial fulfillment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Roll No	Names of Students
110CS0118	John Diptikanta Behera
110CS0152	Biswa Ranjan Sethy
110CS0480	Aditya Kumar

Under the guidance of
Prof. B D Sahoo

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, India – 769008

2010-2014 batch

Application Development for multicore Processor

Submitted in partial fulfillment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Roll No	Names of Students
110CS0118	John Diptikanta Behera
110CS0152	Biswa Ranjan Sethy
110CS0480	Aditya Kumar

Under the guidance of
Prof. B D Sahoo

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, India – 769008

2010-2014 batch

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Certificate

This is to certify that the work in this Thesis Report entitled Application Development For Multicore Processor by students whose names are given below has been carried out under my supervision in partial fulfillment of the requirements for the degree of Bachelor of Technology in Computer Science and Engineering,during session 2013-2014 in the Department of Computer Science and Engineering, National Institute of Technology, Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere

Roll No	Names of Students
110CS0118	John Diptikanta Behera
110CS0154	Biswa Ranjan Sethy
110CS0148	Aditya Kumar

Prof B D Sahoo
(Dept Of Computer Science and Engineering)

Acknowledgments

It is not possible to complete a thesis without considerable support of many great people. First and foremost, We are greatly indebted to our advisor Prof. Bibudatta sahoo, Professor in CSE department of NIT Rourkela for his guidance, encouragement, motivation, and continued support throughout our academic years at NIT. He has allowed us to pursue our research interests with sufficient freedom, while always being there to guide us. Working with him has been one of the most rewarding experiences of our Btech life.

we have been fortunate to have met great friends throughout our BTECH journey. We are forever grateful for their moral support, encouragement, and true friendship

We are grateful to all the staffs of the computer science department for their generous help in various ways for the completion of this thesis. Last but not least, We would forever indebted to our parents. They have been a great source of inspiration to us. This would have not been possible without their love, support, and continuous encouragement.

(John Diptikanta Behera) (Adityas Kumar) (Biswa Ranjan Sethy)

May 2014
National Institute of Technology Rourkela

Abstract

With multicore processors now in every computer, server, and embedded device, the need for cost-effective, reliable parallel software has never been greater. The efficiency of single core processors does not match the necessary levels for the development of applications. Performance means more than wringing additional benefits from a single application because users commonly multi- task, actively toggling between two or more applications or working in environments in which many background processes compete for scarce processor resources. All major Operating systems, including Mac OS X, Microsoft Windows Vista, Windows Server, Red Hat Linux, and Novell SuSE Linux, already are threaded to take advantage of Hyper-Threading Technology and now multi-core architecture. Any program that is in a class of applications where threading is already relatively common video encoding, 3D rendering, video/photo editing and high performance computing/workstation applications are good candidates to be moved from serial to multithreaded multi-core systems. The speed up achieved can be measured using several laws like Ahmdals law, Gustafans law, etc. Multi-core architectures like the C64 could be used to achieve a high performance implementation of FFT both in 1D and 2D cases. Graphics processing units(GPUs) are further improvements on multicore architecture that process the graphics and the CPU codes in parallel, to give improved results. One such example is the speed up achieved in implementing the Radix sort on the CPU and the GPU.

Keywords: multicore architecture , Ahmdals law,Gustafans law, implementing FFT, sorting on GPU

Contents

List of Figures

Chapter 1

Introduction

The line "Slow and steady wins the race" now-a-days does not hold true in a lot of fields. The proverb loses its effectiveness as we have a mountain load of works each and every day. So these days its all about fast and accurate work. Time complexity is a big issue in our day to day life. We are suffering from these kind of problems in the software field as well.

With the growth of our world, Software world is also developing concurrently in a dynamic manner. In this fast forward world every thing needs to be done shortly and accurately. Time complexity is a big issue as we are loaded with huge amount of work.

With the generation of new tools and techniques ,in quick time it forced the software world to look for new way to design software . Here some approaches are mentioned which really shows some kind of solutions to these problems.

The multicore revolution is within our reach. For supercomputers or clusters ,Parallel processing no longer remains as the exclusive domain. Hardware -level parallel processing and software - level parallel processing are more frequently used every where even in the entry - level server and even the basic developer workstations.The main issue is ,what does this mean for the software developer and what impact will it have on the software development process. In the competetion of manufacturing the fastest computer, it is now more attractive for chip manufacturers to carry multiple processors on a single chip rather than increase the speed of the processor. Until now the software developer could rely on the next new processor to speed up the software without having to make any actual modifications to the software. Those are gone. In order to increase overall system performance efficiently,

computer manufacturers have decided to add more processors rather than increase their clock frequencies. This means if the software developer wants the application to benefit from the next new processor, the application will have to be modified to exploit the core part of the computers.

Though single core application development and sequential programming have a place and will remain with us, the landscape of software development now seems to be shifted toward multithreading and multiprocessing. Parallel programming techniques that were once the only concern of theoretical computer scientists and university academics, are in the process of being reworked for the masses. The ideas of multicore application design and development are now a concern for the mainstream.

1.1 Motivation

The past decade has seen tremendous advances in microprocessor technology. Clock rates of processors have increased from about 40 MHz (e.g., a MIPS R3000, circa 1988) to over 2.0 GHz (e.g., a Pentium 4, circa 2002). At the same time, processors are now capable of executing multiple instructions in the same cycle. The average number of cycles per instruction (CPI) of high end processors has improved by roughly an order of magnitude over the past 10 years. All this translates to an increase in the peak floating point operation execution rate (floating point operations per second, or FLOPS) of several orders of magnitude. A variety of other issues have also become important over the same period. Perhaps the most prominent of these is the ability (or lack thereof) of the memory system to feed data to the processor at the required rate. Significant innovations in architecture and software have addressed the alleviation of bottlenecks posed by the data path and the memory. The role of concurrency in accelerating computing elements has been recognized for several decades. However, their role in providing multiplicity of data paths, increased access to storage elements (both memory and disk), scalable performance, and lower costs is reflected in the wide variety of applications of parallel computing. Desktop machines, engineering workstations, and compute servers with two, four, or even eight processors connected together are becoming common platforms for design applications. Large scale applications in science and engineering rely on larger configurations of parallel computers, often comprising hundreds of processors. Data intensive platforms such as database or web servers and applications such as transaction processing and data mining often use clusters of workstations that provide high aggregate disk bandwidth. Applications in graphics and

visualization use multiple rendering pipes and processing elements to compute and render realistic environments with millions of polygons in real time. Applications requiring high availability rely on parallel and distributed platforms for redundancy. It is therefore extremely important, from the point of view of cost, performance, and application requirements, to understand the principles, tools, and techniques for programming the wide variety of parallel platforms currently available. So,The idea is simple,To improve performance by performing two or more operations at the same time. [1]

1.2 Problem Statement and Objective

The idea of a single processor computer is fast becoming archaic and quaint. We now have to adjust our strategies when it comes to computing if we are having Single core processor,But if we have multicore processor but the program that is meant to run on multicore is written using single core coding convention then we are likely to under utilize the availability of resource at hand.

So the Current technology is fastly moving towards Parallel computing and multicore programming.the reasons may be

- It is impossible to improve computer performance using a single processor. Such processor would consume unacceptable power. It is more practical to use many simple processors to attain the desired performance using perhaps thousands of such simple computers. cite1
- Memory systems are still much slower than processors and their bandwidth is limited also to one word per read/write cycle.
- As a result of the above observation, if an application is not running fast on a single - processor machine, it will run even slower on new machines unless it takes advantage of parallel processing.

This is almost like impossible to attain 100 percentage efficiency and accuracy through single core computer where , performance over huge amount of computational intensive data or information or instruction is concerned. So, our main objective is to develop basic benchmark application to be deployed on Multicore platforms which prove to be run faster on multicore environment as compare to single core implementation.

1.3 Organisation Of The Thesis

This thesis is organised into 4 chapters.
Chapter 1: Chapter 1 Provides basic over view of the need of multicore processors and its implementation along with the limitations of the single core processors.Here we also represent our basic Problem statement and our main objective.
Chapter 2:
In chapter 2 We represent the basic architecture of Multicore processors and the basic principles used while designing a program targeting multicore architecture.We also mentioned the basic Principles involved to show that a problem is a proper candidate for Multicore programming.In this chapter we have mentioned the language and library we have used for the simulation.e have also mentioned the Reason for using those languages and library.in this chapter we have also represnted the basic mathematical formulation to calculate the computational complexity of a problem implemented using multicore processor.
Chapter 3: In this chapter we have justified that FFT is a proper candidate for multicore programming. along with the Algorithm and program design paradigm for FFT in multicore.we have compared the Simulation result and theoretical results.
Chapter 4: in this chapter we have moved little bit far from Multicore processor and Provided a extensive study on NVDIA GPUs.We have implemented Radix sort on GPU and compared the result with the conventional CPU implementation.

Chapter 2

Architecture And Program Design

2.1 Architecture

In general we know that a computer is based on a single processor or single core or also known as CPU-chip, where program instructions are read and get executed. various components like register file, ALU, bus interface, system bus are included inside this single core. Mainly two components i.e register file and ALU are combinedly considered as a single core. In the subject of the architecture of a multi-core processor we can say that a component which is composed of two or more number of independent cores. In a simple way It can be described as an integrated circuit which is integrated with two or more individual processors or core in some sense. This is a new trend in computer architecture. In short this is a replication of multiple processor cores on a single core.The chip microprocessor implementations from the major chip manufacturers each handle the Input-Output bus and the Front Side Bus differently. Chip Microprocessors come in different types like two processors (dual core), four processors (quad core), and eight processors (octa core) configurations. There are several variations in how cache and memory are approached in the new chip microprocessors. Some configurations are multithreaded where some are not. The approaches to, processor to processor communication vary among different implementations. The chip microprocessor implementations from the major chip manufacturers each handle the Input-Output bus and the Front Side Bus differently.

The below figure shows the basic architecture of multicore processor.

according to The IEEE Dictionary of Electrical and Electronics Terms ,a multi processor architecture in which parallel processing can be performed.

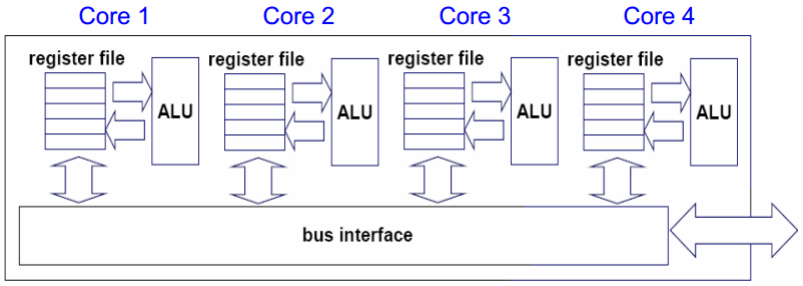

Figure 2.1: Basic Multicore Architecture having 4 cores

It is the job of the programmer, compiler, or operating system to supply the multiprocessor with tasks to keep the processors busy. [2]

2.2 Design Consideration

This section discusses some of the important aspects of the design of parallel computing systems. The design of a parallel computing system requires considering many design options. The designer must choose a basic processor architecture that is capable of performing the contemplated tasks. The processor could be a simple element or it could involve a super scalar processor running a multithreaded operating system.

The processors must communicate among themselves using some form of an interconnection network . This network might prove to be a bottleneck if it cannot support simultaneous communication between arbitrary pairs of processors. Providing the links between processors is like providing physical channels in telecommunications. How data are exchanged must be specifi ed. A bus is the simplest form of interconnection network. Data are exchanged in the form of words, and a system clock informs the processors when data are valid. Nowadays, buses are being replaced by networks - on - chips (NoC) . In this architecture, data are exchanged on the chip in the form of packets and are routed among the chip modules using routers .Data and programs must be stored in some form of memory system , and the designer will then have the option of having several memory modules shared among the processors or of dedicating a memory module to each processor. When processors need to share data, mechanisms have to be devised to allow reading and writing data in the different memory modules. The order of reading and writing will be important to ensure data integrity. When a shared data item is updated by one processor, all other processors must be somehow informed of the change so they use the appropriate data value.

Parallel algorithms and parallel architectures are closely tied together. We cannot think of a parallel algorithm without thinking of the parallel hardware that will support it. Conversely, we cannot think of parallel hardware without thinking of the parallel software that will drive it. Parallelism can be implemented at different levels in a computing system using hardware and software techniques

2.3 Choice of Language And Library

There are numerous Library available supporting almost every known programming languages. I have chosen Pthread and Java Thread Library for programming. Almost all Simulation and program are written using java threads because the only way we can explore functionality of multicore is by using threads.

- **Machine independent :** we can run our application in all machine where JVM is installed. [3]
- **Synchronization :** Synchronization can be achieved easily
- **Memory Management :** Efficient memory management garbage collection by JVM. [4]

2.4 Measuring Benefits

We review in this section some of the important results and benefits of using parallel computing. But first, we identify some of the key parameters that we will be studying in this section

2.4.1 Speed up factor

The potential benefit of parallel computing is typically measured by the time it takes to complete a task on a single processor versus the time it takes to complete the same task on N parallel processors. The speedup S (N) due to the use of N parallel processors is defined by

S(N)=Tp(1) / Tp(N) where Tp(1) is the algorithm processing time on a single processor and Tp(N) is the processing time on the parallel processors. In an ideal situation, for a fully parallelizable algorithm, and when the communication time between processors and memory is neglected , we have Tp(N) = Tp(1)/ N , and the above equation gives

$$S(N)=N$$

It is rare indeed to get this linear increase in computation domain due to several factors, as we shall see Later. [5]

2.4.2 Communication Overhead

For single and parallel computing systems, there is always the need to read data from memory and to write back the results of the computations. Communication with the memory takes time due to the speed mismatch between the processor and the memory.Moreover, for parallel computing systems, there is the need for communication between the processors to exchange data. Such exchange of data involves transferring data or messages across the interconnection network.

1. **Interconnection network delay :** Transmitting data across the interconnection network suffers from bit propagation delay, message/data transmission delay, and queuing delay within the network. These factors depend on the network topology, the size of the data being sent, the speed of operation of the network, and so on.

2. **Memory bandwidth :** No matter how large the memory capacity is, access to memory contents is done using a single port that moves one word in or out of the memory at any give memory access cycle.

3. **Memory collisions :** where two or more processors attempt to access the same memory module. Arbitration must be provided to allow one processor to access the memory at any given time.

4. **Memory wall :** The speed of data transfer to and from the memory is much slower than processing speed. This problem is being solved using memory hierarchy such as
register $\leftrightarrow Cache \leftrightarrow RAM \leftrightarrow ElectronicDisk \leftrightarrow MagneticDisk \leftrightarrow OpticalDisk$.
Different Delays Have been Shown in figure 2.2.

2.4.3 Estimating communication Overhead

Let us assume we have a parallel algorithm consisting of N independent tasks that can be executed either on a single processor or on N processors. Under these ideal circumstances, data travel between the processors and the memory, and there is no inter processor communication due to the task independence. We can write under ideal circumstances

Operation	Symbol	Comment
Memory read	$T_r(N)$	Read data from memory shared by N processors
Memory write	$T_w(N)$	Write data from memory shared by N processors
Communicate	$T_c(N)$	Communication delay between a pair of processors when there are N processors in the system
Process data	$T_p(N)$	Delay to process the algorithm using N parallel processors

Figure 2.2: This figure shows different delays

T p(1)= N τp

T p(N)= τp

The time needed to input the data by a single processor is given by
T r(N)= τp

The time needed by the parallel processor to read data from memory is given by

T r(1)= $\alpha Tr(1) = \alpha N \tau m$

Where alpha(α) = factor that account the limitation of accessing the shared memory. $\alpha = 1/\mathrm{N}$ when each processor maintains its own copuy of data. $\alpha = 1$ when data are distributed to each task in order from a central memory. In the worst case , we could have α ¿ N When all processor request data and collide with each other we could conclude from the above concept as

$$T_r(N)\begin{cases} = & \tau_m & \text{when} & \text{Distributed memory} \\ = & N\tau_m & \text{when} & \text{Shared memory and no collisions} \\ > & N\tau_m & \text{when} & \text{Shared memory with collisions.} \end{cases}$$

Writing back the results to the memory, also, might involve memory collisions when the processor attempts to access the same memory module
Tw(1)=N τm

Tw(N)=$\alpha Tw(1)=\alpha N \tau m$

For a single processor, the total time to complete a task, including memory access overhead, is given by

Ttotal(1)=Tr(1)+Tp(1)+Tw(1)

Now let us consider the speedup factor when communication overhead is considered

Ttotal(N)=Tr(N)+Tp(N)+Tw(N)

$$= 2\mathrm{N}\alpha\tau m + \tau p$$

so , Total Speed up factor is given by

S(N)=Ttotal(1)/Ttotal(N)

Memory mismatrch ratio is given by R= $\tau m/\tau p$

2.4.4 Amdahl's law

Assume an algorithm or a task is composed of parallizable fraction f and a serial fraction 1-f . Assume the time needed to process this task on one single processor is given by

$$Tp(1) = N(1-f)\tau p + Nf\tau p = N\tau p$$

where the first term on the right-hand side (RHS) is the time the processor needs to process the serial part. The second term on RHS is the time the processor needs to process the parallel part. When this task is executed on N parallel processors, the time taken will be given by

$$Tp(N) = N(1-f)\tau p + f\tau p$$

where the only speedup is because the parallel part now is distributed over N processors. Amdahl s law for speedup S (N), achieved by using N processors, is given by

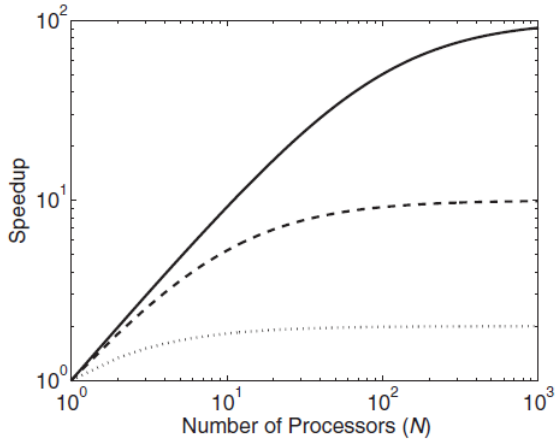

Figure 2.3: Amdhals law

$$S(N) = \frac{T_p(1)}{T_p(N)} = \frac{N}{(1-f)N + f} = \frac{1}{(1-f) + f/N}.$$

To get any speedup, we must have 1-f <<f/N
this shows that f approaches to unity when N is very large.i.e no of processor is large. figure 3 shows the speedup versus f for different values of N . The solid line is for f = 0.99; the dashed line is for f = 0.9; and the dotted line is for f = 0.5. We note from the figure that speedup is affected by the value of f . As expected, larger f results in more speedup. However, note that the speedup is most pronounced when f >0.5. Another observation is that speedup saturates to a given value when N becomes large.

and it can be concluded that

S(N)=0 if f=0 Completely serial code
S(N)=1 when f=1 Completely parallel code

2.4.5 Gustafason's Law

The predictions of speedup according to Amdahl s law are pessimistic. Gustafson [6]*madetheobservationthatparallelismincreasesinanapplicationwhentheproblemsize*

To derive Gustafson Barsis formula for speedup, we start with the N parallel processors fi rst. The time taken to process the task on N processors is given by

Tp(N)=(1-f)$\tau p + fp = \tau p$

To derive Gustafson Barsis formula for speedup, we start with the N parallel processors fi rst. The time taken to process the task on N processors is given by

Tp(1)=(1-f)$\tau p + Nf\tau p$

The speedup is given now by

S(N)=Tp(1)/Tp(N)

S(N)=(1-f)+Nf

$S(N) = 1 + (N - 1)f$

figure shows the speedup versus f for different values of N . The solid line is for f = 0.99; the dashed line is for f = 0.9; and the dotted line is for f = 0.5. Notice that there is speedup even for very small values of f and the situation improves as N gets larger.

To get any speedup we must have f(N-1) >>1

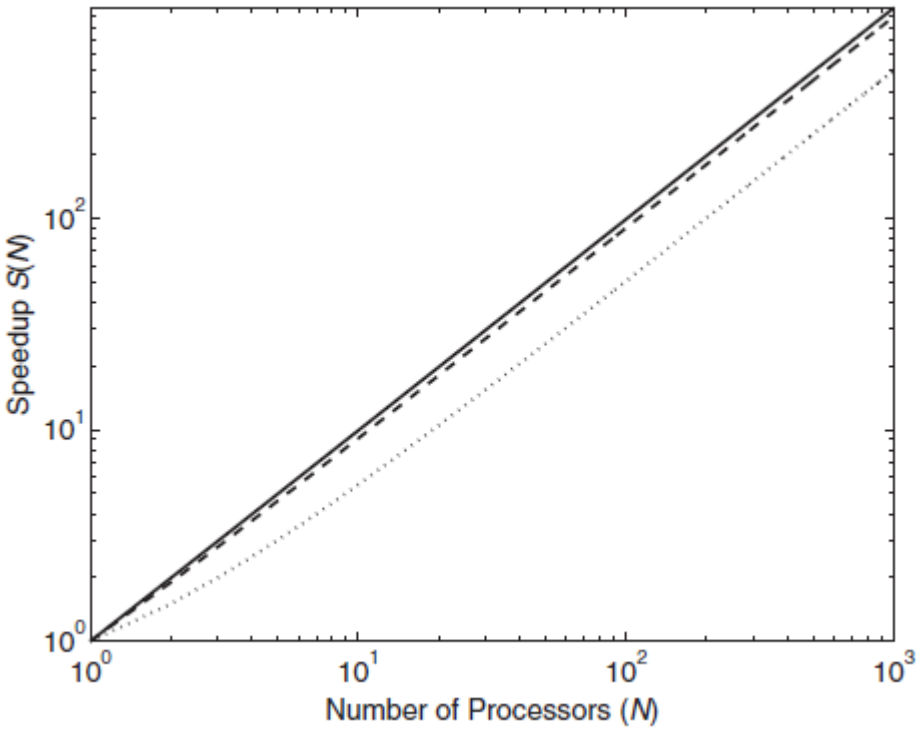

Figure 2.4: Gustafason's Law

Chapter 3

Implementation and simulations

3.1 Fast Fourier Transformation

3.1.1 Introduction

The Discrete Fourier Transform (DFT) is one of the most widely used algorithms in the fields of science and engineering, especially in the field of signal processing. For many practical applications, it is important to have an implementation of DFT that is as fast as possible. Since 1965, various algorithms have been proposed for computing DFTs efficiently. One such algorithm is the Fast Fourier Transform (FFT) which produces the same result as the DFT approaches, yet it is more efficient, often reducing the computation time by factors of hundred. The FFT has been studied extensively as a frequency analysis tool in diverse applications such as image processing, MPEG audio coding, linear filtering, correlation analysis and spectrum analysis. FFT is both computation-intensive and memory-intensive due to the large amount of data involved in the underlying applications, so an efficient algorithm must be developed for splitting up the input data among multiple cores so that the efficiency can be improved when compared to that on a single core. In this paper, we present an algorithm for parallelizing the computation of FFT over a multicore architecture and put forth the way in which the cores would communicate with each other in each round on multicore architectures. Our design of the parallel algorithm is focused on efficient data distribution. The algorithm divides the given inputs equally among the cores that are available. This ensures that the cores are all engaged in FFT computation till their input chunks are processed. Also, a detailed analysis regarding the number of cores that will be necessary for the computation of FFT in each

stage of the calculation has been performed. The results are compared for different values to observe the optimal number of cores required for different number of inputs so that many cores do not remain idle. With our proposed algorithm, the cores do not need a master to calculate their core ids and to calculate which other core to communicate with in the ith round. Instead the cores calculate these themselves and continue with the calculation of FFT algorithm with their input chunks. The rest of the Section is organised as follows. In Section 2 we present the necessary background and the related works for this paper. Section 3 deals with the system model, assumptions regard- ing the system behavior, our proposed algorithm and an illustration. Performance complexity and communication overhead between the cores are discussed. In the Section 4 we present the core utilization formula derived and the simulated graphs.

3.1.2 Background

Let x0,x1,.....xn be complex numbers.The DFT is Defined By the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

In the above equation xn is the input sequence of data of length N. Basically, the computational problem for the DFT is to compute the sequence Xk of N complex-valued numbers given an input sequence of data xn.

The radix-2 Cooley-Tukey algorithm for calculating FFT uses the divide and conquer approach, i.e., decomposing and breaking the transform into smaller transform and combining them to give the total transform [7, 8]. This is because a series of smaller problems is easier to solve than one large one. The decomposition is reordering of the samples in the signal. Then the initial FFT decomposition will be as shown in Figure 3.1 [9] . Table shows the rearrangement pattern. We can observe from Table that the rearranged input indices are in bit-reversed order of the original input indices.

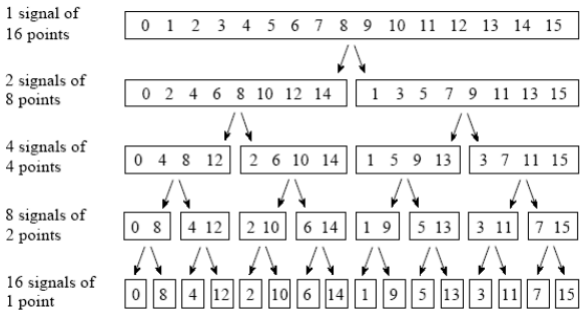

Figure 3.1: FFT Decomposition

Normal order		Bit reversed order	
Decimal	Binary	Decimal	Binary
0	0	0	0
1	1	8	1000
2	10	4	100
3	11	12	1100
4	100	2	10
5	101	10	1010
6	110	6	110
7	111	14	1110
8	1000	1	1
9	1001	9	1001
10	1010	5	101
11	1011	13	1101
12	1100	3	11
13	1101	11	1011
14	1110	7	111
15	1111	15	1111

Table 1: Bit Reversal Order

The algorithm divides this N-point data sequence into two N/2-point data sequences f1(n) and f2(n), corresponding to the even-indexed and odd-indexed points of x(n) respectively. Then the N -point DFT can be computed as,

$$X(k) = F_1(k) + w_N^k F_2(k), 0 \leq k \leq \frac{N}{2} - 1$$

$$X(k + \frac{N}{2}) = F_1(k) - w_N^k F_2(k), 0 \leq k \leq \frac{N}{2} - 1$$

where Wn^k are twiddle factors [9] (twiddle factors refers to the root-of-unity complex multiplicative constants that are multiplied by the data in the course of the algorithm), F1(k) and F2(k) are the N/2-point DFTs of f1(n) and f2(n), respectively. The sub- problems F1(k) and F2(k) are recursively solved to obtain the final solution of the original problem [10, 11].

3.1.3 Algorithm

The System model we propose assumes a multicore architecture with the cores having shared memory through which data can be exchanged between any two cores. The system takes N input points of FFT and calculates the output samples. In this paper we focus on the most common FFT algorithm, the radix-2 Cooley-Tukey DIT FFT algorithm which means the number of output points can be expressed as a power of 2. In this paper we propose an algorithm for implementing the FFT algorithm on a multicore architecture. Our entire algorithm is divided as Algorithm 1 and Algorithm 2. Algorithm 1 describes the way input data should be split among multiple cores, how a core id can be obtained and when a core should communicate with the other core. Algorithm 2 describes the way to identify the core to communicate with and get the results. For reducing the complexity in our system we assume that the number of cores is also a power of 2. Because of this, even if the number of cores available are not powers of 2, we choose the nearest power of 2 and then proceed with our proposed algorithm. As a result of this, we will be able to divide the number of input samples equally among all the available cores.

Description Of Pseudocode

In this section we describe our proposed algorithm for implementing FFT on multicore architectures. Our entire algorithm is divided as Algorithm 1 and Algorithm 2.Algoriyhm 1 is shown in figure 3.2 and Algorithm 2 is represented in Figure 3.3.For parallelizing the FFT, we propose the algorithm as given in the pseudocode Algorithm 1.We take N number of input points and P number of cores and give each core equal chunks of data of size S = N/P. Each core calculates the number of bits that will be used to represent the input indices (n) and the number of bits used to represent the core ids (p). Using the input sequence, each core calculates its own unique core id by taking the last p bits of the input indices that it gets.
The N point signal is decomposed into N/2 points in each stage until there are N signals each containing a single point. An interlaced decomposition is used each time a signal is broken into two, that is, the signal is separated

Algorithm 1 Parallelize FFT

```
1: for i = 0 : P − 1, in each core P_i do
2:    n = log2N & p = log2P
3:    S = N/P
4:    N_i[S] = get_inputs(i)
5:    core_id_i = get_ coreid(N_i[S])
6:    for w = 2 : N do
7:       if W ⩽ S then
8:          get_ results(core_id_i)
9:          FFT(W)
10:      end if
11:      if W > S then
12:         get_ results(core_id_i)
13:         if core_id_j > core_id_i then
14:            In core_id_i
15:            FFT(W)
16:         else
17:            In core_id_j
18:            FFT(W)
19:         end if
20:      end if
21:   end for
22: end for
```

Figure 3.2: Algorithm 1(part 1) For FFT

into its even and odd numbered samples.
Figure 2 shows how the input samples are divided among given number of cores and how

butterfly diagram is calculated at each stage. Initially width W will be 2 and the width increases by a factor 2 for each stage. Each core calculates the FFT of the input points recursively till it computes the S point FFT. Till this point the FFT window (W) is less than S and no communication with another core is required as each core will have sufficient input samples to calculate the W point FFT. After this the FFT window W becomes greater than S and the cores need to communicate with other cores for getting the previously computed results of the W/2 point FFT from other cores for further computing the W point FFT. This process continues until the width of the FFT window (W) equals the number of input samples (N) i.e., until N point FFT is calculated
When W becomes greater than S, the algorithm to identify the core with which communication has to be done is shown in the pseudo code Algorithm 2. According to the algorithm, each core gets the results from the other core that differs by i^{th} bit in the i^{th} round. This can be formulated by XOR the core id with the binary equivalent of $2^{(n-i)*(n-1)}$ which gives the core id that

Algorithm 2 Identify the core to communicate with

```
get_ results():
round, i ← 0
for every W = 2 : N do
    if W ⩽ S then
        return(core_id_i)
    end if
    if W > S then
        i = i + 1
        core_id_j = core_id_i XOR 2^((n−i)∗(n−1))
        return(core_id_j)
    end if
end for
```

Figure 3.3: Algorithm 2(part 2) for FFT

differs by one bit in the i^{th} place for the i^{th} round. Here, it is observed that after each core computes the S point FFT, at each round i, starting from 1, the core communicates with another core whose core id differs by one bit in the i^{th} position from the left. And after the W point FFT is calculated, the core whose id is smaller of the two stores the results.

Illustration

In order to understand the proposed parallel algorithms let us consider an example where N = 16 and P = 4. Hence n = 4 and p = 2 where n is the number of bits required to represent the given input samples in binary form and p is the number of bits required to represent the core id.
Then the initial FFT decomposition will be as shown in Figure 3.1. The N point signal is decomposed into N/2 points in each stage until there are N signals each containing a single point.
Now the input to our proposed algorithm is the bit-reversed input sequence. The size of input chunks given to each core will be S = 4(S = N/P), i.e., each core will be given four bit-reversed input data samples. So in our example, core 0 will get the input data samples located in the indices 0, 8, 4, 12, core 1 will get the input samples located in indices 2, 10, 6, 14, core 2 will get the samples in the indices 1, 9, 5, 13 and core 3 will get the input samples located in indices 3, 11, 7, 15. Now according to Algorithm 1, for obtaining unique core ids, we take last p bits of the n bit input as core id. Therefore for the example taken, the core ids are 00, 10, 01, 11 for the cores 0, 1, 2, 3 respectively.
Figure 2 shows how the input samples are divided among given number of cores and how the butterfly diagram is calculated at each stage. We can see

from figure that intitially the width will be W = 2, so no communication is required with the other core until W¿=4. Each core will have four input samples and intitally each core will take two input samples (W = 2) and calculate a 2 point FFT. Then two 2-point FFTs are taken and a 4-point FFT is calculated.

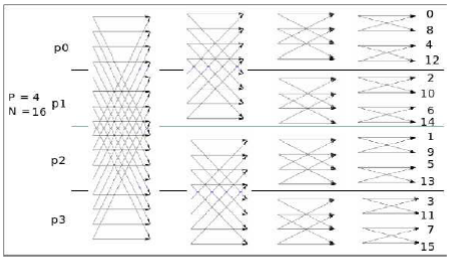

Figure 2: Input divided among the given cores and butterfly diagram calculated at each stage

At this stage W = 4, and no communication between cores is required till this point as each core contains four input samples.So no communication with other cores occurs for the following input pairs: (0, 8), (4, 12), (2, 10), (6, 14), (1, 9), (5, 13), (3, 11), (7, 15), (0, 4), (2, 6), (1, 5), (3, 7). When W ¿ S, in round 1, each core communicates with the corresponding core which has a one bit change in the first place in order to calculate an 8-point FFT. Therefore, communication between processors is given by

Round 1 :

After the 8-point FFT is calculated by the cores,the result is stored in the cores with lowest coreid, i.e. in our example result will be stored in the cores having coreids coreid0(00) and coreid2(01) .so in the round 2,of the first pair one core (with lowest value) is chosen and it communicate with a core having a one bit change in the second place to calculate the 16 point FFT.

Round 2 :

00 (coreid0) ¡-¿ 01 (coreid2)

Figure 3.3 Shows the communication between the cores at each stage for the example taken,till a 16 point FFT is calculated.

3.1.4 Performance Complexity

Algorithm 1 uses the FFT algorithm to calculate N point FFT. Hence from the formula of FFT , there are N/2*log2N complex multiplications and N*log2N complex additions. The reads and writes into the shared memory can be given by the formula where s = log2S . This is derived from the knowledge that the cores start communicating with the other cores only after their input chunk is processed.

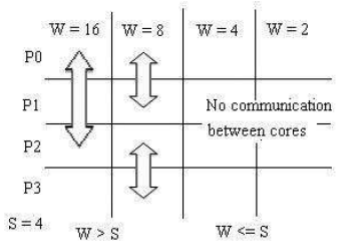

Figure 3: Communication between cores at each stage

Figure 3.4: Communication Betwwen cores

3.1.5 Communication Overhead between cores

The algorithm we propose is focused on efficient data distribution. In our proposed Algorithm 1, based on the number of cores available for calculating the N point FFT, we divide the input points equally among all the available cores.

The reason for this approach is due to the fact that FFT algorithm has both sequential and parallelizable parts. For example, in order to calculate a 8 point FFT, first four 2-point FFTs, then two 4-point FFTs and finally one 8-point FFT has to be calculated. The step of calculating the four 2-point FFTs is independent and can be parallelized. But the calculation of two 4-point FFTs can be carried out only after calculating the all the 2-point FFTs. And finally 8 point FFT can be calculated only after the 4-point FFT computation is performed. These parts of the algorithm are the serial execution parts. Such points in the algorithm can also be called as the synchronization points . If the input data is not divided equally among all the cores, then the waiting time at the synchronization points will become larger than the total computation time. In our algorithm, based on the number of cores, we divide the input samples evenly so that each core takes the same amount of computation time, there by the waiting time at the synchronization points will become negligible when compared to the computation time.

Algorithm 2 gives the pseudocode for each core to find out which core to get the value from, for calculating the next round of FFT. Hence, there is no necessity for each core to wait for another core to inform about the corresponding core to communicate with in each round. This reduces the communication overhead that is faced while the master core communicates with the other cores to inform the same.

3.1.6 Analysis Of Core Utilization

The maximum number of processors that will be needed for the calculation of an N-point FFT will be N/2. But if we are given more number of cores than the size of input samples, then the core utilization decreases. In contrast if we have more of input samples for which FFT has to be calculated and less number of cores , then the core utilization increases. However at each stage of the butterfly diagram the number of cores utilized keep decreasing by a factor P/2 and in the last stage only one core will be utilized to calculate the final N-point FFT. Table 2 shows the core utilization for different input pairs corresponding to different number of cores.
For calculating the overall core utilization, we have to consider two cases. One where the number of cores are greater than or equal to half of the input samples and the other case where the number of cores are less than half of the input samples. We consider half of the input samples because the maximum number of cores that will be needed for calculating an N point FFT is N/2. For deriving an overall core utilization we sum up the number of cores being utilized in each stage and compare it with the total number of cores available until the algorithm completes.
In both cases the number of cores that will be utilized will be same; its only the total number of cores that differs. The number of cores used in every round will be decreased by a factor of 2 i.e., in round 1 where there is no communication with other cores , all the cores will be utilized and in last round only one core will be utilized to calculate the final point FFT. We can also observe that until there is no communication with other core (i.e., when the cores compute the input samples given to them), all the cores will be utilized. When W becomes greater than S (number of input samples given to each core), then the cores will communicate with other cores and from there by the number of cores utilized will be reduced at each round. So the total number of cores that will be utilized is given by the formula:

$$\sum_{i=0}^{n-1} \frac{N/2}{2^i}$$

In Case 1, where the number of cores are less than half of the input samples, each core will have more number of input samples (more than 2). So it means there will be more number of rounds where the cores compute without having to communicate with the other cores. In this case, we have to find out the number of rounds where there is no communication with other cores and number of rounds where there is communication. Then multiply each of these with the number of cores i.e., P. Hence the formula in this case for the total number of available cores is given as:

$$(\frac{N}{P} - 1)P + p * P$$

where(N/P - 1) are the number of rounds where there is no communication with the other core and p is the number of rounds where there is communication with other cores.
In Case 2, where the number of cores are greater than or equal to half of the input samples, for finding out the total number of cores available, we first have to find out the number of rounds that algorithm takes to compute the final point FFT and then multiply with the number of cores i.e., P. Hence the formula in this case for the total number of available cores is given by n*P.
Based on all the above details, the overall core utilization can be derived using the below formulae:
Case 1 : P <N/2

$$\eta = \frac{\sum_{i=0}^{n-1} \frac{N/2}{2^i}}{(\frac{N}{P} - 1)P + p * P}$$

Case 2 : P $\geq N/2$

$$\eta = \frac{\sum_{i=0}^{n-1} \frac{N/2}{2^i}}{n * P}$$

Figure 4 shows the core utilization for different input samples given P (number of cores) as 8, 32 and 256 respectively.
From Table 2 and Figure 4, we can deduce that when the number of input samples is less, then lower number of cores gives a better core utilization. But as the number of input samples increases, the core utilization increases significantly with the number of cores.

3.1.7 Conclusion

Assuming F, the percentage of code that can be parallelized to be 95, the maximum speedup that can be achieved by using N processors (for whatever be the value of N) cannot exceed 7.8 times the speed of the program using a single core.

Table Showing Core Utilization

No Of Input

P	N								
No Of Processors	4	8	16	32	64	128	256	512	1024
8	0.187	0.291	0.468	0.645	0.787	0.881	0.937	0.967	0.983
16	0.093	0.145	0.234	0.387	0.562	0.721	0.838	0.912	0.954
32	0.046	0.072	0.117	0.193	0.328	0.496	0.664	0.798	0.888
64	0.023	0.036	0.058	0.096	0.164	0.283	0.442	0.614	0.761
128	0.011	0.018	0.029	0.048	0.082	0.141	0.249	0.399	0.570
256	0.005	0.009	0.014	0.024	0.041	0.070	0.124	0.221	0.363

Figure 3.5: Table showing data for FFT implementation

Comparison of core utilization

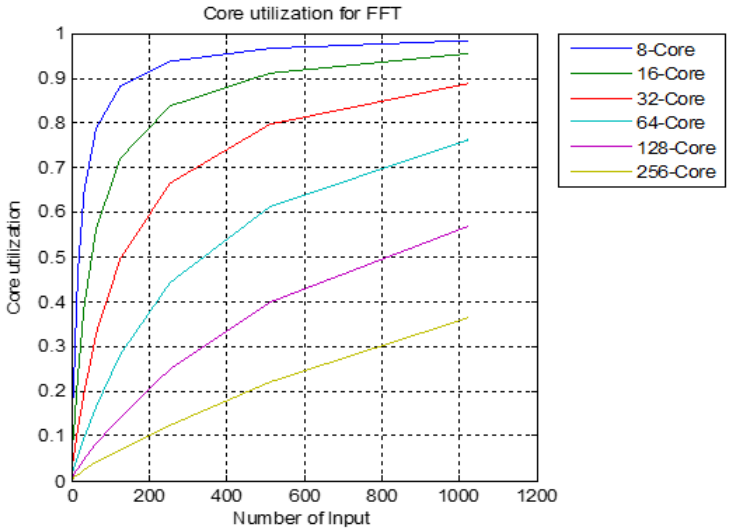

Figure 3.6: Graph Showing Core Utilization For FFT

3.2 Sorting On GPU

3.2.1 Introduction

Sorting is one of the most widely researched subjects. In this hugely loaded data world we need to search things in a quick succession. So in order to speed up the operation on thousands or millions of records during a search operation we need some optimized way or path or concept which is called as sorting. Sort is a fundamental central thing used in many database operations. The main purpose of sorting concept is to optimize its usefulness for some particular tasks to get the dataset arranged in an increasing or decreasing order. So that dataset can be categorized easily. Sorting is a computational building square of major significance and is a standout amongst the most broadly considered algorithmic issues. The imperativeness of sorting has likewise prompt the outline of efcient sorting calculations for an assortment of parallel architectures. Numerous calculations depend on the accessibility of efcient sorting schedules as a premise for their own particular efciency, and numerous different calculations might be helpfully stated as far as sorting. Database frameworks make far reaching utilization of sorting operations. The development of spatial information structures that are crucial in machine design and geographic data frameworks is in a general sense a sorting methodology. Efcient sort schedules are additionally a valuable building square in actualizing calculations like inadequate framework augmentation and parallel programming examples like Mapreduce. It is in this manner critical to give efcient sorting schedules on essentially any programming stage, and as machine architectures develop there is a proceeding need to investigate efcient sorting methods on rising architectures.

One of the overwhelming patterns in chip construction modeling as of late has been constantly expanding chip-level parallelism. Multicore Cpusgiving 24 scalar centers, ordinarily expanded with vector unitsare presently mundane and there is much evidence that the pattern towards expanding parallelism will proceed towards "many-core" chips that give far higher degrees of parallelism. Gpus have been advanced by distributing multicore CPU sort running on a 4-center 3.22 Ghz Intel Q9550 processor[12].

At the heading edge of this drive towards expanded chip- level parallelism for quite a while and are as of now generally manycore processors. Current NVIDIA Gpus, for instance, hold up to 240 scalar transforming components for every chip [9], and as opposed to prior eras of Gpus, they might be modified straightforwardly in C utilizing CUDA. In this paper, we depict

the outline of efcient sorting calculations for such manycore Gpus utilizing CUDA. The programming exibility gave by CUDA and the current era of Gpus permits us to think about a much more extensive reach of algorithmic decisions than were advantageous on past eras of Gpus. We specically concentrate on two classes of sorting calculations: a radix sort that straightforwardly controls the twofold representation of keys and a consolidation sort that requires just a correlation work on. The GPU is a massively multithreaded processor which can support, and indeed expects, several thousand concurrent threads. Exposing large amounts of ne-grained parallelism is critical for efcient algorithm design on such architectures. In radix sort, we exploit the inherent ne-grained parallelism of the algorithm by building our algorithm upon efcient parallel scan operations. We expose ne-grained parallelism in merge sort by developing an algorithm for pairwise parallel merging of sorted sequences, adapting schemes based on parallel splitting and binary search previously described in the literature. We demonstrate how to impose a block-wise structure on the sorting algorithms, allowing us to exploit the fast on-chip memory provided by the GPU architecture. We also use this on-chip memory for locally ordering data to improve coherence, thus resulting in substantially better bandwidth utilization for the scatter steps used by radix sort. Our experimental results demonstrate that our radix sort algorithm is faster than all previously published GPU sorting techniques when running on current-generation NVIDIA GPUs. Our tests further demonstrate that our merge sort algorithm is the fastest comparison-based GPU sort algorithm described in the literature, and is faster in several cases than other GPU-based radix sort implementations. Finally, we demonstrate that our radix sort is highly competitive with multicore CPU implementations, being up to 4.1 times faster than comparable routines on an 8-core 2.33 GHz Intel E5345 system and on average 23 percentage faster than the most[13,14].

Here we are doing some comparison test for efficient implementation of sorting algorithm in CPU cores and GPU cores, where we got the conclusion that GPU cores are more efficient than conventional way in CPU as they take less time to complete the sorting process successfully.

3.2.2 CPU Vs. GPU

Comparison is based on how they process and what is their performance. In a general way we can say that a CPU consists of a few number of cores grouped together for sequential serial processing, while a GPU is consists of thousands of smaller and more efficient cores designed for handling multiple

tasks simultaneously. We can implement sorting algorithm in a conventional way ,so we have to concentrate on GPU implementation. Before the implementation part, lets discuss something about GPU in a quick succession [16]

GPU

A representation transforming unit (GPU), likewise sporadically called visual handling unit (VPU), is a particular electronic circuit intended to quickly control and adjust memory to quicken the production of pictures, in a casing cradle proposed for yield to showcase. Gpus are utilized within inserted frameworks, cell telephones, PCs, workstations. Current GPUs are extremely effective at controlling machine illustrations, and their profoundly parallel structure makes them more successful than broadly useful-CPUs for calculations where handling of vast pieces of information is carried out in parallel. GPU purpose or accelerate computing is based on the use of a GPU together with a CPU to accelerate scientific, engineering, enterprise applications. In this paper, we show a quick and efcient GPU string memory development that enormously enhances the execution. Conversely, an union sort approach dully stacks each one string from the worldwide memory (at whatever point ties happen) in every consolidation step. We demonstrate a velocity up of more than 10 over the best GPU string sorting methodologies. We acquire a sorting throughput of 83 Mkeys/s on a dataset of 1 million arbitrary strings. Our methodology can scale to a request of bigger info size than the reported GPU string sort. On a 10 million words dataset we attain a throughput of 65 Mkeys

If question arises that how GPU is working, then here its answer. GPU quickening processing offers extraordinary provision of execution by offloading process serious bit of the requisition to the GPU, while the rest of the code still runs on the CPU. From users' point of view requisition basically runs fundamentally[15]

3.2.3 Architecture

Each GPU is made up of SM(streaming) multi processors that is streaming multi processors. The SM multiprocessors again consists of 8 scalar processors. Those SM processors only get one instruction at a time which means 8 scalar processors will execute the very instruction. And this is processed through a wrap i.e 32 threads. Considering the whole GPU to be a couple of SIMD (single instruction multiple data) units. NVIDIA calls it SIMT (single instruction multiple thread).

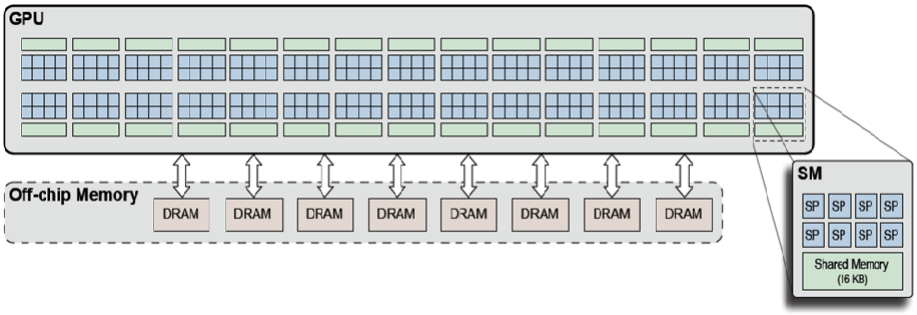

Figure 3.7: Basic GPU Architecture

When examining the configuration of our sorting calculations, we briey audit the remarkable subtle elements of NVIDIA's present GPU building design and the CUDA parallel programming model. As their name infers, Gpus (Graphics Processing Units) happened as quickening agents for representation provisions, prevalently those utilizing the Opengl and Directx programming interfaces. Because of the huge parallelism inalienable in design, Gpus have long been enormously parallel machines. In spite of the fact that they were initially simply xed-capacity gadgets, Gpus have quickly advanced into progressively exible programmable professional- cessors. Cutting edge NVIDIA Gpusstarting with the Geforce 8800 GTXare completely programmable manycore chips manufactured around a cluster of parallel processors [9], as represented in Figure 1. The GPU comprises of a show of SM multiprocessors, each of which is equipped for supporting up to 1024 co-occupant simultaneous strings. NVIDIA's present items extend in size from 1 SM at the low end to 30 Sms at the high end. A solitary SM demonstrated in Figure 1 holds 8 scalar SP processors, each with 1024 32-bit registers, for what added up to 64kb of register space for every SM. Every SM is likewise outfitted with a 16kb on-chip memory that has low get to idleness and high transfer speed, like a L1 cache. All string administration, including creation, booking, and hindrance synchronization is performed altogether in fittings by the SM with basically zero overhead. To efciently deal with its expansive string populace, the SM utilizes a SIMT (Single Instruction, Multiple Thread) structural engineering. Strings are executed in gatherings of 32 called twists. The strings of a twist are executed on partitioned scalar processors which impart a solitary multithreaded guideline unit. The SM transparently deals with any dissimilarity in the execution of strings in a twist. This SIMT structural planning permits the equipment to attain generous efciencies while executing nondisparate information- parallel the CUDA parallel programming model. As their name infers, Gpus (Graphics Processing Units) happened as quickening

agents for representation provisions, prevalently those utilizing the Opengl and Directx programming interfaces. Because of the huge parallelism inalienable in design, Gpus have long been enormously parallel machines. In spite of the fact that they were initially simply xed-capacity gadgets, Gpus have quickly advanced into progressively exible programmable professionalcessors. Cutting edge NVIDIA Gpusstarting with the Geforce 8800 GTXare completely programmable manycore chips manufactured around a cluster of parallel processors [9], as represented in Figure 1. The GPU comprises of a show of SM multiprocessors, each of which is equipped for supporting up to 1024 co-occupant simultaneous strings. NVIDIA's present items extend in size from 1 SM at the low end to 30 Sms at the high end. A solitary SM demonstrated in Figure 1 holds 8 scalar SP processors, each with 1024 32-bit registers, for what added up to 64kb of register space for every SM. Every SM is likewise outfitted with a 16kb on-chip memory that has low get to idleness and high transfer speed, like a L1 cache.

All string administration, including creation, booking, and hindrance synchronization is performed altogether in fittings by the SM with basically zero overhead. To efciently deal with its expansive string populace, the SM utilizes a SIMT (Single Instruction, Multiple Thread) structural engineering. Strings are executed in gatherings of 32 called twists. The strings of a twist are executed on partitioned scalar processors which impart a solitary multithreaded guideline unit. The SM transparently deals with any dissimilarity in the execution of strings in a twist. This SIMT structural planning permits the equipment to attain generous efciencies while executing non-disparate information- parallel codes. CUDA gives the intends to designers to execute parallel projects on the GPU. In the CUDA expert- gramming model, a provision is composed into a successive host program that may execute parallel projects, alluded to as parts, on a parallel gadget. Commonly, the host project executes on the CPU and the parallel bits execute on the GPU, despite the fact that CUDA parts might likewise be ordered for efcient execution on multicore Cpus. A piece is a SPMD-style (Single Program, Multiple Data) processing, executing a scalar successive program over a set of parallel strings. The developer arranges these strings into string obstructs; a part in this manner comprises of a framework of one or more squares. A string piece is a gathering of simultaneous strings that can chip in around themselves through obstruction syn- chronization and a for every-square imparted memory space private to that square. At the point when summoning a bit, the developer species both the amount of pieces and the amount of strings for every square to be made when propelling the bit. The string pieces of a

CUDA part basically virtualize the SM multiprocessors of the physical GPU. We can think about each one string piece as a virtual multiprocessor where each one string has a xed register foot shaped impression and each one square has a xed distribution of for every-square shared memory.

3.2.4 Conclusion

Through our research we have concluded that GPU has several aspects that operate better than the CPU that are stated below categorically

- With graphics processor technology, CPU systems can more efficiently perform during complex work that does not need to be done in sequence. The CPU takes any sequential tasks in the application and executes them in a typical serial fashion while the GPU runs all of the computationally-intensive work.
- Due to their established commercial use, they are readily available and relatively inexpensive compared to the number of multi-core CPUs we would have needed to achieve a similar number of cores.
- GPUs are effective at performing rapid calculations due to their pipeline, which allows up to 320 calculations at the same time on a single GPU. The CPU can do parallel computations, but this is limited by the number of cores available. GPUs also have a larger bus width which makes their memory faster.
- It is becoming increasingly common to use a general purpose graphics processing unit as a modified form of stream processor. This concept turns the massive computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphical operations. In certain applications requiring massive vector operations, this can yield several orders of magnitude higher performance than a conventional CPU.

We have implemented Radix sort on the GPU and cross checked the results with the running time on a CPU and noted that the GPU indeed performs the operation manifolds faster. The graph showing the comparison of clock cycles taken is presented below.

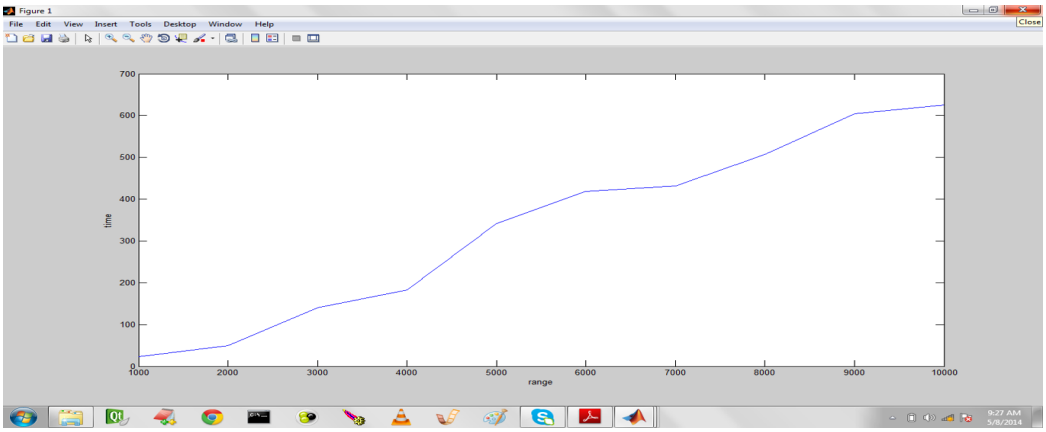

Figure 3.8: Graph For Sorting on GPU

References

[1] L. Oliker M. Wehner and J. Shalf. A real cloud computer. *IEEE Spectrum*, 46(10):24 29, 2009.

[2] Terms Standards Coordinating Committee 10 and Definitions. The ieee standard dictionary ofelectrical and electronics terms. *IEEE Spectrum*, 1996.

[3] Yu Lin Vilas Jagannath, Matt Kirn and Darko Marinov. Evaluating machine-independent metrics for state-space exploration. *Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[4] Oracle Inc. The java virtual machine specification. 7th Edition, 2013.

[5] Fayez Gabeli. Algorithm and parallel computing. *Willey Publication*, 1st Edition:16–22, 2011.

[6] J.L. Gustafson . Reevaluating amdahl s law . *Communications of the ACM,*, pp:532–533, 1988.

[7] S. Casselman. Fpga-based hardware acceleration of c/c++ based applications. *http://www.pldesignline.com/howto/201201188 (1997).,*.

[8] G. Angelopoulos and Pitas. algorithms on a hypercube. in proc. parallel computing action,. *Workshop ISPRA (1990).*

[9] Lewis P. Cooley, J. and P Welch. The fast fourier transform and its application to time series analysis. *John Wiley and Sons, New York,*, pp, 1977.

[10] Voronenko Y. Franchetti, F. and M. Puschel. Fft program generation for shared memory: Smp and multicore. *In Proceedings Of The ACM/IEEE 2006 Conference on Supercomputing,*, ACM Press.

[11] L. Johnsson and D. Cohen. Computational arrays for the discrete fourier transform.technical report caltechcstr:1981.4168-tr-81. *Institute of Technology*, 1981.