Performance Analysis of Parallel Bucket Sort

H.I.S.Wijayabandara

2018

Performance Analysis of Parallel Bucket Sort

A dissertation submitted for the Degree of Master of Computer Science

H.I.S. Wijayabandara
University of Colombo School of Computing
2018

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: HIS Wijayabandara

Registration Number: 2013/MCS/075

Index Number: 13440757

_____

Signature: Date: 13/07/2018

This is to certify that this thesis is based on the work of

Mr./Ms. HIS Wijayabandara

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Mr M.K. Silva

_____

Signature: Date:

Abstract

One of the most fascinating problems in computer science is sorting. Starting from small scale applications, Sorting algorithms are used in variety of applications such as large scale databases and large scale search engines.

Thus, optimizing the sorting algorithms has a big advantage. Bucket sort is one of the most popular comparison based integer sorting algorithm.

Optimizing the bucket sort is an interesting research area. Threads, SSE instructions and GPU programing is used to optimize the algorithm. GPUs (graphics processing units) are becoming an attractive computing platform not only for traditional graphics computation but also for general-purpose computation, because of the computational power, programming capabilities and their comparatively low cost modern GPUs have. This improvement of GPUs with highly parallel programming environments such as CUDA has led to a variety of complex general purpose applications with remarkable performance improvements.

Algorithm is implemented and tested on windows platform. Visual studio is used to do the implementation. Two methods are used to do the profiling of the algorithm. Own profiler is created using windows timer and Intel Inspector is used as a third party tool for profiling. CUDA was used to for do the General Purpose Computing on Graphics Processing Unit(GPGPU).

Results shown that how the algorithm works with different kind of parallelization. For small number of elements, it is better to do the sorting in single thread. Increasing the number of threads doesn't give any positive results for the optimization. Bottleneck of GPU and CPU is shown on this research clearly. CPU base parallelization is enough for bucket sort which is proven on this research.

Acknowledgements

I would like to take this opportunity to state my appreciation for all who has helped me out by spending their valuable time for making this research a success.

I am truly grateful for the invaluable guidance and assistance provided through-out by my supervisor, Mr M.K Silva, senior lecturer at University of Colombo School of Computing, at all times. Without his thorough experience and constant advice, the work presented here would be incomplete.

Special thanks for my office staff members at Simcentric Technologies for given me the technical helps and courage for complete the thesis successfully.

Finally, I would like to express my heartfelt thanks towards my family for backing me up and encouraged me through many days and nights dedicated to the completion of this thesis.

Contents

List of Figures

List of Abbreviations

CPU - Central Processing Unit

GPU - Graphics Processing Unit

API – Application Programing Interface

GPGU - General Purpose Computation on Graphics Processor

SSE ─ Streaming SIMD Extensions

SIMD ─ Single Instruction Multiple Data

Chapter 1

Introduction

One of the most fascinating problems in computer science is sorting. Starting from small scale applications its use by large scale databases and large scale search engines, sort algorithms are used. Thus, optimizing the sorting algorithms have a big advantage.

Bucket sort is one of the most popular comparison based integer sorting algorithm. Optimizing the bucket sort is an interesting research area. Bucket sort is a non- comparison sorting algorithm in which elements are scattered over the buckets. Researchers are using different algorithms to sort the bucket. Some research has proposed ways to optimize the sorting elements and some propose to parallelize algorithms inside the CPU or GPU.

With the single core processor increase the number of transistors doesn't improve the performances of the system. Development of the multi core processor is given one of the solution for improving the performances. Multi core computing will improve the user experience in many ways. Such as improving the performance of activities that are bandwidth-intensive and compute, boosting the number of PC tasks that can be performed simultaneously. Various companies such as Intel, AMD. AMR and VIA have already worked on this multi-core solutions [1].

Over the past few years Graphics Processing Unit (GPU) has become competitive computer hardware against (Central Processing Unit) CPU [2]. The GPU has grown into a powerful programmable processor with application programming interfaces and also hardware which supports for programming aspects. Today GPU's provide highly parallel programmable processor which does not support only for graphic. In early general approach to GPU is bit difficult. The standard graphic APIs like OpenGL and DirectX were supported only one way to communicate with GPU because it uses only for rendering purposes. But now it provides like CUDA\C, OpenCL lead to map complex non graphical interfaces to the GPU. General Purpose Computation on Graphics Processor (GPGPU) also known as GPU Computing and this feature leads GPU to the next generation of high performance computing with accessible programming interfaces. Heterogeneous computing model of CPU and GPU is shown in figure 1.1 below.

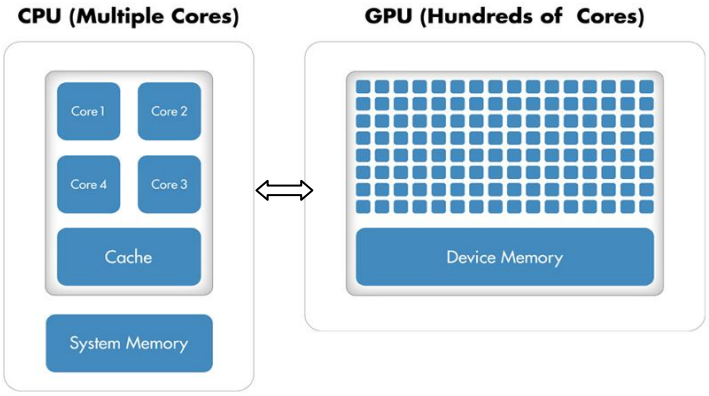

Figure 1.1 : Heterogeneous Computing Mode

.

1.1 Aims and Objectives

The research is to evaluate the Bucket sorting algorithm with various parallelize techniques. Since buckets works independently it is easy to apply any parallel techniques to optimize this sorting algorithm. Since GPU is going to be the most powerful and efficient device that can be used to optimize the Bucket sort algorithm`. Comparison between both GPU and CPU for bucket sort is done on this research; since we have large number of data to copy form CPU to GPU and again from GPU to CPU it would going to be an interesting analysis.

In present CPU has number of cores and thread affinity is handled by the computer operating system. So we hope to break the buckets into number of threads and look on how the threads are context switch and work with multi cores. Since Soring is not going to be a IO operation it could be interest to find how the soring algorithm will with the large number of thread count.

GPU is consisting of number of Cores and buckets can be run in parallel without context switching issue in CPU threads. Since large number of data have to be copy from CPU to GPU and GPU to CPU, PCI bus has to copy all those data it will going to be an interesting algorithm to compare with CPU optimized algorithms.

1.2 Scope

This research would be to evaluate Bucket sort with some parallel techniques. Bucket sort is done on multicore environment and with a CUDA supported VGA. Sorting is done with single threaded environment, multithreaded environment and inside the GPU. Comparison is done among various element sizes start from small number of elements to large scale of elements. Pre generated number set will be used for do the comparison.

Comparison is going to be done on

1. Single thread vs Multi-threaded environments.
2. GPU vs CPU
3. Number of elements with single thread and with various numbers of threads.

Ultimate goal of this comparison is to find an optimal way to do the bucker sorting algorithm.

1.3 Expected Contribution

When considering implementation of bucket sort algorithm as mention below there are algorithms implemented for optimized the sorting algorithms. But no one has done a detailed analysis to find, how bucket sort can be optimized using CPU threads and GPU threads. This research is done using new generation GPUs and CPUs and latest version of CUDA. This analysis is done in detail with those factors.

This research will give the idea of how the threads will optimize the bucket sort and how the modern GPUs will optimize the soring algorithms. So this is going to be an interesting research topic.

1.4 Outline

The rest of the document is structured as follows.

Chapter 2 - Literature Review: Detailed literature survey based on the entire project, includes background areas of the project and related researches and their contribution.

Chapter 3 - Methodology: Heterogeneous computing model (GPU and CPU usage models), the design of Bucket Sort algorithm and memory access patterns are discussed in this chapter. Different sorting optimization techniques used to optimize is declared here.

Chapter 4 is the implementation part. Four type of implementations are done here. Bucket sort in main thread, Bucket sort in threads, Bucket sort with SSE instructions and bucket sort in GPU using CUDA.

Chapter 5 will evaluate the algorithm implementation and find the best possible scenario for bucket sort.

Chapter 6 will summarize the worked done here. And describe future plans on this chapter.

Chapter 2

Background

2.1 Introduction

Sorting is one of the interesting research areas in computer science. Sorting can be optimizing by changing the algorithm or using high performance computing. Development of the multicore processors and with general purpose computing with GPUs there was a huge change in high performance computing. Multicore processor can run instructions separately on a processor core. Single processor core can run multiple instructions at the same time.

With the introduce of general purpose computing in GPU more processor power was taken for process run a program. There are number of physical processor that can run parallel. Although there are multi cores still one thread can run on one processor core. With the context switching thread will switch and works. So it the processor is i7 only 7 threads can be run at a time. But with GPU there are huge number of processors we can run multiple threads in parallel.

pThreads, std::threads and OpenMp can used to parallelize the algorithms in CPU. CUDA\C /OpenGL can be used to optimize the algorithm in hardware leave in GPU. Capabilities in the CPU as well as GPU are used to analysis the parallelism of the Bucket sort algorithm.

On this chapter soring the bucket level and hardware level optimization is discussed in deep. How these technologies can be applied to optimize the bucket sort will be the purpose of this chapter.

2.2 CPUs in High Performance Computing

In past years Central Processing Units (CPUs) had been increased their performance by adding more transistors to the microprocessor and increasing the clock frequency. Since 2003, microprocessor venders have switched to implement multi-core and many-core models to increase the processing power of microprocessors. As a result of this advantage of performance improvement in multi-core processors, parallel programming became more popular in computer industry intend of traditional sequential programming. The high

performance community has been developed parallel programs since late 1980s and early 1990s, but practicing in parallel programming was limited to a small number of software developers because of those parallel programmed applications need to execute on expensive, large scale computers.

There are many concepts that can use in applications to get the advantages of modern CPU's. Managing the work on each node, Managing the works for each core, Vectorising the loops and Minimizing the communication are some of them. Managing the work in each node is considered as domain parallelism. During the application work assign to a node should be isolated. OpenMP can be used to achieve this. Managing the work for each core or thread will need one level down of control. It's better to execute independent tasks in each core or thread. OpenMP or PGAS can be used to achieve it. By minimizing the communication between the CPU and RAM will make more effective when running an algorithm more efficiently. For all those concepts developer should have a good knowledge in the algorithm and how it works [3].

.

2.3 GPU's in High Performance Computing

Parallelism is the future of computer science. Microprocessors developer will consider on adding cores rather than work on a single core. The GPU is basically designed as a fix function processor design for rendering graphics. But in present GPU has become a powerful programmable processor, with both application programming interfaces (APIs) and hardware increasingly focusing on the programmable aspects of the GPU. The GPU is a processor with the enormous arithmetic capability.

The performance of the GPU is higher than a CPU, programing model and the architecture has a bit different than a single chip processor. The GPU has become a competitive hardware solution for parallel programming against the CPU because GPU provides hundreds of computing cores and it is more than a fixed function graphic pipeline with programming capabilities for other general purpose problems. Performance analysis of GPU and CPU is given in the figure 2.1 below. This huge gap makes the developers more fascinated in moving the more complex part into GPU. This attempt is simply known as General Purpose Computation on Graphics Processor (GPGPU) [6].

Inputs to a GPU is a list of geometric primitives in 3D coordinate system. GPU uses graphical pipelines. There are so many barriers when working with GPUs considering with the CPU. Programming environment is tightly constrained and the underlying architectures are largely secret. As a solution for the above problem in 2007, Compute Unified Device Architecture (CUDA) was released by the leading GPU manufacture in the industry called NVIDIA. CUDA is more popular and widely used for GPU related programing. CUDA provides c interface for the developer [4].

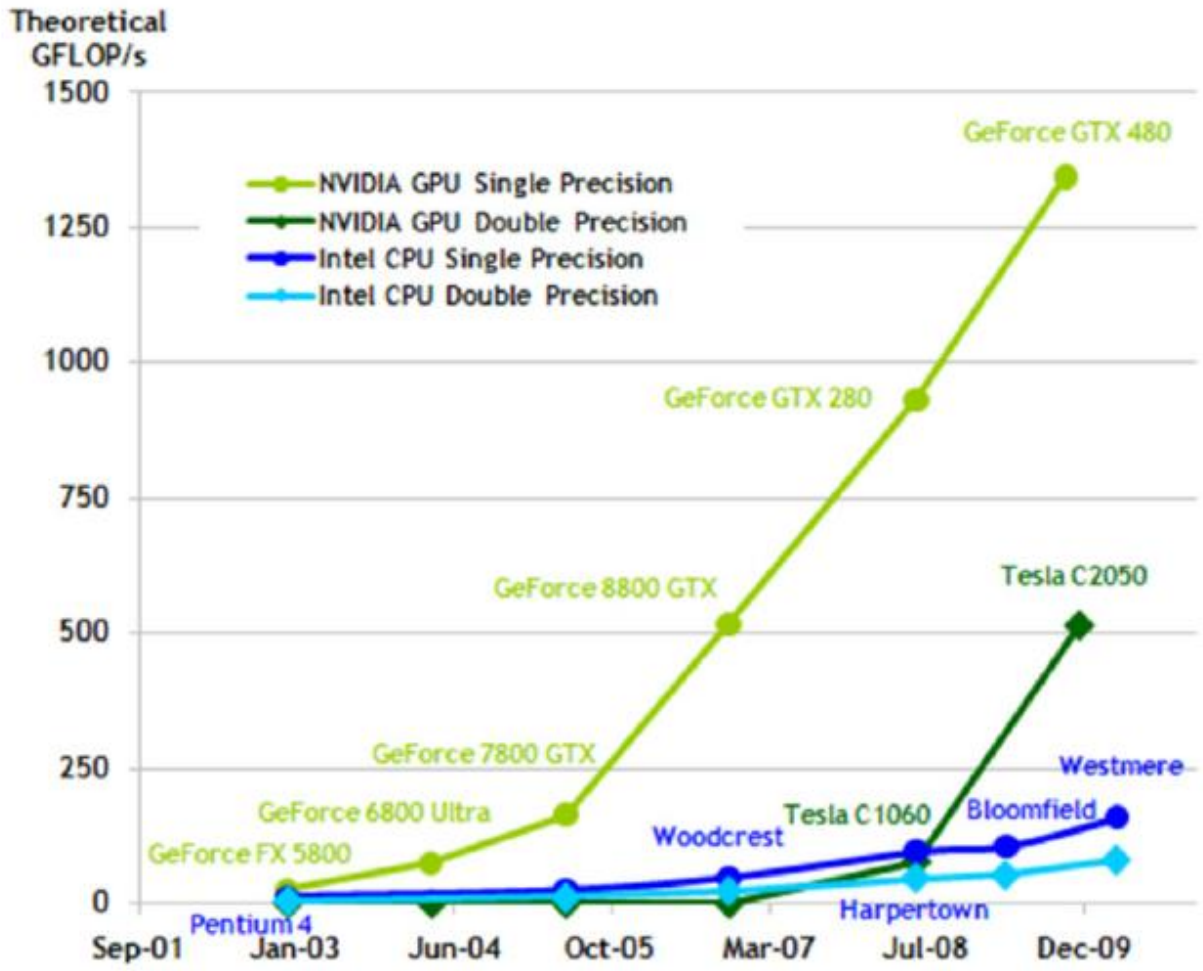

Figure 2.1:Floating-Point Operations per Second for the CPU and GPU[5]

2.4 Threads and Process Binding

Considering of Optimize the algorithm in the CPU, process binding is going to be an interesting topic. OpenMP and MPI are some of the APIs provided for bind the threads into a processor die.

2.5 Registry Level Sort

Modern High-performance processors provide multiple hardware threads within one processor. Many processors also provide a set of Single Instruction Multiple Data (SIMD) instructions, such as the SSE instruction set [10] or the VMX instruction set [11]. From the generation to generation improvements in CPU has improved rapidly. Some of the changes may optimize the program without the acknowledge of the programmer and some of them depended on the capability of the programmer. As an example, in modern CPU's and OS's thread affinity setting is not necessary to set. To take the SMID capability implicitly it is needed to vectorize a code. Modern CPU's are up to a factor 4 in most CPUs(AVX) and up to 8(AVX-512) on the KNL. SMID instructions are not implicitly work on data processing algorithms because it is not designed for data processing algorithms [12].

There are some special kind of CPUS that the size of SIMD- vector and the number of cores in the chip has been increased impressively. Intel Knights Landing(KNL) processor is an example to this. It will support AVX-512 instruction set.

Streaming SIMD Extension (SSE) is and SIMD instruction set extension to the x86 architecture designed by Intel and introduced in 1999 in their Pentium III serious. CPU instruction is said to be SIMD when same operation is applied to multiple data at the same time.

2.6 Computer Unified Device Architecture (CUDA)

CUDA is a parallel computing platform as well as an application programming interface (API) created by NVIDA allowed to use in CUDA enabled GPUs, used for General Purpose Computing on Graphics Processing Unit (GPGPU). This gives a direct access for the GPU.

CUDA works with C, C++, Java, python, .Net and Fortran. Considering prior APIs like Direct3D and OpenGL which required advanced skills in graphics programming

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

- Scattered reads
- Unified virtual memory
- Unified memory
- Shared memory- memory can be shared among the threads
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups [7].

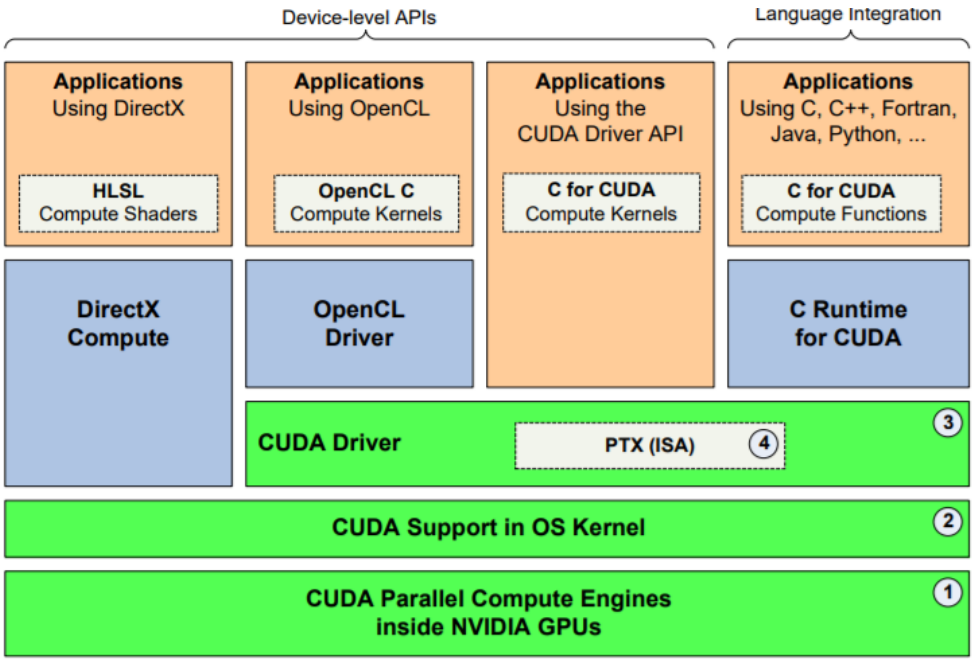

Figure 2.2: CUDA Architecture [7]

Figure 2.2 shows the architecture of CUDA. CUDA includes CPU serial code and GPU parallel code which includes functions that are executed many times, but on different data independently. Portions of codes that are compiled for the GPU are called the kernel. Kernel operations are executed by a set of threads to take the advantage of data parallelism. The operations which are represented in CUDA kernel function executed by a single thread. CUDA arrange those threads into a hierarchy called blocks and grids. Block contains set of independent threads which are worked as a work-group, grid contains a set of independent thread blocks and kernel is executed by a grid of thread blocks. Figure 2.3 shows the hierarchy of grid of thread blocks [7]. Each thread is uniquely identified by its global ID or

by a combination of its local ID and work-group ID. Each block also has its own unique block ID. The grid layouts can be 1, 2, or 3-dimensional.

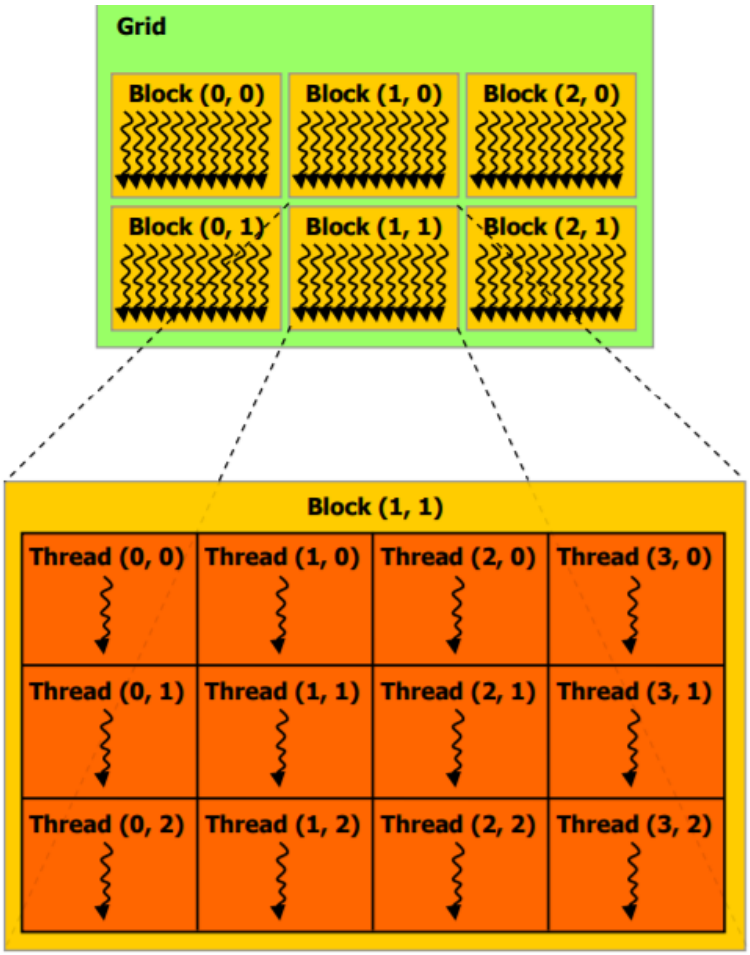

Figure 2.3 : Grids of Thread Blocks

2.7 Related work

Neetu Faujdar has done a detail analysis for bucket sort. Insertion, count and merge sort has been done for the buckets. Sorting benchmark has been used for testing the algorithm. He has found that if the range of key element increases then count sort will be worst in comparison to other sorting algorithms in both memory and time aspects [8].

Panu Horsmalahti compared radix and bucket sort algorithms against the memory consumption and time usage for different inputs. This has been shown that bucket sort was faster than radix sort but in some cases more memory is occupied by the bucket sort. Burak takmaz et al determined some improvement using some different sorting technique. The

author has mixed the bucket and shell sort. New approach is suggested and it works with greater performance [9].

N. K. Govindaraju, J. Gray, R. Kumar and D. Manocha are proposed GPUTeraSort. It is an improvement for bitonic merge sort. The bitonic merge sort has computational complexity of $O(N.\log(N)^2)$ and it can be executed by up to N processors in parallel. Comparing with AA-sort to the GPUTeraSort, both of them can be giving thread level parallelism and SIMD instruction implementation. An advantage of our AA-sort is smaller computational complexity of $O(N.\log(N))$ compared to complexity of $O(N.\log(N)^2)$ for the GPUTeraSort [14].

Berenger Bramas discussed CPU Vectorization. The term called SMID-vectors to call the data type managed by the CPU. He discussed SSE [14], AVX [15], and AVX-512 [4], supporting SIMD-vectors of size 128, 256 and 512 bits, respectively. He has discussed new sorting strategies using the AVX-512 instruction set on the Intel KNL. And it works only for special type of processors [12].

Chapter 3

Methodology

This chapter describes the analysis and design of bucket sort algorithm with different approaches. Some existing solutions and algorithms are discussed in chapter 2. After analysis them, I have come with better solution for bucket sort optimization.

Bucket sort will be tested on both CPU's and GPU's. OpenMP and std::threads will used to parallelize the algorithm in CPU. CUDA will used to parallelize the algorithm in GPU. C++ is used as the language to implement the algorithm. Visual Studio 2013 is used as the development tool used to develop algorithm in c++ and CUDA.

As the literature review done on above chapter 2 normally processing power of GPU is greater than a multi core CPU. But the issue behind the GPU is copping data from CPU to GPU and GPU to CPU.

3.1 System Architecture

The bucket sort algorithm will be run on both CPU and GPU. In figure 3.1 there is a sample overview of the system going be implemented. Input data will be predefined and generated by a program. A Profiler will run for calculating the running time of algorithm. A timer creates with QueryPerformanceCounter and Interl Inspector 2017 is used to calculate the runtime of the algorithm. Run time is displayed as the result. Runtime is taken as total time taken for run the algorithm.

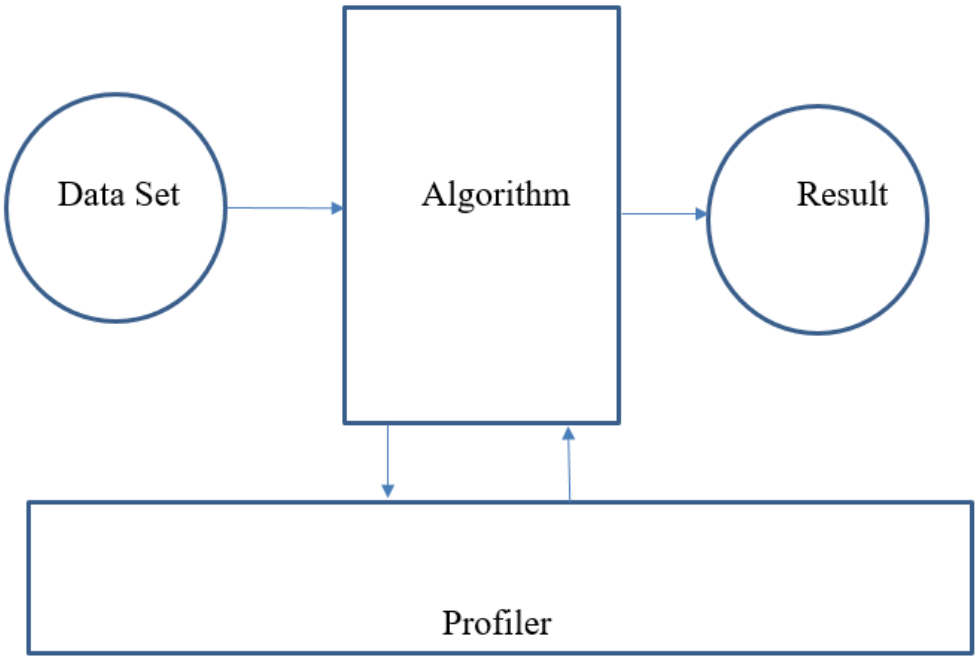

Figure 3.1 : Architecture Diagram

The implementation of the algorithm is done in different way.

- Normal Single Threaded Bucket Sort.
- Bucket Sort with std::thread
- Bucket Sort with OpenMP
- Bucket Sort in GPU

3.2 Single Threaded Bucket Sort

Sorting is done in main thread. Array (arr) and number of bucket is taken as n. Function is used to divide the array in to buckets. Interval count is calculating using

```
float interval = float(max - min + 1) / bucket_count;
```

```
min - Minimum number in the array
```

```
max – Maximum number in the array
```

to find the index of the element

```
int index = (arr[i] - min) / interval;
```

i denote the index of the array.

After breaking into buckets these individual buckets are sort separately.

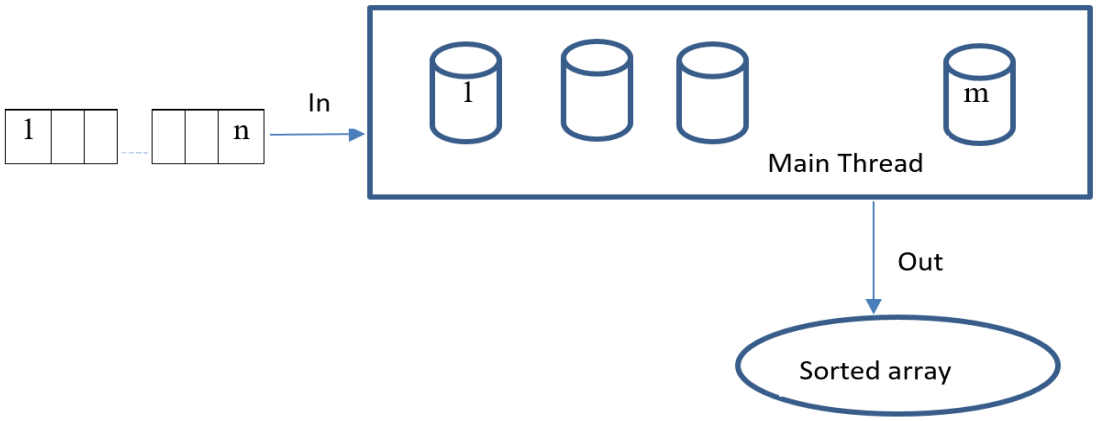

Figure 3.2: Bucket Sort

For individual bucket we have apply std::sort. All buckets are sorted in main thread and give the output.

3.3 Bucket Sort with Multi-Threaded Environment.

Std::thread class and openMp are used to create the multithreaded environment. Each bucket is sorted individually applying sorting algorithms separately for buckers. Bucket sort is a generalization of pigeonhole sort and is a cousin of radix sort. The figure 3.3 describe the overview of bucket sorting with threads.

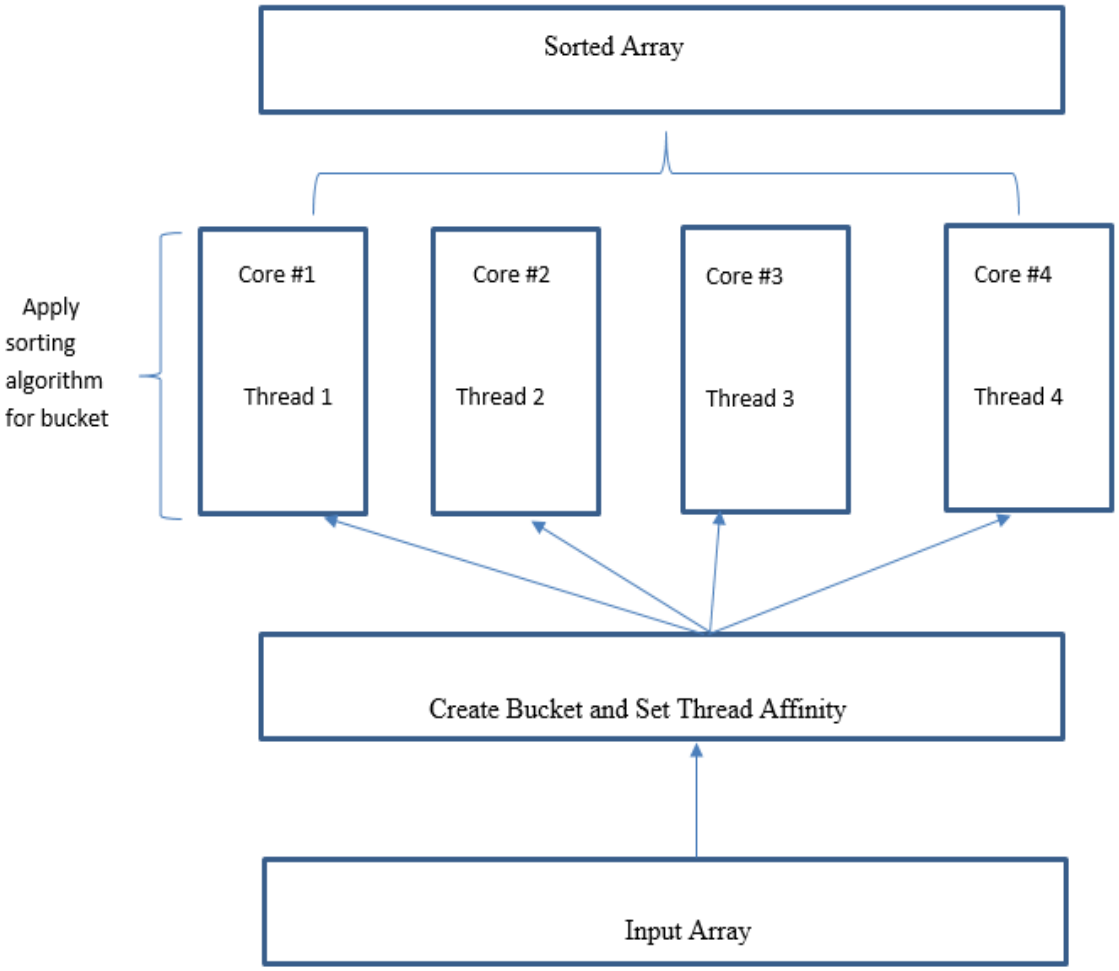

Figure 3.3: Bucket Sort with threads

When working with threads there are two issues one is thread affinity and next one is cache line invalidation issue. As discussed on chapter 2 Literature analysis modern OS's support this issue, we don't need to set thread affinity explicitly. Only issue we should address is cache line invalidation issue. This issue will be going to address in the implementation. Input array will process and created buckets. Each bucket will sort in threads and merge into array after all the buckets are sorted. Here number of threads can be changed. With changing the number of buckets and threads runtime will be calculated.

3.4 Bucket Sort in GPU

As mentioned in earlier chapters GPGPU or GPU Computing is a heterogeneous computing model and it allows executing the serial part of the program on the CPU and parallel part on the GPU. This CUDA architecture enables massively parallel execution of general purpose applications in GPU.

CUDA program consists of one or more phases that are executed on either the host CPU or device such as GPU. There is no data parallelism implemented in host code. The host code is ANSI C code and it runs as an ordinary CPU process. Device code is also written ANSI C extended with keywords which are supported for data parallel functions called kernels. Kernel function generates a large number of threads to make use of data parallelism. Figure 3.5 shows execution of a CUDA program.

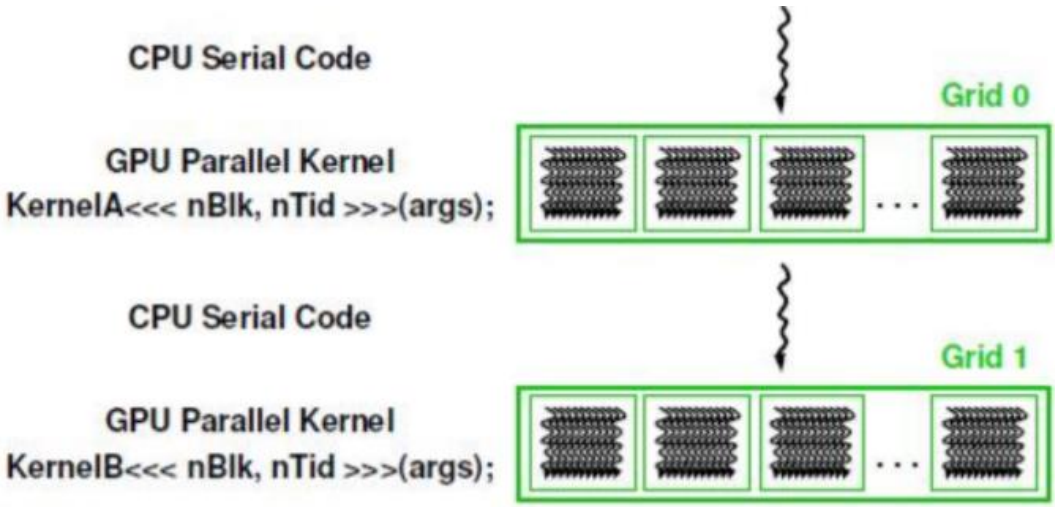

Figure 3.4: Execution of a CUDA program

Not like with std::threads number of threads can be run on GPU, we can sort more number of buckets in parallel. Copy data through GPU to CPU and CPU to GPU is an overhead. So it's a must to reduce the communication between this main memory and device memory communications.

When we consider efficient memory access pattern blocking become core strategy.

1. Parallelism among threads within a single block
2. Parallelism among threads within a multiple blocks

Bucket sort optimization has mainly we have to consider these factors.

1. Pruning the array and preprocessing done in CPU
2. Create buckets in host memory and move them into GPU
3. Sort the buckets inside device memory
4. Copy the device vectors to host memory.

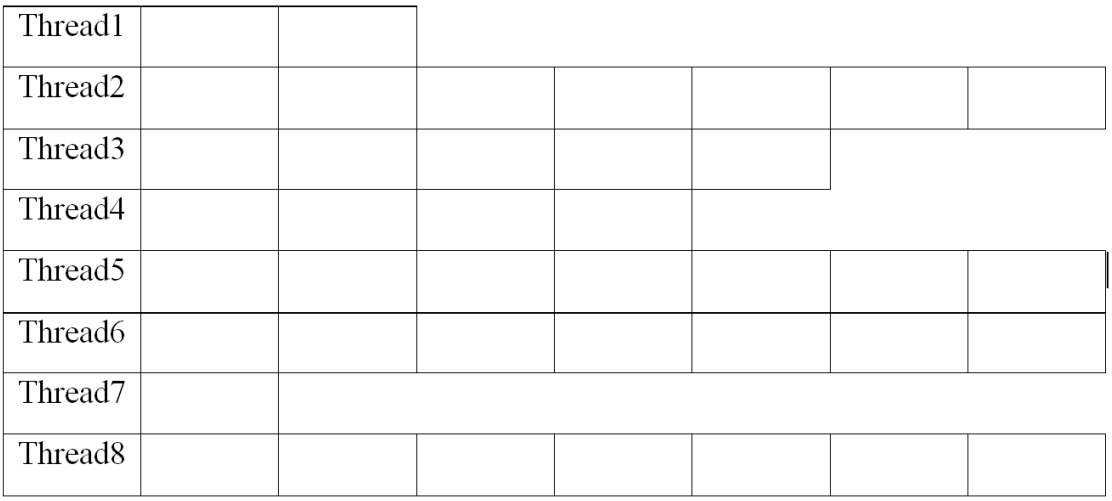

Figure 3.5: Thread representation of buckets

Figure 3.5 represent the bucket representation on the GPU memory. Each bucket assign to a new thread each thread runs on its own. Finally we have to copy each buckets in to main memory.

Chapter 4

Implementation

This chapter explains the steps taken to implement the proof of concept for Bucket sort with different approaches. According to the analysis here we discussed technologies used, frameworks, APIs with the justifications.

4.1 Hardware and Software Environments

We have test the system with following hardware and software

Interl(R) Core(TM) 7i-6700HQ CPU @ 2.60GHz
16GB ram
NVDIA Geforce GTX 960M

C++ Is use to implement the algorithm, Visual Studio 2013 is used as the tool for development. CUDA and std is used as packages for development.

Intel Inspector is used as an external profiler.

4.2 Algorithm Implementation

There are four type of implementations are mainly considering here.

- Single Threaded Bucket Sort
- Bucket Sort with thread
- Bucket Sort with OpenMp
- Bucket Sort in GPU

4.2.1 Single Threaded Bucket Sort

With this algorithm we will done the implementation in one single thread. Pseudocode is given below.

```
bucketSort(float arr[], int bucket_count)

{

std::map<int, std::vector<float>> Buckets;

var max = max value of array

var min = min value of array

var interval = (max - min + 1) / bucket_count;

do while i < num_of_element_in_array

       var index = (arr[i] - min) / interval;
       Buckets[index].push_back(arr[i])

End while

do while i < size_of_bucket

       // sort each bucket with std::sort

End while

}

Total algorithm is running on a single thread.
```

4.2.2 Multi-Threaded Bucket Sort

Mainly pruning part is done on main thread. Create buckets and assign the values on to buckets are done in main thread. After that according to the bucket count we will create separate threads for each. In modern OS it is support thread affinity so here we don't go to handle that issue. The only issue we are going to address is cache line invalidation here. It will discuss separately.

```
bucketSort(float arr[], int bucket_count)

{

std::map<int, std::vector<float>> Buckets;

var max = max value of array

var min = min value of array

var interval = (max - min + 1) / bucket_count;

do while i < num_of_element_in_array

       var index = (arr[i] - min) / interval;
       Buckets[index].push_back(arr[i])

End while

do while i < size_of_bucket

       // create threads and sort the bucket seperalty

End while

}
```

4.2.3 Bucket Sort in GPU

As mentioned in Design phase GPGPU or GPU Computing is a heterogeneous computing model and it allows executing the serial part of the program on the CPU and parallel part on the GPU. This CUDA architecture enables massively parallel execution of general purpose applications in GPU.

Mainly pruning part is done on CPU. Create buckets and assign the values on to buckets are done in CPU. After that each bucket is sort in GPU.

```
bucketSort(float arr[], int bucket_count)

{
```

```
std::map<int, host_vector<float>> Buckets;//

device_vector[numb_of_buckets]

var max = max value of array

var min = min value of array

var interval = (max - min + 1) / bucket_count;

do while i < num_of_element_in_array

       var index = (arr[i] - min) / interval;
       Buckets[index].push_back(arr[i])

End while

While j == num_of_buckets

Copy host_vector --→ device_vector

End while

Sort host_vectors

Copy device_vectors --→ host_vectors

End while

}
```

The pseudocode implementation is given above and trust::sort is done instead of std::sort here. I have use my own timer with QueryPerformanceCounters to calculate the time gone for executes the algorithm and Intel inspector also used to calculate the execution time. Calculate the median value for depict the final results through graphs.

Chapter 5

Evaluation and Results

We are going to do a detail statically analysis to find the best approach for the bucket sorting algorithm. Following implementations are done for the bucket sort.

- Sorting in single thread.
- Multi-threaded environment using std::thread and OpenMp.
- General Purpose Computation on Graphics GPGPU) (used CUDA)

What I need is to do a comparison among them. Comparison is done between,

- Among Multithread environments. (Look on std::thread and OpenMp. This is done to see whether std::thread done with full optimization)
- Threads with GPGPU.
- Single thread and multi-threaded environment.

5.1 Results

Before moving to optimizations let's have a look on basic one without optimizations. This works on a single threaded environment. Here the execution time shown in milliseconds in y axis. And number of buckets shown in y axis. Figure 5.1 shows the results.

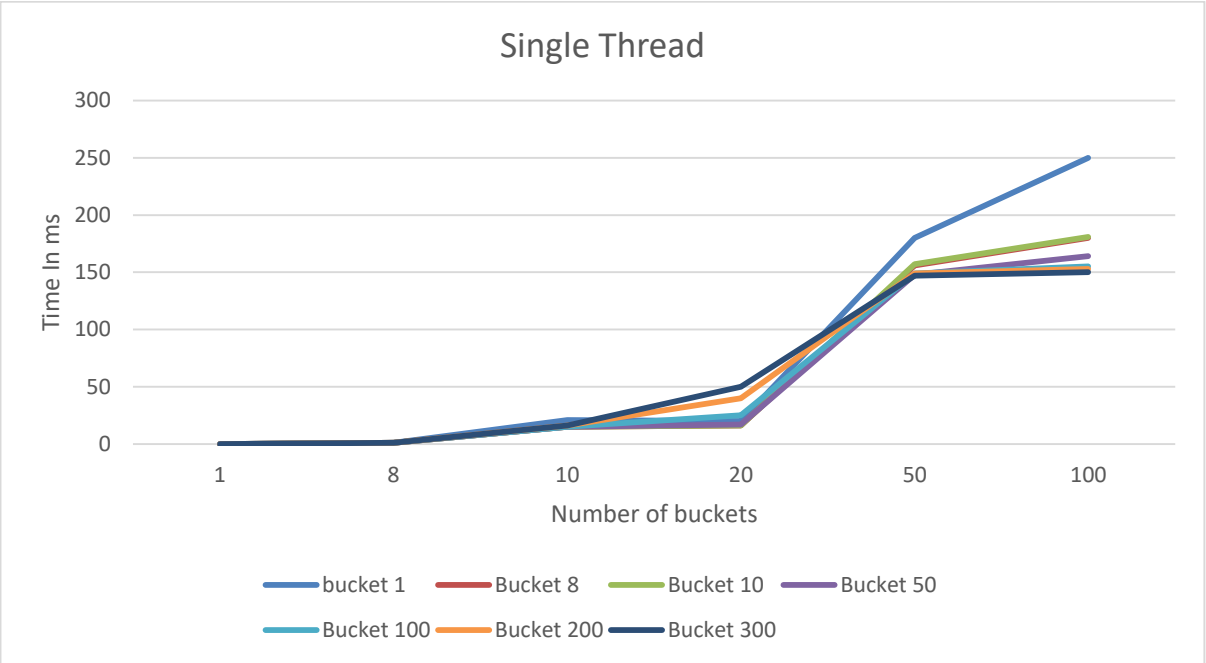

Figure5.1: Bucket sort in single thread without optimization

5.1.1 Bucket Sort in std::threads Results

In this result number of threads change according to the bucket count and check how the number of threads and bucket count can change the running time of the algorithm. Number of CPUs the machine used to optimization is 7, along with hyper threading 8 threads can run parallel. Figure 5.2 shows the results.

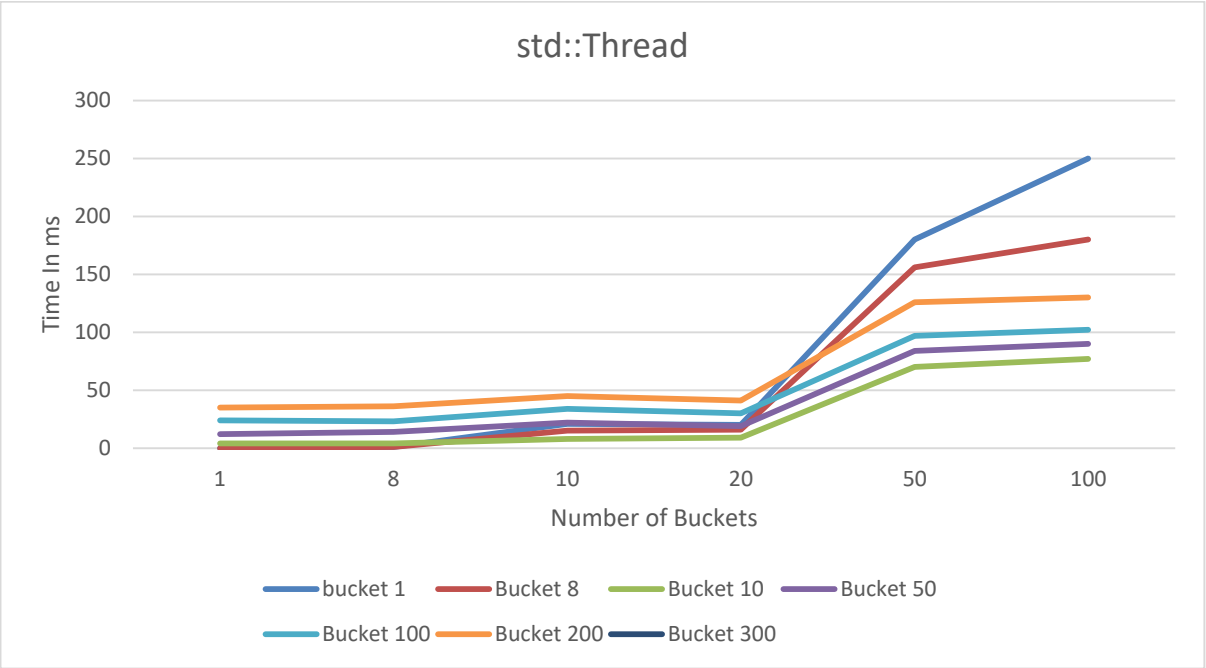

Figure 5.2 Bucket sort in std::thread

5.1.2 Bucket Sort in OpenMp Results

Std::threads and OpenMp both are multithreaded environments used in CPU. But to see the optimization on std::thread and OpenMp also used to do the optimization on bucket sort. Figure 5.3 shows the test results.

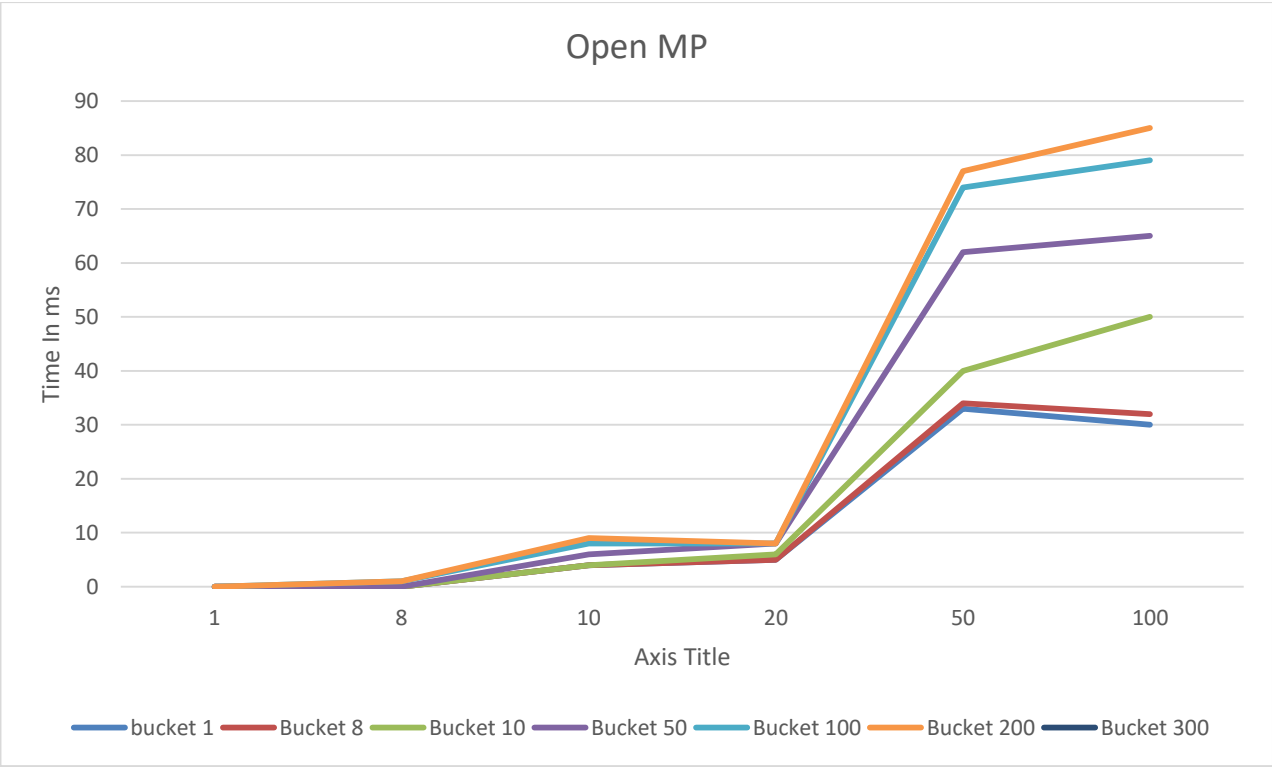

Figure 5.3: Bucket sort in OpenMp

Let’s do a comparison between std::threads and OpenMP. Figure 5.4 shows the test results.

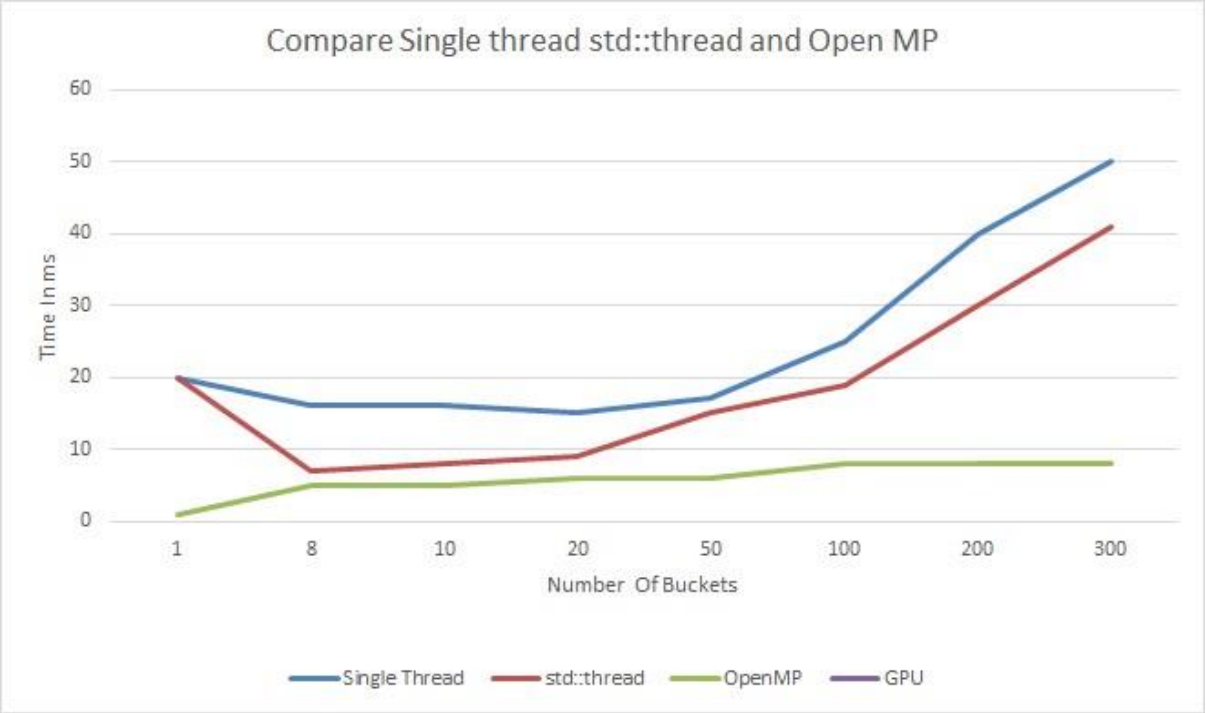

Figure 5.4: compare std::threads with OpenMp

5.1.3 Bucket Sort in GPU

This is our main scenario that that what need to test is the optimization on GPU. Figure 5.5 shows how the results shown in Intel inspector.

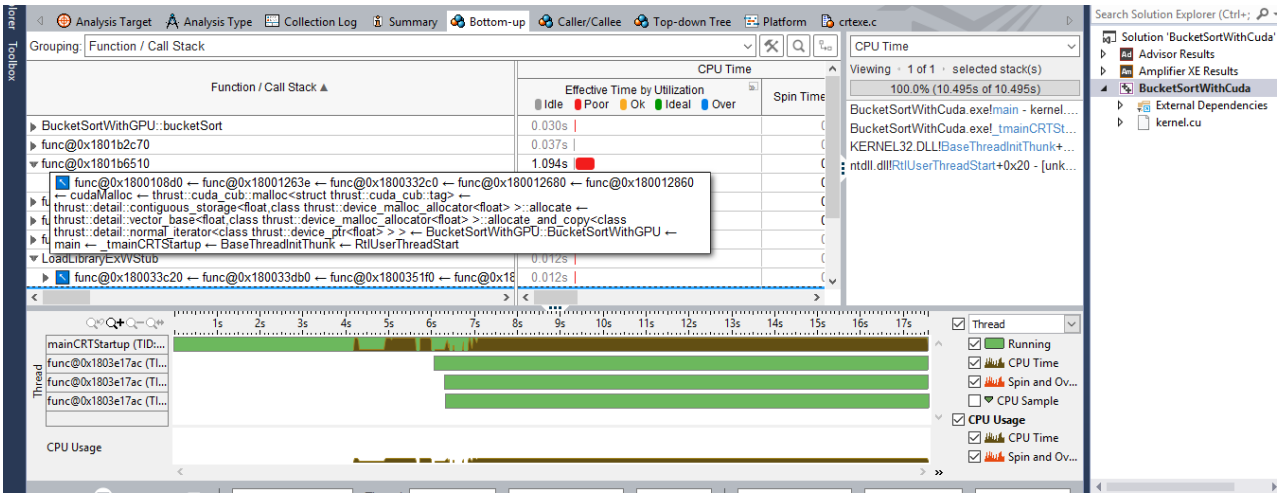

Figure 5.5: Intel Inspector view of Analysis

Sort the bucket inside the GPU and copy in back to main memory. This is the time calculated as the total time taken. `thrust::sort is used to do the sorting.`

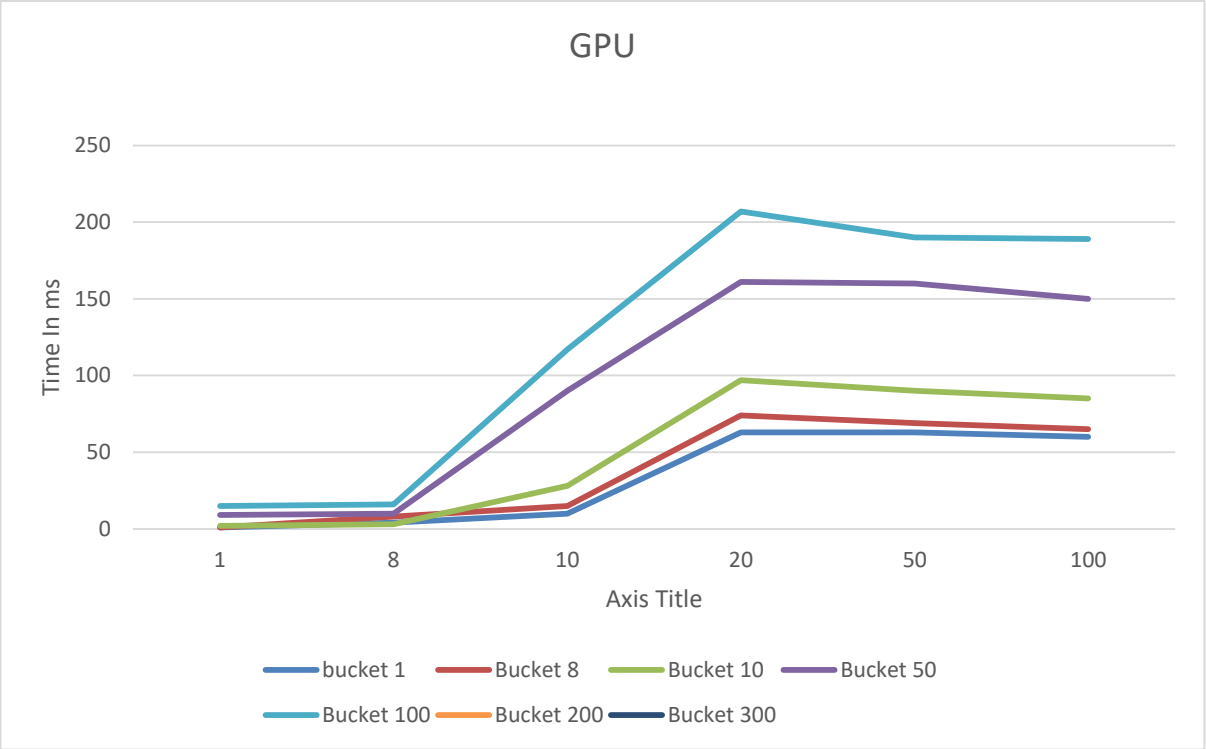

Figure 5.6: GPU sorting

5.2 Analysis and Discussion

Let's consider figure 5.1 and bucket sort happen on single thread. And it shows that this will works fine when the size of the array is small but when considering large number of data size and large number of buckers soring with single thread doesn't give good performance for a small number of element size using a GPU optimization is a waist.

Let's consider figure 5.2 and 5.3 together. Both of these are shows how the soring happens on multi-threaded environment. If the number of buckets gone high, number of threads also gone high. So we should assume that run time should go down. But according to the given diagrams that shows that thread 8 will be give the best results. Number of threads that can be run on this CPU simultaneously is also 8. So that shows how the thread context switching is depends on run time of the algorithm. On this scenario context switching of threads highly effected for the runtime of algorithm. It has slow down the runtime of algorithm.

Figure 5.4 shows a comparison between std::threads and OpenMp. According to this diagram OpenMp work more efficiently than std::thread.With the figure 5.7 we show the performance

of bucket sort in GPU. But considering with other optimizations with other algorithms it looks medium performance. This will happen because copy large array into GPU and again have to copy back them into CPU. According to figure 5.5 also more time is allocated to malloc the memory on GPU and copy data into GPU.

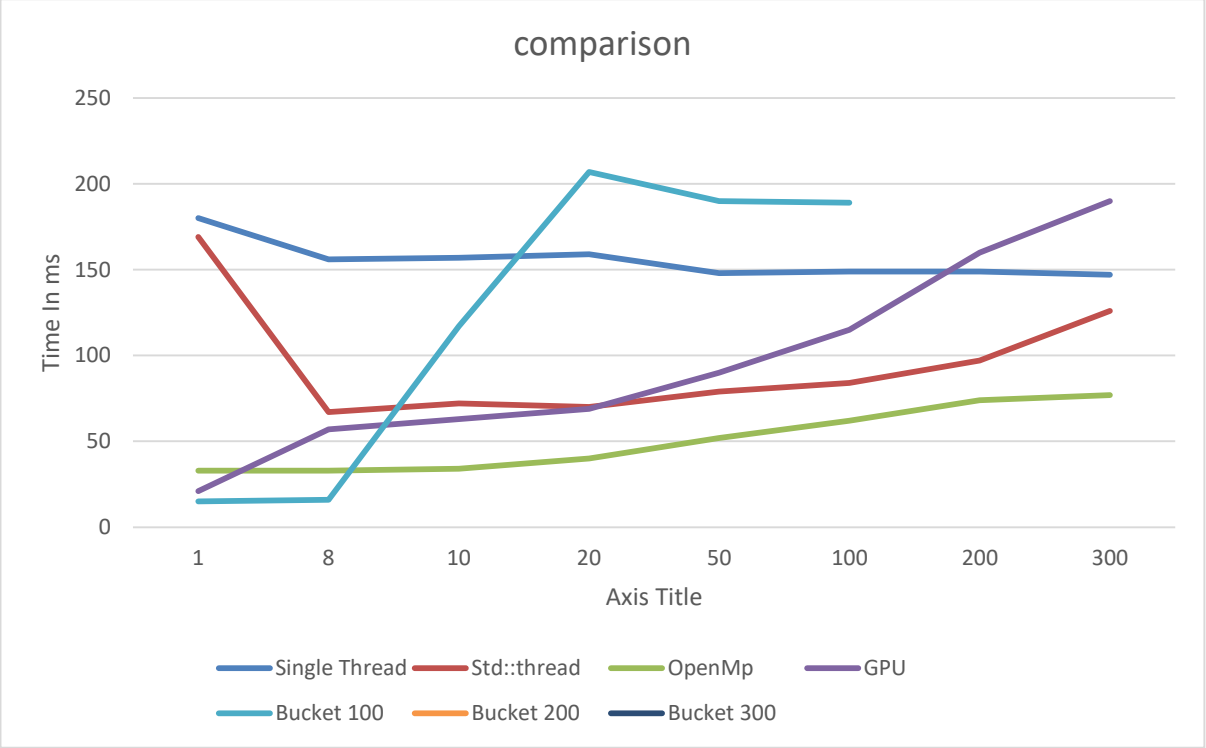

Figure 5.7 Compare Single thrrad std::thread OpenMP and GPU

Final results for the observations are, OpenMP is become the best way to do the bucket sort optimization. There is no any relationship with performance with the number of threads. If the numbers of threads are nearly equal to CPU count performance get high for bucket sort algorithm. This doesn't mean that GPU computing is not worth for bucket sort. Copy data into GPU have to optimize more. If that part is optimize sorting an array doesn't take a considerable time. When going to optimize GPU the researches have to more concern on optimizing the copy data from CPU to GPU.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

In this research work, highly parallel architecture of GPU and parallel architecture of CPU is used. Using the GPU will be give more performance than any other parallel algorithms was guess done before the research start. Use small scale of arrays to 10^6 numbers of elements to do the final analysis for the given optimizations.

After doing the analysis we came up with several conclusions:

- Working on single thread for small number of elements are more effective
- Maximizing the number of threads doesn't optimize the bucket sort
- When the numbers of threads are equal to CPU core count, give the maximum performance
- Context switching and waiting will affect more on performance of the algorithm
- GPU doesn't give more optimization for bucket sort

The final thing we found was GPU doesn't support as expected doesn't mean that GPU doesn't do a massive change on parallelism. It may cause because of copy the huge array for both sides. Figure 5.5 confirm that copy data from CPU to GPU take more time. There are two waits on this GPU optimization. One is waiting until sorting finish. And other wait is for copy GPU memory for CPU. These things may directly effect for the runtime of the algorithm. Optimizing more on memory allocation on GPU will reduce the run time of bucket sort algorithm.

6.2 Future works

This performance analysis is done using both CPU optimization and GPU optimization. But some issues can be arisen when working with large vectors. Its call cache line invalidation issue. This issue will be addressed with the future improvements. On this research we haven't look into the vectorization issue. SSE instructions can be used to optimize an algorithm. So better going for SSE instruction for do a performance analysis on a future research. When optimizing a code have to think more on memory allocation, branch predictions and etc.

Minimizing the copying data from GPU to CPU and CPU to GPU will make the algorithm more efficient. Rather than CUDA, OpenCL is becoming the standard architecture in GPGPU. Achieving cross platform support for sorting may be lead to better performance with better vendor support capability.

References

[1]] Multi-Core Program Optimization: Parallel Quick Sort in Intel Cilk Plus, Muhammad Ikram Lali(University of Gujrat) M. Saqib Nawaz(Peking University) Aboubakar Nauman(Sarhad University of Science & IT)

[2] D. Luebke S. Green J. E. Stone J. D. Owens, M. Houston and J. C. Phillips. Gpu computing. Proceedings of the IEEE, 96(5):879–899, May 2008.

[3] Programming for High Performance Processors by MichaelS

[4] GPU Computing By John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips

[5] Architectural Exploration and Scheduling Methods for Coarse Grained Reconfigurable Arrays by Giovanni Ansaloni and Laura Pozzi

[6] D. Luebke S. Green J. E. Stone J. D. Owens, M. Houston and J. C. Phillips. Gpu computing. Proceedings of the IEEE, 96(5):879–899, May 2008

[7] NVIDIA CUDA C Programming Guide Version 4.2

[8] The Detailed Experimental Analysis of Bucket Sort by Neetu Faujdar, Department of CSE,Amity University, Noida India, Shipra Saraswat ,Department of CSE,Amity University, Noida India

[9] Takmaz, Burak, and Murat Akin, "A new approach to bucket sort," Proceedings of the 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, World Scientific and Engineering Academy and Society (WSEAS), pp. 184-186, February 2008

[10] Graefe, G. (2006) Implementing sorting in database systems. ACM Computing Surveys (CSUR), 38, 10.

[11] Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M. (1998) Designing a pc game engine. IEEE Computer Graphics and Applications

[12] Fast Sorting Algorithms using AVX-512 on Intel Knights Landing , Berenger Bramas , Max Planck Computing and Data Facility (MPCDF)

[13] T. Furtak, J. N. Amaral, and R. Niewiadomski. Using SIMD Registers and Instructions to Enable InstructionLevel Parallelism in Sorting Algorithms. In Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures, pp. 348–357, 2007.