A STUDY OF PARALLEL SORTING ALGORITHMS

USING CUDA AND OpenMP

A MASTER’S THESIS

IN

SOFTWARE ENGINEERING

ATILIM UNIVERSITY

by

Hakan GÖKAHMETOĞLU

OCTOBER 2015
A STUDY OF PARALLEL SORTING ALGORITHMS
USING CUDA AND OpenMP

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY
BY
HAKAN GÖKAHMETOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF SOFTWARE ENGINEERING

OCTOBER 2015
Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

_____________________
Prof. Dr. İbrahim Akman

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____________________
Prof. Dr. Ali Yazıcı

Head of Department

This is to certify that we have read the thesis A study of Parallel Sorting Algorithms using CUDA and OpenMP by Hakan GÖKAHMETOĞLU and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____________________
Prof. Dr. Ali Yazıcı

Supervisor

Examing Committee Members

Assist. Prof. Dr. Gökhan Şengül

Assist. Prof. Dr. Murat Özbayoğlu

Prof. Dr. Ali Yazıcı

Date: 19.10.2015
I declare and guarantee that all data, knowledge and information in this document has been obtained, processed and presented in accordance with academic rules and ethical conduct. Based on these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Hakan GÖKAHMETOĞLU

Signature:
ABSTRACT

A STUDY OF PARALLEL SORTING ALGORITHMS USING CUDA AND OpenMP

GÖKAHMETOĞLU, Hakan

M.S., Software Engineering Department

Supervisor: Prof. Dr. Ali YAZICI

October 2015, 117 pages

This thesis reviews the parallel languages according to their computational complexities, in terms of time, while using sorting algorithms coded in CUDA and OpenMP.

The thesis evaluates the solution for parallelism at a maintainable cost of money and other efforts, for achieving acceptable results of timing when compared to parallel languages together, as well as the sequential execution. The detailed timetables comparing timings from CUDA, OpenMP and sequential code, are present in related chapters.

Eventually, the findings are examined to find the benefits of the parallel languages. The findings from the Chapter “Test and Comparisons” indicate that refactoring an algorithm to OpenMP language is easier when compared to CUDA language. However, results from comparing time values indicate that it is always the design of the algorithms, which makes it possible to achieve the performance that is expected of using the parallel languages.

Keywords: Parallel languages, GPGPU programming, CUDA, OpenMP, Sorting algorithms, empirical comparison
ÖZ

CUDA VE OpenMP KULLANARAK BİR PARALEL SIRALAMA ALGORİTMALARI ÇALIŞMASI

GÖKAHMETOĞLU, Hakan

Yüksek Lisans, Yazılım Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Ali Yazıcı

Ekim 2015, 117 sayfa

Bu tez, paralel dilleri hesaplama zorluklarına göre, zamanlama cinsinden, CUDA ve OpenMP dillerini kullanarak irdelemektedir.

Tezde sunulan çözüm önerileri, hem sıralı yürütmeye göre zamanlamada kabul edilebilir sonuçlar elde etmek için, hem de parasal ve diğer çabaların sürdürülebilir bir maliyetle paralel çözümünü değerlendirmektedir. CUDA, OpenMP ve sıralı kod zamanlamaları karşılaştırıldığında edinilen detaylı zaman tabloları, ilgili bölümlerde bulunabilir.


Anahtar Kelimeler: Paralel diller, GPGPU programlama, CUDA, OpenMP, sıralama algoritmaları, deneysel karşılaştırma
To My Mother
ACKNOWLEDGMENTS

I express sincere appreciation to my supervisor Prof. Dr. Ali YAZICI for his guidance and insight throughout the research. Without his contributions, this thesis would never be as complete as it is for now.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ÖZ</td>
<td>iv</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF ALGORITHMS</td>
<td>xiv</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>xv</td>
</tr>
<tr>
<td>CHAPTER 1</td>
<td>1</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Thesis Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Research Objectives and Statement of the Problem</td>
<td>1</td>
</tr>
<tr>
<td>1.3. Thesis Outline</td>
<td>2</td>
</tr>
<tr>
<td>CHAPTER 2</td>
<td>4</td>
</tr>
<tr>
<td>BACKGROUND INFORMATION AND LITERATURE SURVEY</td>
<td>4</td>
</tr>
<tr>
<td>2.1. Introduction to parallel computing</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1. Parallelism</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2. Sequential and Parallel programming</td>
<td>5</td>
</tr>
<tr>
<td>2.2. Literature Survey</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1. Definition of Parallelism</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2. Goal and Research questions</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2.1. Questions</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2.2. Field Research according to GQM metrics</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2.3. Research questions for this thesis</td>
<td>29</td>
</tr>
<tr>
<td>2.3. Conclusions for Chapter 2</td>
<td>31</td>
</tr>
</tbody>
</table>
7.7.5. Test cases for the quicksort and merge sort......................... 77

7.7.5.1. Test cases for merge sort .......................................... 77

7.7.5.2. Test cases for quicksort ............................................. 80

CHAPTER 8 ............................................................................. 84

CONCLUSIONS AND FUTURE WORKS .................................. 84

REFERENCES ........................................................................ 87

APPENDIX A ........................................................................... 98

General information about test environment.................................. 98

APPENDIX B ........................................................................... 98

Extra Profiler results ............................................................... 98

APPENDIX C .......................................................................... 101

Source Codes ......................................................................... 101

  Reduction Sum Algorithm ..................................................... 101

  hybrid merge sort function (in CUDA) .................................. 101

  HBquick sort (in CUDA) ....................................................... 106

  hybrid merge sort (in OpenMP) ............................................ 109

  HBquick sort (in OpenMP) ................................................... 112

  How to Build the Codes ....................................................... 115

APPENDIX D .......................................................................... 117

Survey documents .................................................................... 117
LIST OF TABLES

1. Performance metrics occurrences in the papers ........................................... 20
2. Compiler commands for function callable scopes ........................................ 41
3. Compiler commands for variables ................................................................ 41
4. Theoretical complexity values for sequential algorithm .................................. 47
5. Description of all Algorithms used in this thesis ............................................ 52
6. Time values for constant array size and changing defer threshold .............. 61
7. Hybrid merge sort timings with changing threshold ...................................... 62
8. Timing Values for HBquick vs. Hybrid merge sort, in CUDA and OpenMP 63
9. Execution times values for all HBquick sort versions .................................. 64
LIST OF FIGURES

1. Parallel execution vs. Sequential................................................................. 6
2. NVIDIA GeForce 8800 GTX, introducing first unified-shader architecture.  
   (Courtesy of NVIDIA.) ....................................................................................... 7
3. Comparisons of theoretical peak performance in GFLOP/sec. for double  
   precision format ................................................................................................. 8
4. Comparisons of theoretical peak performance in GFLOP/sec in single and  
   double precision formats for various devices .................................................... 8
5. Comparisons of theoretical peak performance in GFLOP/sec for double  
   precision format ................................................................................................. 9
6. Heterogeneous programming ............................................................................. 10
7. Achieved bandwidth of memory for Algorithm 1 ............................................. 12
8. Achieved bandwidth of memory for normal deep copy operation ................. 12
9. Papers grouped according to publication years .............................................. 17
10. Count of papers that give pseudo code or name a BM ................................. 18
11. Distribution of count of papers according to hardware and parallelism  
    relation ........................................................................................................... 23
12. Sorting algorithm names count that appeared in pool of papers .................... 26
13. Reasons for the choice of sorting algorithms ................................................. 27
14. Names of parallel languages occurred in the papers ..................................... 27
15. Abstract component view of CPU vs. GPU [7] ............................................ 32
17. Simple, data-level, vector addition [20] ......................................................... 34
18. CUDA memory model [7] ................................................................................ 37
19. CUDA execution model [20] .......................................................................... 41
22. Dynamic parallelism improvements [98] ......................................................... 43
23. Time values for HB quicksort vs. built-in ....................................................... 48
24. HBquick (CUDA) sort execution model ....................................................... 49
25. Execution time for HBquick (CUDA), 400K numbers, and altering defer threshold................................................................. 50
26. Conventional sequential sorting algorithms compared............................ 51
27. Hybrid merge sort ........................................................................ 54
28. HBquick sort timing when defer threshold is equal to array size to sort...... 60
29. Speed-up achieved against CUDA official sample cdpSimpleQuickSort .... 63
30. Parallel vs. serial sort compares with large array sizes......................... 65
31. Merge sort Nvidia VP profile, size=1M DT=1024................................ 66
32. Insertion sort profile shows warped operation ..................................... 67
33. Merge function profile shows warped operation ................................... 68
34. Merge sort profile with size=300K and DT=1024................................. 69
35. Occupancy statistics for TopDownMerge kernel................................... 70
36. Profiler stats for CopyArray kernel................................................... 71
37. Test case 1 for hybrid merge sort..................................................... 78
38. Hybrid quicksort with 8 numbers....................................................... 80
39. UMA automatic memory management............................................... 98
40. SM spreadsheet ............................................................................. 117
LIST OF ALGORITHMS

1. Code to minimize latency .......................................................... 11
2. Code to minimize latency (but creates bank conflicts) ............. 13
3. Split function for hybrid merge sort ........................................... 54
4. Split function for hybrid merge sort in OpenMP ...................... 55
5. Split function for hybrid merge sort in CUDA .......................... 56
6. Merge sort function for hybrid merge sort in CUDA ................. 56
7. Selection sort (in OpenMP) function for hybrid merge sort in CUDA....... 57
8. Quicksort function from HBquick in CUDA (dynamic parallelism) .......... 58
9. Selection sort (in CUDA) function from HBquick in CUDA .......... 59
10. cuRAND random number generator for CUDA ............................. 73
11. Random number generation for OpenMP .................................. 73
12. CUDA or OpenMP test case with intentional redundant code .... 75
13. CUDA kernel code for testing sorted array .............................. 76
14. A function for runtime error checking in CUDA ....................... 76
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>three-dimensional</td>
</tr>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices, Inc.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>Giga Floating-Point Operations Per Seconds</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Computing on GPUs</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>INTEL</td>
<td>Intel Corp.</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>Nvidia</td>
<td>Nvidia Corp.</td>
</tr>
<tr>
<td>OpenACC</td>
<td>Open Accelerators</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction, Multiple Threads</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessors</td>
</tr>
<tr>
<td>SP</td>
<td>Streaming Processors</td>
</tr>
<tr>
<td>UMA</td>
<td>Unified Memory Architecture</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1. Thesis Scope

Development of computer hardware has taken a step towards shrinking the size of components while improving the hardware architectures of modern CPUs and GPUs. This evolution has come to a point that modern CPUs now have multiple numbers of identical cores [1] on the same chip. That means any computation, which has been done in sequential, using older CPUs, can now be done in parallel using the cores of newer CPUs.

The aim of the work and the tests done in this thesis is to find out the value added by increasing the computational capacities and increased number of cores in the newer CPUs and GPUs.

1.2. Research Objectives and Statement of the Problem

Both parallel languages and sorting algorithms have been studied and applied in industrial projects since 1960s. There are already many successful projects for specially designed hardware for parallelism and parallel sorting algorithms [2].

Since parallelism is not a new field for computer science, in this thesis the focus is on the new opportunities for achieving the benefits of hardware-accelerated parallelism, similar to the development of supercomputers. However, our approach involves using parallel languages that are designed for PCs. This means, we are seeking to lower the hardware costs, while sacrificing some performance.

This study proposes a cheaper way of obtaining some performance gains in terms of time and money, by using the parallel languages designed for modern PC hardware, instead of using the traditional computer clusters, grids or supercomputers.
The objectives of the study are:

- Review challenges and trade-offs using newer parallel languages.
- Calculate and compare results of timings between parallel and sequential algorithms.
- Analyze the findings from our applied tests, to validate the hypothesis.

By aiming the objectives above, the study will reveal an overall view of using the parallel languages designed for personal PCs, i.e. CUDA and OpenMP. According to the CUDA’s developer company Nvidia, “the GPU is a sleeping giant to be awakened to do general purpose and scientific computing” [3]. In addition, many vendors maintain OpenMP to work with their compilers. Both of these frameworks can be obtained free from their vendors’ websites. In addition, starting from CUDA v6.5, the CUDA’s “nvcc” compiler also supports OpenMP. That means CUDA and OpenMP can be used at the same time, which is very important since CUDA cannot parallelize CPU code but OpenMP can. Then CPU code once run in serial in CUDA programs can now run in parallel. A more detailed view about how important this event is presented in Chapter 5.

1.3. Thesis Outline

This thesis is organized as follows:

In Chapter 1 the leading factors for this thesis’ motivation are given. Then the scope of this thesis is determined using the research objectives. Chapter 2 states background information and literature survey about parallel architectures and their differences. A field literature review about the parallel computation is made using the relevant papers from a chosen set of related fields of GPGPU programming and many core programming. Chapter 3 provides information about the GPU architecture. Chapter 4 provides information about GPGPU programming with CUDA. Moreover, samples of sequential code versus their parallelized versions are discussed in this chapter to supply background information to viewers of this document. Chapter 5 provides information for designing the sorting algorithms in parallel. Moreover, generating the sorting algorithms is explained using CUDA and OpenMP languages, which are many-core GPU versus many-core CPU respectively.
In Chapter 6, pseudo code samples of parallel sorting algorithms are explained in detail. In Chapter 7, test and comparisons of test results are explained. Chapter 7 reveals the most appropriate approach for parallelization on a personal computer. In Chapter 8, conclusions are discussed about our finding, as well as the references for future studies to be made.
CHAPTER 2

BACKGROUND INFORMATION AND LITERATURE SURVEY

2.1. Introduction to parallel computing

Parallel computing makes use of concurrently running the processes that are belonging to larger computation, for this reason the divide-and-conquer approach is usually preferred over other techniques [1].

Parallel computing is the use of computing resources, actual CPUs or CPU cores, in concurrence for better performance. In this thesis, a similar approach has been taken to CPU parallelism. The GPU cores allocated to share computation tasks with CPU through the use of CUDA language.

Traditionally, sequential programs are optimized with single CPU efficiency in mind. The optimization is made usually only to the algorithm. However, since all traditional sequential languages consist of blocks and reserved words, it is expected that there would be many ways to implement a code, with different algorithms. Therefore, using another algorithm might reveal better results than trying to tweak the parts of the code. However, traditional parallel languages, such as MPI, have some limitations when speaking of data-level computational granularity. That consists of the inflexibility of implementing every possible algorithm in the code.

In our thesis, CUDA and OpenMP are used, which both have the facility to implement any desired algorithm, without losing the data-level computational granularity.
2.1.1. Parallelism

There are two levels of parallelism:

- Task-level parallelism is the distribution of the same execution process among many computational nodes (e.g. CPUs or CPU cores). There is use of threads in this type of parallelism, although the threads shares work for executing functions. However, communication overheads can result in slow processing similar to synchronous processing.

- Data-level parallelism is executing many data items simultaneously, via distributing the data among multiple threads (or processes). These threads share the data to execute not the functions, unlike the task-level parallelism.

This concept may not be self-explaining. Although, an example from an operating system might be more helpful, such as in an operating system a thread is used to operate processes. A process may have many threads, but a single thread, usually main thread, will be active during lifetime of a process. The operating system can run each thread associated with processes concurrently if there is more than one CPU or many CPU cores. Task-level parallelism, is similar to operating system executing many threads at once. Then data-level parallelism is analogous to; those processes that executes concurrently are belonging to the same process. This means, data-level parallelism has a more fine-grained level of parallelism.

2.1.2. Sequential and Parallel programming

A computer program consists of tasks that are small pieces of code where an input is consumed, a function is applied or an output is generated. These operations are called data dependent, i.e. if one task uses the output of a previous one. Moreover data dependencies, negatively affect the concurrent execution of tasks, additionally the parallelization of the sequential code.

In view of that, it is the programmers’ capabilities and understanding of parallel language to overcome the undesirable performance of data dependent tasks. Parallel execution and sequential execution of processes is illustrated in Figure 1 [1] below.
2.2. Literature Survey

2.2.1. Definition of Parallelism

The ever increased performance demand for 3D graphics on the market for computers, overwhelmed the capabilities of CPU which lead to use of specially designed hardware to process graphics data that is called GPU. Since 2008, Nvidia GPUs are available with programmable processors, making it possible to process large data-blocks.

GPGPU (General Purpose computation on GPU) is the use of GPU to execute computational tasks, which were conventionally carried by the CPU [4]. GPGPU only turn out to be popular with the introduction of both programmable shaders and floating-point support on GPUs. Figure 2 demonstrates the NVIDIA’s Fermi architecture presenting the first unified shader architecture [5]. The Fermi architecture furthermore improves CUDA programming language, which is also considered in this thesis.
In Figure 2, the little boxes which SP (stream processors) is written on them, are the unified-shader architecture mentioned above, they were once called the *shaders* when graphic processing was about less granular *task-level* processing. In addition, the L1 and L2 in this illustration represents the on-chip, low latency, memories. They are same type of components, which can be found on modern CPUs. A more detailed explanation is present in CUDA thread parallelism topic, in Chapter 4.

Precision format is a computer number format, which is governed by the IEEE standards. For floating-point numbers if the number is represented in 8 bytes (64-bits) the number is then in double precision floating-point format. Until 2007 neither CPUs nor GPUs were capable of doing double precision arithmetic. Some GPUs and CPUs, mostly the lower end ones, are still missing that kind of competency. As a result, those devices can only compute in single precision format. In addition, the remaining few devices, that are capable, are slower at double precision than single precision while doing computational operations. Figure 3 shows the theoretical peak performances of CPUs from Intel® and GPUs from AMD® and NVIDIA®. Green lines, NVIDIA GPUs are mostly the high-end professional GPUs and the blue line, INTEL processors belong to Intel MIC processors, which are later called XEON processors. Figure 3 [6] also tells us that the precision format is slowly evolving according to the general market needs.
Figure 3 Comparisons of theoretical peak performance in GFLOP/sec. for double precision format.

Figure 4 shows the comparison of NVIDIA professional accelerators versus high-end gaming GPUs and Intel CPUs [7].

Theoretical GFLOP/s

Figure 4 Comparisons of theoretical peak performance in GFLOP/sec in single and double precision formats for various devices.
It is clear from Figure 4 that, most computationally intensive engineering problems will need too much effort, in terms of time. Because, the line trends of the graph shows crawling increase in throughput values in each year. This means, there is still some progression needed for professional accelerators’ throughput for double precision to catch up with, gaming GPUs’ throughput for single precision.

Finally, a rather interesting sight is the performance gain obtained by per processing element in the devices. That is, GFLOPS (Giga Floating-Point Operations per Seconds) throughput of each core of the CPUs or each SM of the GPUs. In Figure 5, peak performance comparisons using double precision per computing element reveals that Intel processors with less than 20 cores have better per core performance, when compared to GPUs with 16 or so SMs.

![Theoretical Peak Performance per Processing Unit, Double Precision](image)

**Figure 5** Comparisons of theoretical peak performance in GFLOP/sec for double precision format

Heterogeneous computing refers to systems, mostly embedded on the same circuit, using more than one kind of processor. Apart from 3D graphics rendering, with the use of GPGPU, these heterogeneous systems can now perform intensive arithmetic or algorithmic tasks, by distributing the serial task to CPUs and parallel tasks to other devices.
Figure 6 shows the device hierarchy between the host device, the CPU, and the GPU. Serial tasks or functions execute on CPU, are named the host functions. Additionally, the kernel code, which is called the global functions are executed on the GPU, than the control defers back to the host.

OpenCL and OpenGL are already both capable of HP (heterogeneous programming). Both allow users to create more than one context for an executable, and then user is free to choose from all OpenCL/OpenGL capable devices in the system to carry executing that context.

Since CUDA v6.5 and on GPUs with compute capabilities SM 3.5 or higher the unified memory architecture is introduced. If a memory region allocated with command `cudaMallocManaged` than that memory region will be accessible from both GPU and CPU via a device pointer of desired type. Only sequential operations are allowed for now (i.e. CPU has to wait the GPU to finish any current operations to
complete). That means faster computation time because UMA (in CUDA) eliminates expensive memory copy operations needed before.

On the other hand, latency will be introduced using off chip-memory; because CUDA unified memory allocates the memory on the machine’s main memory. Yet, laptops, game consoles and other embedded GPUs share system memory; as a result, there will not be any performance drop in these systems.

Warp (only available in CUDA context) means, a group of threads that synchronously executing a single instruction. The current size of this group is called a warp size and it is constant for all architectures, which is 32. In addition, every thread in a warp has to operate on the same instruction at the same time. Otherwise, (e.g. in case of a branching condition) a warp cannot happen. Moreover, the warp count limit, where warps concurrently execute on an SM is 64, for current Maxwell architecture (Nvidia). The detailed explanation is present is Section 7.5.

Latency [8] is the time required to perform an operation, and it is measured in wall-clock time in computing. This measure is common to all architectures; it is approximately 20 clock cycles for arithmetic operations and 400 or more cycles for memory I/O operations. According to [8,9] latency hiding is quite different from the traditional methods. Algorithm 1 shows array sum operation, strided for 64 threads per warp, only available in SM 5.0 or higher machines. In addition, each thread does n load/store operations instead of one operation, line 2, where n is equal to stride.

**Algorithm 1** Code to minimize latency

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>stride = blockDim.x*gridDim.x</td>
</tr>
<tr>
<td>2:</td>
<td>while i = threadIdx.x<em>gridDim.x+blockIdx.x</em>gridDim.x → size of(arrays)</td>
</tr>
<tr>
<td>3:</td>
<td>b[i] = c[i] + c[i]</td>
</tr>
<tr>
<td>4:</td>
<td>i = i + stride</td>
</tr>
<tr>
<td>5:</td>
<td>end while</td>
</tr>
</tbody>
</table>

The occupancy achieved for normal copy and Algorithm 1 are; 74% with 6553 total warps and 45% with 410 total warps respectively. Moreover, instructions per warp (IPW) are 92 and 281 respectively. That means, for certain operations, there exist an algorithm for parallel CUDA code, where using less warps per kernel with less occupancy does makes the program run much faster, hence the latency hiding.
In addition, the calculation of effective bandwidth is like the following: \( \text{BW}_{\text{Effective}} = \frac{R_B + W_B}{\text{time} \times 10^9} \), where \( R_B \) is the total number of Read operation on a clock cycle [10], and \( W_B \) is the total number of Write operations on a clock cycle. In the CUDA versions programs in this thesis, kernels have timing value of 650\( \mu \)s for normal copy and 136\( \mu \)s for Algorithm 1, and thus kernel written with Algorithm 1 has much more higher throughput as the above bandwidth formula implies.

Figure 7 Achieved bandwidth of memory for Algorithm 1

Figure 8 Achieved bandwidth of memory for normal deep copy operation
Figures 7 and 8 above obtained by Nvidia Nsight Visual Studio Edition. Figure 7 is obtained profiling the kernel written using Algorithm 1, and Figure 8 is obtained with profiling the kernel written in the traditional way (i.e. threads*blocks = array size). Therefore, using same thread to do more operations, not only helped hiding latency in the program but we achieve more efficiency in terms of occupancy and memory usage. In addition, percentages in the $L1/\text{tex cache}$ and $L2 \text{ cache}$ boxes show the memory hit values, where there is an improvement of 33.3% to 33.4% and 33.3% to 66.7% for L1 and L2 caches respectively. Moreover, the improvement achieved using 64 blocks with 1024 threads each (i.e. 65,536 threads) instead of 1024 blocks with 1024 threads each (i.e. 1,048,576 threads). Successively, fewer threads are used for covering latency.

Algorithm 2 Code to minimize latency (but creates bank conflicts)

2: block_start = threadIdx.x + (blockIdx.x * skew_value)
3: block_end = block_start + desired_work_size_for_each_thread
2: for i = block_start → block_end
3: $b[i] = c[i] + c[i]$
4: $i = i + (\text{desired\_work\_size\_for\_each\_thread})$
5: endfor

There is even a third algorithm (written with Algorithm 2), which computes the same 1M array addition, with just 64 blocks and 1024 threads in 46 $\mu$s. This means, it hides the latency even more. However, the algorithm introduced many bank conflicts, while the explanation of that is beyond the scope of this chapter. However, for readers’ advice there exists another explanation is present in the Section 3.5 shared memory.

Fork-join model is the thread hierarchy model commonly used in shared memory systems. In OpenMP language, a main thread operates the main function, and when the parallelism needed other threads join the operation (with special pragmas written before the start of parallel section of the code), which is called the fork (of threads). After that parallel part finishes the operation is deferred to the main thread again which is called the join (of threads). In this thesis, same model is applied to the algorithms and code samples are presented in Appendix C.
2.2.2. Goal and Research questions

2.2.2.1. Questions

In this section, a systematic mapping [11] research is made by searching the parallelism subject according to the venue appearances with the subject of this thesis (i.e. computer parallelism and sorting with parallel languages).

The papers studied are all from reputable resources such as IEEE Xplore, ACM DL, Google Scholar and Web of Science. The related papers selected from these revenues according to their association to the parallel computing and parallel-sorting subjects.

For the assessment of our subject in field papers, the Goal, Question, Metric (GQM) methodology is used. Since the GPGPU is a relatively, very new area and papers usually concentrate on how the results are affected using GPGPU and mostly give little or no clue for future improvements. For that reason, GQM method will supply some way to categorize the general subject to smaller subcategories. Meanwhile, giving opportunity to search for what future studies may reveal using the data found in the current field of papers.

Below are the research questions and their short descriptions:

- **RQ 1** – Does the speed-up that is gained with parallelism give enough reason to refactor whole program to a parallel language? Existing languages such as MPI, Cray computers, were about keeping the parallel part in minimum because of communication and programming overheads. However, CUDA, OpenMP, OpenCL are all capable of parallelizing the whole parts of the program without creating bottlenecks.

- **RQ 2** – How one can measure the performance (e.g. speed-up, scalability, efficiency, throughput, latency, occupancy, branch divergence, active warps achieved, registers used per block or per SM, warps used per SM, shared memory used) of the code? There are many performance metrics for calculating the performance gain for CUDA programs and other languages. Especially, on CUDA, Nsight Visual profiler is a very efficient way for profiling the program’s behavior. In other words, the reader has to know the gains of the parallelism other than speed-up, and the other metrics have had provided to prove the performance gains from parallel languages. Besides
that, in Section 2.2.1 the latency hiding section, it is proven that two kernels doing same operations have the same timing values. However, one of the kernels has very higher throughput than the other even if the occupancy is lower. That means kernel with lower occupancy is capable of computing more data, when the input size grows. Therefore, providing more metrics other than timing would be useful.

- **RQ 3** – How does the parallel languages differ from the traditional languages, other than the efficiency of the parallelism? The languages mentioned in this thesis are actually APIs. This means they use some traditional languages as the intermediate. However, at some point using parallel languages become harder to implement the algorithms or refactor the program and test the possible outputs.

- **RQ 4** – Are the achievable parallelism and the capabilities of the hardware used related? An older SM 2.0 architecture GPU has higher clock frequencies than the newer SM 5.0 GPU. This means a basic memory I/O operation will run 2x to 10x faster than the newer GPU. However, there are many functions implemented with new architecture such as GPU managed variables, invocation of kernels from inside another kernel (which both eases programming and speed-up the execution). Now, even these two newly added functions, can change everything, from code refactoring to easy maintenance, which were not possible with a SM 2.0 capable GPU. Therefore, there should be a relation with parallelism and hardware capabilities.

- **RQ 5** – What are the difficulties of debugging and testing? Debugging always seems hard with C language where both the CUDA and OpenMP use it as intermediate language. Although with CUDA there is a choice of using FORTRAN or C++, it is hard to say they are more testable.

- **RQ 6** – What affect does refactoring a traditionally written code to parallel have on stability or on maintainability. Most papers discuss the benefits such as timing performance on task-based conversions to parallel languages. Therefore, focus is still on the kernel code, in which few ways of testing are printf function and kernel debugging which is only possible on few platforms.
- **RQ 7** – Is there an effective way to achieve parallelism, without refactoring the code or writing it from the scratch? OpenMP achieves task and even the data parallelism with built-in directives; these are compiler directives such as `#pragma omp parallel for`. Therefore, there does not seem a need to refactor the code but to add some directives and leave the structure of the code unchanged. However, that type of ease can be very expensive using CUDA. Because CUDA focuses on more fine-grained data parallelism, there is a need to refactor the code or even the algorithm of the code to achieve a healthy parallelism in CUDA.

- **RQ 8-9-10-11** – What is the sorting algorithm used. Name of algorithm, the reason of choice, and in what language the algorithm is written in. These are pooling questions, rather than exploring the paper quality, these questions assess the quality of our pool.

### 2.2.2.2. Field Research according to GQM metrics

A pool of some 500 papers is collected from the reputable venues. The criteria was the keyword search in the papers and finally quickly checking the files before saving to the pool.

The keywords are carefully selected according to their relation to the parallel programming and sorting subjects. The toll of papers is publicly accessible for view as a Google spreadsheet document from [16].

It turns out only 90 of the 500 papers were related to our goal-question metrics; these metrics were (parallel AND (sorting OR sort) AND (parallel languages AND sort)). In the final pool, only (parallel AND sort) keywords are used for a final elimination criterion.

#### 2.2.2.2.1. Publications according to their publication years

In the final papers, the publication years are grouped into the years between 1965 and 2015.
It is clear from the Figure 9 most of our selected papers from the post 2000 era. However, this does not mean the sorting subject became popular recently, because most of papers in our pool are about CUDA language (28 of 47 papers, where the paper is about sorting subject). Then the figure tells us, 45 papers are published between 2010 and 2015; the sorting is popular with the data-level parallelism using languages.

Where, 40 of papers in our pool is about OpenCL, CUDA or OpenMP, and only 10 papers is about MPI.

In addition 8 out of 10 papers whose main language was MPI, also describes a sorting algorithm in the paper. Moreover, 18 out of 30 papers, whose language is CUDA, are describing a sorting algorithm in that paper.

2.2.2.2. Results for systematic mapping

In this section, extracted information from all papers will be used to answer the GQM questions. By doing so, readers should get the overall idea for the trends in this subject. In addition, one would expect to find the weak and strong research points in our pool of papers.
In our pool, more than half of the papers used a benchmark or give pseudocode for the algorithms. Since authors who has academic career write the majority of the papers, it is expected that these papers should contain more information about the code. This is a very important property because the papers that contains more information about algorithms, contribute for further researches from other authors, so they are more valuable.

There are 20 papers in the pool, which are not studying a sorting algorithm; they are about improvements to the parallel languages. Some of them also contributed to give the benchmarks used or give the pseudo codes for the proposed algorithms, if any.

In addition, Fig. 10 shows the papers whose main subject is sorting but do not give any clue about the algorithm (i.e. not giving pseudo codes). Similarly, some papers neither use a benchmark nor give a pseudo code for the algorithm.

Rightmost bar shows, there are 17 papers, which are about sorting but does not give code or name a benchmark. In nine of these papers, the main language was OpenMP or MPI [33,50,63,69,74,82,83,84,91]. However, authors of those papers did not give any clue about the design of the algorithm. This is pure non-sense because timing comparisons will never be enough to re-create that work again. In other words, their work does not contribute as much as a paper with adequate information given about an algorithm.

**Figure 10 Count of papers that give pseudo code or name a BM**
2.2.2.2.3. Answers for research questions

Eleven questions are defined as our GQM questions, and then each paper’s content is searched for answers to these questions.

2.2.2.2.4. RQ1 - *Does the speed-up, gained with parallelism gives enough reason to refactor whole program to a parallel language?*

The majority of the papers, 37 out of 68, give a direct answer as yes to this question. The answer is assumed in some papers as “yes” because code was a toy program, or a part of program that is already all other parts of the code refactored in to the parallel language.

In paper [52], the findings indicate the proposed hybrid algorithm is order of a significant magnitude faster than the previous sorting algorithms. However, this paper accepted as saying as “yes”, because the paper indicate that the proposed sorting algorithm is also faster than the CPU based single thread application.

In addition, there are papers where the answer to the RQ1 is explicitly negative.

- The paper [33] mentions that the code has some parts that run on CPU.
- The paper [61] uses CUDA, says heterogeneous computing can make for the lack of CPU lower throughput and limitations of GPU memory size.
- The paper [77] proposes a technique to minimize the communication overheads in HPC, so that more parts of the program would be refactored into parallel language.
- The paper [85] concludes that the involvement of the large data files in database nodes requires load-balancing and special algorithms, on nodes, where parallel operations cannot be carried.

2.2.2.2.5. RQ2 - *How one can measure the performance of the code?*

The answer for the RQ2 is straightforward, there are common keywords in the era of parallel programming, and some of them are mutual with the other computing areas. Table 1 show the keywords and their meanings.

For example, in paper [32] use of more CUDA blocks is expected to increase the occupancy of the GPU, this is marked as *scalability* (SC). In the same paper,
Coalesced memory I/O operations are expected to be more efficient (EF) in terms of wall clock time, when using aligned vector types (e.g. float16). In addition, working in larger portions of the memory that hold data, is used for hiding latency (LT).

Moreover, in the papers [63, 67, 77, 86] other forms of performance metrics are discussed; communication overhead, power efficiency, network connection speed, propagation delay, respectively. Nevertheless, since each occurred in only one paper, they are not explicitly mentioned in Table I, below.

<table>
<thead>
<tr>
<th>Performance metrics</th>
<th>Abbr.</th>
<th>Meaning</th>
<th>Occurred in papers</th>
</tr>
</thead>
<tbody>
<tr>
<td>bandwidth</td>
<td>BW</td>
<td>Memory I/O operations per second</td>
<td>24, 26, 32, 33, 40, 43, 47, 48, 63, 67, 68, 83</td>
</tr>
<tr>
<td>branch divergence</td>
<td>BD</td>
<td>Possible paths in code, causing threads to diverge</td>
<td>43, 47, 48, 63, 67</td>
</tr>
<tr>
<td>code optimization</td>
<td>OP</td>
<td>Any sort of refactoring to the code, where a benefit occurs as a result</td>
<td>24, 26, 43, 47, 48, 63, 67</td>
</tr>
<tr>
<td>efficiency</td>
<td>EF</td>
<td>Any benefit that can occur using parallel lang.</td>
<td>27, 32, 33, 39, 40, 43, 45, 46, 47, 48, 52, 63, 67, 83, 84</td>
</tr>
<tr>
<td>latency</td>
<td>LT</td>
<td>A delay in program’s execution time, measured with wall-clock.</td>
<td>27, 35, 36, 39, 43, 47, 48, 52, 63, 67, 68</td>
</tr>
<tr>
<td>occupancy</td>
<td>OC</td>
<td>Percentage of device utilization</td>
<td>43, 47, 48, 63, 67</td>
</tr>
<tr>
<td>registers /block /SM</td>
<td>RG</td>
<td>Limit of temporary memory per thread</td>
<td>43, 47, 48, 62, 63, 67, 80</td>
</tr>
<tr>
<td>scalability</td>
<td>SC</td>
<td>Maximum concurrently running thread count that can be achieved on particular GPU.</td>
<td>27, 33, 36, 37, 39, 43, 47, 48, 52, 53, 58, 59, 63, 65, 67, 69, 80, 82, 83, 84, 89, 90, 91</td>
</tr>
<tr>
<td>shared memory usage</td>
<td>SM</td>
<td>On chip memory, accessible to threads only</td>
<td>24, 26, 32, 43, 47, 48, 63, 67</td>
</tr>
<tr>
<td>speed-up</td>
<td>SU</td>
<td>Ratio of wall-clock time comparisons</td>
<td>Nearly all papers</td>
</tr>
<tr>
<td>throughput</td>
<td>TP</td>
<td>Number of Arithmetic operations executed per second</td>
<td>24, 26, 27, 37, 38, 39, 43, 47, 48, 49, 55, 56, 59, 60, 61, 63, 67, 69, 71, 84, 85, 90</td>
</tr>
<tr>
<td>warps /SM</td>
<td>WS</td>
<td>Limit of concurrently executable blocks of threads for each SM (CUDA only)</td>
<td>43, 47, 48, 63, 67</td>
</tr>
<tr>
<td>warps achieved</td>
<td>WA</td>
<td>Limit of concurrently executable blocks of threads</td>
<td>32, 43, 47, 48, 63, 67</td>
</tr>
</tbody>
</table>
Table I shows the performance metrics according to their appearances in the related papers. It is clear that most of the papers do not give enough metrics about performance other than the speed-up achieved.

This study revealed that 61 papers from our pool, uses the speed-up (SU) metric either solely or with few other metrics to summarize the performance of parallel languages. In addition, 18 papers only gives the results in speed-up (SU) achieved and there are no other metrics given. This will lead us thinking, most authors expects better timing results from using a parallel language and nothing more. (For example, if I tell you that, a rigid body accelerates from 0 to 200 km/h in 15 seconds, and tell afterwards, this means 2x times the force applied to that body compared to another body of unknown mass. The force applied is impossible to find in here.) Then, this is same with computation; the speed-up achieved will never reveal the facts that are if the full occupancy of device is achieved or if the memory bandwidth is used efficiently and so on.

Other common metrics from Table 1 are, bandwidth in 12 papers, efficiency in 15 papers, latency in 11 papers, scalability in 23 papers, and throughput in 22 papers. Interestingly, above properties are mostly mentioned in the papers about one or more sorting algorithms is discussed. That means, authors who study sorting knows that measuring just the time is not enough to conclude the work is done, and they use adequate metrics for proving the benefits achieved from using a parallel language.

2.2.2.6. **RQ3 - How do the parallel languages differ from the traditional languages, other than the efficiency of the parallelism?**

In the pool of papers, 48 papers tell almost nothing other than giving the performance as the single answer to this question. Since, the performance is not a measurable metric, we add papers to this category that either refers performance explicitly or does not give any results other than the speed-up achieved.

However, some papers explicitly mention differences between parallel and traditional languages. Paper [35] mentions that the usage of recursive functions in OpenMP decreases performance in OpenMP. In paper [36] three issues are addressed. These were portability; lack of a wide variety of tools present for parallel languages; moreover, it is mentioned that the readability is a certain issue throughout
in parallel languages. Paper [39] states that, the algorithms differ significantly especially when there is a need for porting a code line-to-line from a parallel language. Paper [46] proposes a solution to increase performance, which is the low cost of buying GPUs, gives an opportunity for using more hardware to achieve more performance. Paper [48] states, in CUDA, memory is not efficient enough, because of the reason that GPU device memory size is limiting. Paper [53] highlights, the scalability in CUDA is an issue because thread/block model and there is a limit to the number of blocks an SM can execute concurrently. Paper [60] draws attention to the code design; it is said in the paper that the algorithm plays the crucial role achieving the performance expected of the parallelism. In addition, the same paper states that global memory is the limiting factor achieving better timing values. Paper [67] states that, energy efficiency of GPUs, enables use of more hardware, which in return increases the parallelism.

2.2.2.2.7. RQ4 - Are the achievable parallelism and the capabilities of the hardware used related?

This is the most anticipated subject in most of the papers, 48 papers has directly addressed the relation between the hardware capabilities and the parallel performance. Only 6 papers either do not mention the relation or the subject of the paper does not fit to this relation.

However, in some papers there are solid references to relation of hardware and achievable parallelism. The paper [69] mentions the increase in computation time is occurred in MPI due to communication overhead. In other words, adding more nodes for speeding-up the execution makes a reverse effect and slows down the overall process. However, paper [40] says their algorithm made a difference, and adding more processors to the computation do actually speed-up the process. In the papers [45, 47, 52, 58, 62, 75] at least 2 GPUs used for comparing the hardware capabilities. The proposed algorithm, or test bed run on all GPUs separately to test the hardware contribution to parallelism achieved. Then all of these papers conclude that the better hardware exposes more parallelism.

In addition, paper [67] states that GPU is best for memory related and batch operations, e.g. matrix operations, and CPU is best for data-dependent operations. Furthermore, paper says the heterogeneous computing is most suitable for scalability.
2.2.2.8. **RQ5 - Does testing or debugging mentioned? What are the difficulties of debugging and testing?**

The testing and debugging subjects are the least occurred ones among all relative subjects in our pool of papers. In fact, debugging was not even mentioned once in a single paper. In addition, testing has never been mentioned in any of the studies as in its original meaning, i.e. to test the behavior of the code for desired output. However, almost every paper mentioned the word “test” as comparing the timing values of different algorithms.

Although, there were some faint references to actual testing; paper [24] states that OpenCL function names makes debugging/testing harder. Paper [34] does not test the code because adding test code affects the wall-clock time measurement of the program. Hence, the testing is being left for another run. The paper [37] states that, in OpenACC language debugging is easier when compared to other parallel languages. Paper [59], testing the code increases the communication overhead in MPI based environment.

Finally, in paper [62], a tool for CUDA is proposed, called GPU_PF, which is a library to debug; memory copy operations; kernel functions; file I/O operations; and function calls.
2.2.2.2.9. RQ6 - What affect, does refactoring a traditionally written code to parallel have on stability or on maintainability?

Again, the subject was mentioned in only a dozen of papers, and far less made a direct reference to writing the parallel code. As a result, all papers carefully examined for references relevant to stability and maintainability. Unfortunately, the word *maintainability* was not even occurred once in any paper. Although, some papers give clues about the other subject, that is stability. In papers [32, 33, 40, 64, 84], the stability issue is addressed. Nearly, every parallel language has some limitation on the syntax of the code, which affects the algorithm. Now converting from a single threaded language to parallel language would require the change of algorithm, which is then, becomes an issue of stability. Because the single threaded code usually tested before but with the algorithm change, newly constructed algorithm needs additional tests.

In addition, paper [35] states that, refactoring to parallel code brings loss of readability (of the code). Then, communication overheads occur due to old design was being made for single threaded computation. In addition to above, the paper suggest a method for busy waiting a node to cover-up the latency caused by branch-divergence in the code. Another paper [53] states if an SM in a CUDA capable GPU is faulty, every code run on this machine will result in an erroneous state. In that paper, it is sought to build an automated tool to expose if a faulty SM resides in the GPU, and afterwards converting the algorithm to by-pass the faulty SM. By doing so, the code executes on other available SMs on GPU and the tool (by refactoring the code) prevents faulty SM from joining the execution operation.

Finally, a couple of papers adopt the heterogeneous computing. Thus, in case some part of the code is no longer feasible after refactoring to parallel, that portion of the code is left unchanged in single threaded language.
2.2.2.10. **RQ7 - Is there an effective way to achieve parallelism, without refactoring the code or writing it from the scratch?**

The parallel languages need change in the syntax of the code, or in the extreme, the change of whole algorithm to achieve full capacity of the medium used, with the exception of *shared memory* languages. For example, OpenMP pragmas simplifies coding. Although a straightforward answer for easily refactoring the code to parallel is not easy to find. For instance in most programming languages, there are only suggestions about programming optimizations, or structuring algorithms, and the programmer up to his/her mind after that point. For example, the new CUDA *dynamic parallelism* model eliminates the need for complicated code design, by enabling run time dynamic allocation of number of threads to run each kernel. This ability, simplifies the CUDA code, allows code structure similar to single threaded code. However, it is still a matter of code design to achieve the performance.

Refactoring the code is mentioned in some of the papers. Paper [33] states the different ways (algorithms) of parallel coding results in 3 different classes of algorithms. Another paper [35] mentions, using OpenMP *pragmas* makes easier to decide the code paths, this is more similar to the single threaded code. The paper [37] states that OpenACC language is easy to program with, and eliminates complex code structures to achieve parallelism. The paper [39] proposes an optimizing algorithm for refactoring single threaded code to CUDA language. The paper [44] proposes Mesi-CUDA optimizer to do the refactoring. The paper [46] explains their sorting algorithm, type of count sort, as being in the similar form already with the single threaded language. The paper [54] proposes a tool for CUDA to translate the *for loops*, from C code. The tool only translates for loops but these primitives are mostly used building blocks for most of the algorithms, as well as the matrix transpose functions which the paper proposes to convert automatically to CUDA language. Another paper [62] translates the *matlab* related code to GPU architecture.

In conclusion, the code refactoring is much related to speed-up achieved. For the reason that, validity of the output of the code in desired time, cannot be more important than time spent for coding that algorithm.
2.2.2.2.11. RQ 8-9-10-11 – What is the sorting algorithm used. Name of algorithm, the reason of choice, and what is the language algorithm written in.

In the pool, 48 of 69 papers are about sorting algorithms. Meanwhile choosing the papers to the pool criteria was only sorting. Therefore naming the sorting algorithm names mentioned in each paper, might help readers of this thesis document to figure out the trend of sorting algorithm choices in the literature.

The result is not surprising, if we recall most of the papers are in the CUDA language. Usage of data partitioning sorts (bitonic sort, merge sort, and quicksort) is the most continent way to data-level parallelism in CUDA. For example, in this thesis merge sort and quicksort algorithms were used for the same reason. Other popular sorting algorithms are n-way sorts and count sorts (sample sort, histogram sort, and count sort) are most used in shared memory or cluster systems, because the suitability of the algorithms on those systems.

The reason of choice, for particular algorithm, in each paper is shown in the Figure 13. Some papers, naturally, gives more than one reason. Consequently, the total number of the columns is more than 48, which is number of papers about sorting. Not surprisingly, the most mentioned reason is, better timing. If we recall that, in RQ 2, the answer for most of the papers was speed-up achieved. It is safe to say that, most desired benefit from a parallel language is executing code in less time.
The other less common reasons are memory efficiency, load balancing, and throughput. However, these three properties are far more important than decreasing the execution timing. Because parallel languages are not all about speeding up the execution, but doing it while working on very large data structures. Consequently, any code, which was being executed in competitive time with a small program, could fall behind a single threaded language if the program size grows.

Figure 14 shows the choices of parallel languages in all papers in our pool of papers, grouped by the frequency of the name occurrence.
In the pool of papers, in 17 papers in which CUDA language is used, describes a sorting algorithm. All of the 7 papers OpenMP is the programming language, a sorting algorithm is described. Finally, 8 out of 10 papers where MPI is used as the parallel language, a sorting algorithm was described.

2.2.2.2.12. Summary of Research Questions

Answers to RQ1 revealed that most of the authors do consider a parallel language as the primary language when structuring a whole program. Although, this was already done with the shared memory languages, it seems constructing the whole program in parallel is getting a trend in data-level parallel languages too. Therefore, it is clear that, future researches in this area will be more comprehensive. Meaning that, programs that are more complex will be studied, instead of toy programs or just algorithms. RQ2 revealed speeding-up the execution time is the most desired situation. This is just fine by now because most appealing feature of data-level parallelism is achieving high throughput in simple arithmetic operations. However, some parallel languages are harder to code (e.g. CUDA, OpenCL), it is better to take in account other metrics that affects program behavior before switching to these languages. RQ3 considers the difference between single threaded and multi-threaded languages. Consequently, most common answer is performance gain. Therefore, efficiency in terms of scalability, memory, and bandwidth usage is not enough for some papers. In majority of them, it is claimed that performance (i.e. decreased execution time) is more important than others are. This is somewhat acceptable, because again most of the programs were small (i.e. algorithms, toy programs), so other measurements than measuring wall-clock times is almost impossible. Actually, paper [60] (where I am involved as one of the authors) states that with small test data size, it is not feasible to measure the effect of time complexities of the algorithms in wall-clock time, and then we suggest using a more advanced GPU. This is also an answer for the RQ4, where we are seeking to find a relation between hardware and increased benefit from using parallelism. There are some 48 papers, in which it is answered hardware relation explicitly as yes. RQ5 considers the testing and debugging issues. Inappropriately, none of the research papers addressed to these issues. However, this is the same with our paper [60], and it is decided to omit the testing in our paper to keep the focus on description of the proposed algorithms. This may be the same reason, why so many other papers are missing the
testing/debugging subjects. RQ6 seeks to find if the stability of the program or maintainability of the code is considered. Regrettably, it seems to be omitted in the papers to keep the focus on the main subject (i.e. proposing a new algorithm). A different reason, that may have led us to finding less answers than we expected, is programs being so small (i.e. probably they are just sample codes) that they would not need maintaining in future. RQ7 mentions the difficulties to code with a parallel language. It seems that, shared memory languages are easier to refactor in to parallel. Because of that, this thesis compares data-level parallelism (i.e. CUDA) to shared memory (i.e. OpenMP).

2.2.2.3. Research questions for this thesis

Literature survey in the previous section (i.e. Section 2.2.2.2) shows us nearly all papers in the literature, misses almost the same subjects. Therefore, four more research questions are decided to be added to this thesis to prove the importance of these subjects. The answers will be given in the Chapter 7. These subjects were:

- **RQ 1 - Initializing the data set** (Is it possible to test all ranges of data input. Is it possible to fill the data set using a parallel language?):

  Most of the papers proposes a new algorithm, or mentions an algorithm derived from an already existed one. Nevertheless, none of them mentions initializing the input data that is going to be used for the code. Actually, this is the most important issue for a sorting algorithm. Since, for most of the algorithms, the input data could change code behavior (i.e. stability), or even cause the program to crash for a certain input range. As a result, any type of research paper that was proposed an algorithm and did not give any information about initializing the input set, is missing an important part for the research which could be done with all their knowledge quite easily. In another words, the initialization is seemed too easy for them that it is omitted in the papers. However, the research quality is decreased in those papers because of that.
- **RQ 2 - Memory type used** (only for CUDA) (Is there a gain of benefit using different types of memory instead of global in CUDA for a sorting algorithm):

  Generally, CUDA programs utilize more than one type of memory. There was a couple of papers in previous section, which mentioned the relation between different memory types in CUDA and other languages. However, the new CUDA dynamic parallelism restricts the use of local memory to backwards compatibility (with the older GPUs) \[22\]. Moreover, dynamic parallelism brings new ways to access data that resides on the system memory. Therefore, it is better to mention the new model in this thesis. Finally, in Chapter 7 the gained benefits from this new memory model will be explained with examples.

- **RQ 3 – Scalability of threads** (Does the used algorithm, arises or solves scalability issues for possible number of threads that can be used concurrently?).

  In the previous section, we observed that some of the papers mentioned the scalability issue. This is where the actual number of threads, which can be used concurrently, is more than the algorithm allows without a slow-down (in the measured wall-clock execution time). The same issues are happen to exist in other shared memory systems, including OpenMP, and MPI. Therefore, a section is present in Chapter 7, for examining the algorithms used in this thesis.

- **RQ 4 – Testing the output for correctness** (Is there a way, possibly in parallel, to test the output for validity.)

  In the papers, about parallel languages, the results show a significant order of speed-up in favor of the parallel code. This means testing the parallel code, with sequential test code, may require a very long time. Therefore, coding the test cases in parallel languages is added to this thesis.

  Later, in Chapter 7, we will be seeing some interesting facts about the difficulties occurred while coding the test cases (in parallel).
2.3. Conclusions for Chapter 2

In this chapter, firstly, literature background is given, and then a small literature review is made. These studies show us the trends, strong points and weaknesses in the literature of parallel computing. Strong point in the literature is every paper gives the same metric for testing the achievements. However, we concluded that doing so might not increase the academic value of the paper. For this reason, we added Sections 7.2-7.7 to this thesis, to give enough information to recreate the test made in this thesis.
CHAPTER 3

GPU ARCHITECTURE

The advances such as the programmable shaders, made it possible to share instruction and cache memories in GPUs. These shaders are grouped in a way that it is called the SMs (streaming multiprocessors); this model is also called the Tesla® architecture. Figure 15 shows CPU and GPU component view in an abstract level. A CPU has limited number of cores, however larger controller and larger on-chip cache, making CPU threads heavyweight. With the Tesla architecture threads, barrier synchronization and atomic operations made the GPU programming into a more generic model. Efficient threading support enabled the more fine-grained data-level parallelism, where each part of an algorithm is programmed in parallel, as a substitute of the traditional task-level coarse parallelism.

In addition, scalable programming model for CUDA allows programs to span compatibility over a wide-range of GPUs available on the market. In other words, block and thread hierarchy of CUDA programming allows scalable CUDA programs...
Consequently, 4 blocks are assigned to each SM for processing. The GPU on the right has 4 SMs, than each SM has only 2 blocks assigned per SM. The assignment process is done automatically by the GPU, and currently it is not programmable. Conversely, there is a hard limit of active blocks on current architecture, which is 16 blocks per SM (i.e. 16 blocks for Fermi Architecture, current Maxwell architecture has a limit of 64 blocks for each SM). In a sense, the GPU on the right will have better performance for larger data sizes, because it can have up to 64 blocks active whenever possible.

3.1. Data-level Parallelism vs. Task-level Parallelism

There are other types of parallelism; hardware level, thread level, task level, data level and instruction level. In this thesis, data-level and task-level examples are used with CUDA and OpenMP respectively. Instruction-level parallelism is the machine setting the out-of-order execution of the instructions. Thread-level parallelism is in the form of computer-clusters or a single CPU with multi-threading capability (e.g., Intel’s hyper-threading). Finally, hardware-level parallelism is cloning a single core many times on a single die (e.g. Intel’s core processors).
Data level parallelism uses a unique combination of SIMD (Single Instruction Multiple Data) architecture and data pipelines, which is called SIMT (Single Instruction Multiple Threads). Data level parallelism happens when cores on a single CPU, or many CPUs of the same type (i.e. shared memory model), process different parts of a data. Likewise, GPUs have the same kind of architecture, where there are SMs and many cores on those SMs reflects a similar behavior when doing data-level computations. However, on many occasions, the code should need refactoring to exploit the data parallelism on the GPUs.

On the latest OpenMP version, Version 3.0, a special pragma with desired chunk size selects how data on an array is divided and then processed by the available processors to OpenMP. However, it is still not a match for the data-level parallelism.

Task-level parallelism happens when execution processes are divided among many CPUs or many cores of a CPU. Unfortunately, it is not possible with GPUs for now, because GPU threads are very lightweight compared to CPU threads.

![Figure 17 Simple, data-level, vector addition](image)

Figure 17 shows, a simple data-level addition where each index of vector A is added to same index of vector B and the result is stored to the same index of vector C. This type of calculation gives best results with CUDA language. For example, on a latest SM 5.0 version Nvidia GPU there can be up to 20.000 of threads active at the same time. That means, if the above vectors are shorter than 20,000, the whole
computation time will take the time needed for a single addition operation to complete [60].

3.2. Warp

A warp is the group of threads, 32 threads and 32 times the powers of 2, that are executed on an SM. Since the numbers of threads that can be active on each SM are limited to 2048, at most 64 warps can be active on each SM at any time for current Maxwell architecture (Nvidia GPUs). Although, the warps are not the only way to manage a group of threads, it is for now the fastest way. For example, Algorithm 1 uses the warps, in Section 2.2.1 and it is almost 6x times faster than non-warped code.

3.3. Concurrency

Concurrency was the issue until the occurrence of the CUDA v.6.5; however, since then both the allocated regions on the GPU memory and the system memory are controlled by the GPU via a pointer. In other words, the explicit data copies between the GPU and CPU are not needed anymore. In addition, as long as there are no data dependencies on a GPU managed memory region via a pointer, that same pointer can be processed preemptively by CPU and GPU.

Preemption may also happen, if the input size is much larger than the GPU can activate at the same time (i.e. approx. 20,000 for a Tesla K20), which causes the number of blocks, to exceed the number of warps (i.e. 64 simultaneous kernels for each SM for SM 5.0 and above), that the GPU can process concurrently. Then all the remaining kernels is executed preemptively, see the Chapter 7.5 for further explanation.

Accordingly, the algorithm is the main reason to achieving a better performance in parallel languages. For example, using a strided data access of threads makes possible the use of warped operation (i.e. upto 64 blocks of 32 threads running concurrently) in CUDA. However, the same approach cannot be used, if the operations are data-dependent (e.g. in Chapter 6, the Algorithm 6(strided) vs. the Algorithm 5(recursive)).
3.4. Conditional Branching

Threads in a warp executes in parallel. However, if the kernel code has branch divergence (i.e. if clause, or if else clause) at any point, than the threads diverges to execute these paths, and the branches executed one after another, causing the operation to slow down.

In fact, branch divergence is a serious issue with the serial programming as well. For the reason, the unnecessary conditional paths should be avoided.

3.5. Occupancy

For different GPU architectures, there is a certain limit for number of the SMs residing on the GPU. Those SMs also has a limit of concurrently executing the warps. For example, the warp limit for each SM, currently is 64 (it is explained Chapter 7.5). Therefore, the occupancy means the limit of concurrently executing warps on a GPU [96], and it has calculated as the following. Occupancy = Active Warps / Maximum Active Warps, where the “Maximum Active Warps” is a constant value for the particular architecture of the used GPU and “Active Warps” is specific to the used algorithm.

3.6. Shared memory

GPUs have relatively small caches on-chip, similar to CPU caches but only in size of kilobytes. Shared memory allocated to each thread block, so every thread in a block can access to it. On some cases, where data size is small, and assuming the total number of threads used is low accordingly. The use of shared memory has overwhelming benefit in terms of latency, compared to use of global or device memory in its place. In reality, shared memory latency is 100x times lower than the global memory.

On the other hand, the heavy use of shared memory will lead to bank conflicts. In the new Maxwell architecture, the compiler decides the use of shared memory. For example in section 2.2.1 we told that, a third algorithm with much lower latency is possible, but then Nvidia CUDA profiler tests revealed the following data. The algorithm introduced as much as 250000 conflicts in total, for only 1700000 I/O operations. That many conflicts happened, simply for the reason that the algorithm involves use of only thousand threads for the whole computation. Nonetheless, each
thread computes thousands of operations instead of only one operation. The shared memory space stays constant until the lifetime of a block, afterwards the bank conflicts occurs in that algorithm. Since there are more operations for a thread to do, and there is only a limited memory space (i.e. shared memory). This operation took the one hundredth of the original index-by-index operation; however, keep doing this operation will lead to heavy register usage, also known as increased register pressure.

3.7. CUDA Memory model

CUDA threads can access data from multiple memory spaces during their execution. Each thread has private local memory; each thread block has shared memory visible to all threads, but only has the same life span of the block. All threads have access to the same global memory. Figure 18 shows the memory hierarchy in the thread-level.

![CUDA Memory Model Diagram](image)

Figure 18 CUDA memory model [7]

There are currently 64 registers allocable per thread and total 65,536 registers device wide. The allocation is done automatically by device for the current devices and total thread number and code algorithm are the defining factors for this allocation.
Even though the name and figure implies, actual local memory is resided off-chip, the name local implies locality on thread level. In other words, it is local in scope of each thread. Shared memory is mentioned in the previous section. Moreover, the global memory meant here, is host managed memory space that is only accessible by the device via memory copy operations for devices SM3.0 and earlier. For the devices with compute capability SM3.5 and later global memory is device managed via command `cudaMallocManaged()`.

### 3.8. Hardware model

The Nvidia GPUs are built in a way that SMs resembles a close relativeness to the CPU core; the actual number of SMs differs with each architecture. When a CUDA program is invoked, the numbers of blocks are automatically distributed among the available SMs according to their block limit capacity. With newer architectures (i.e. SM 3.5 and above) this increases the probabilities of executing more blocks concurrently.

#### 3.8.1. SIMT architecture

SIMT (Single Instruction Multiple Threads) is a derivation of the SIMD (Single Instruction Multiple Data). The difference occurs where SIMD has coarse instruction-level parallelism and SIMT has fine-grained thread-level parallelism. In SIMT, the performance improvements are gained with designing the algorithm according to the thread execution behavior. These are avoiding the branch divergence, and using the warp model whenever possible.

In CUDA model, a warp means, a team of threads grouped in 32 parallel threads where a SM manages the execution. The grouping process can be done manually with built-in tasks or else the SM warp scheduler choses the grouping process according to the algorithm. For this reason, the latter case can be less performance efficient if the warp limit cannot be achieved.

#### 3.8.2. Hardware multithreading

The needed resources, during the lifetime of an active warp of threads, are allocated and are managed by a warp scheduler on an SM.

In practice, every SM has a total of 64KB of memory available for partitioning among warps, active blocks and shared memory. Therefore, there is a hardware limit
on active threads, which can reside on a GPU at any time, and it must be considered when designing the kernel functions. The total number of warp limit is: 
\[ \text{ceil}(\frac{T}{W_{\text{size}}}, 1) \]
where T is the thread number per block, Wsize is 32, and ceil(x,y) is round-up to nearest multiple of y.

### 3.8.3. CUDA parallelism

Success of achieving the parallelism in CUDA depends entirely on using the full capacity of underlying hardware. Some of the factors that affect using full capacity are scalability, avoiding branch divergence, and using appropriate memory type. Above, Figure 16 shows the scalability of blocks and threads according to the capabilities of the GPU used. In the meantime, different architectures have different SM count and different block execution capabilities, it is a good practice to design an algorithm according to the architecture used. A guide is available to users from the Nvidia developer website [14] and this tool calculates best threads per block ratio for users according to their GPU architecture.

An additional factor to exploit parallelism is avoiding branch divergence. In addition, executing the blocks in block warps, which is discussed in previous section, also helps in obtaining better performance.

Moreover, the latest advancement (i.e. CUDA dynamic parallelism) enables the automatic thread management by using special hardware on the GPU. That means, some constraints for coding CUDA kernels do no longer exists. This will help users to exploit parallelism more easily. Although dynamic parallelism introduced in the recent architectures, it is the most convenient way to achieve parallelism.

### 3.8.4. Dynamic parallelism

CUDA dynamic parallelism is only supported with the new architectures, as the needed hardware support only available with those devices. Dynamic parallelism brings CUDA agility writing CUDA code and brings support for memory management and compiler stack management issues. However, it is more of related to the CUDA programming; therefore, a more advanced topic will be covered in the next chapter of this thesis.
CHAPTER 4

CUDA PROGRAMMING MODEL

The latest CUDA programming architecture is based on “dynamic programming” and GPU managed variables or pointers. Although, these concepts are only applicable where an SM3.5 architecture GPU is available and CUDA programming language 6.0 and above installed.

In this chapter, both concepts will be introduced side-by-side to give the readers idea about the recent changes in the CUDA programming architecture. Consequently, architecture based on the devices SM3.5 and earlier will be mentioned as before, and SM3.5 and later will be mentioned as the current programming architecture.

4.1. CUDA execution model

CUDA language previously based upon CPU being the main processor and the GPU being the co-processor in the program execution model. That was a necessity, because simply GPU devices cannot manage the memory space allocated on the main memory. Besides, the programming model was not capable of executing the C programming model entirely. Figure 19 shows the CUDA execution model where the program executes on the main thread managed by the CPU. The parallel kernel code is executed on the GPU and when parallel part of execution of the code finishes, the execution is deferred back to the main thread of CPU.
4.2. CUDA syntax

4.2.1. Function declarations

Table II shows the keywords for C compiler to distinguish between a traditional host function and the CUDA kernel function. The second and third columns show the function caller identification. For example, in early CUDA versions the kernel functions were only callable from the host code. However, this is not the case for the devices with the SM 3.5 capabilities and higher. That means a kernel function can launch another kernel function now.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Executed on the</th>
<th>Callable from the</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void kernel</td>
<td>device</td>
<td>device / host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

4.2.2. Variable declarations

CUDA language has also some additions to the variable declarations.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Location</th>
<th>Accessibility</th>
<th>Lifespan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>Global memory</td>
<td>All active threads</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td>Constant memory</td>
<td>All active threads</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>Device registers</td>
<td>Threads from same block</td>
<td>Block</td>
</tr>
</tbody>
</table>
There is also “__managed__” with the current architecture, which indicates the GPU will manage the address space allocated for the variable and it is callable from both host and device.

4.2.3. Return types

Every CUDA built-in function returns the error code, *cudaerror_t*. Although there are other built-in library functions to handle the return codes, users can create their own handle functions as well. It is then programmers’ responsibility to collect the data about any fault caused by the code.

4.2.4. Threads, blocks and grids

Every CUDA kernel must be invoked with at least one block and one thread. If the algorithm is designed according to the CUDA model, then more blocks and more threads can execute the data. This means, more hardware is exposed to the execution, which is called occupancy.

![Figure 20 Thread - block hierarchy](image)

Figure 20 shows the thread execution model for CUDA, in the figure each block seems to have only 8 threads but this is only for demonstration purposes; normally it is at least 32 times the 2’s powers. As the figure shows, every thread in a block executes the same data. This is the SIMT model, already explained in this thesis, where multiple threads execute a single instruction. Moreover, the number of blocks (block limit is 2.5 billion in SM 5.0 above) that resides on the same SM, also called a grid.

4.2.5. Execution of threads from hardware point of view

Execution of threads bind to a special ALU called the CUDA cores; every Nvidia GPU has plenty of them. These CUDA cores reside in the SMs and then managed by the sources available on the particular GPU. Figure 21 [15], shows the
relation of each thread to the underlying hardware. Small boxes with SP written on them are the streaming processors (or the CUDA cores), grey boxes indicates the SMs. Since, every architecture has a different SM count, designing the algorithm (block and thread relations) according to this model, helps gaining more performance.

**Figure 21** Thread - hardware execution relation [15]

### 4.3. CUDA dynamic parallelism

CUDA dynamic parallelism is the name given to the new programming model added on top of the traditional capabilities of the CUDA language.

**Figure 22** Dynamic parallelism improvements [98]

Figure 22, shows the improvements the dynamic parallelism brings to the CUDA programming. Dynamic parallelism is most useful where a batch of *kernels* needs to be launched one after another, and when the *grid size* for the kernel to be launched is
unknown before runtime. (i.e. a recursive function, where recursion is solely dependent on the data processed in the runtime).

In addition, library calls within the kernels are now possible. Consequently, using printf for debugging a kernel or using built-in tasks is all made possible with dynamic parallelism.

Moreover, the GPU now manages the allocated memory, and there is no need for explicit memory copying from GPU memory to global memory. An example, for the implicitly GPU managed memory model is present in our code examples in this thesis, and can be found in the Appendix Section.

4.3.1. Parent-child execution model

With dynamic parallelism, child kernels can be launched from the parent kernel. A parent kernel is still host launched, as with the previous architectures. However, in the new model a kernel does not return the control back to CPU, to launch another kernel. Instead, the desired operation is carried from within the currently executing kernel, which is called a parent kernel. Therefore, simplifying the programming and bringing faster execution with eliminated memory-copy operations.

4.3.2. Heterogeneous computing

Using dynamic parallelism, the GPU can allocate memory and manage launching of multiple kernels without changing the hierarchy of launches. This behavior, speeds-up the program execution because kernel launches made from another kernel uses same resources as the parent and for this reason they launch faster than a host-launched kernel.

The tests made in this thesis show that there is a limit to dynamically launched kernels. That is when the input vector is large enough; the resources for the kernel launches are depleted. When the depletion occurs, the dynamic parallelism is not as fast as it was while processing less resource exhaustive data. However, the advances in the market show that this behavior is expected, as a result the newer GPUs are produced with improved hardware capabilities.
4.4. CUDA memory model

Current memory model uses the dynamic parallelism, where GPU manages the memory allocated in the *global* memory. This means, expensive memory copy operations are not needed any more, making the coding easier and program execution faster. An example is present in the Appendix B of this thesis, where Nsight Virtual Profiler shows, how the memory operations are automatically managed by the GPU while the program runs.

4.4.1. Unified Memory Architecture in CUDA

Unified memory architecture, in CUDA language simplifies coding efforts by enabling the memory allocations to be made and managed from the GPU, instead of the old model. That was deferring back the control to CPU for copying the memory content. In addition, UMA brings an ease to writing *C structures*, in the old model these structures were allowed, although the need for deep-memory-copy operations was causing extra steps of copy operations. With the advance of UMA the structured data can now be used with no performance decrease, with zero-copy. In other words, the host memory (i.e. RAM, or system memory) can be accessed directly by the device.
CHAPTER 5

PARALLEL SORTING

Sorting algorithms are used for ordering elements in an array; the most conventional way is using alphanumerical ordering. Another use of sorting is increasing the human readability of an output. Most common sorting algorithms such as quicksort, selection sort, insertion sort and merge sort used in this thesis is to reflect the behavior of the different algorithms in a parallel language.

5.1. Research Method

The sorting algorithms based on comparisons need at least one compare, and one swap operation. Additionally, more than one array-to-array deep copy operations needed for a merge sort. All of these arithmetic operations are already very expensive in terms of computational time for a sequential code. Nevertheless, for parallel sorting those kinds of arithmetic operations might be a complete killer of any benefits that could have earned from a parallel implementation.

Parallel sorting is also considered in many other researches in the context of different parallel architectures. General organization of some of the basic sorting algorithms for multithreading is considered in [92]. A parallel bucket-sort algorithm, which is presented in [93] requires $O(\log n)$ time with the use of $n$ processors. A pipelined insertion sort for sorting n numbers with n processes using MPI is given in [94]. In the same, an inherently parallel sorting method, namely, Bitonic sort is discussed which implements effectively in shared memory architectures.

For the GPUs, efficient strategies for parallel radix sorting on GPUs are discussed in [95]. In a technical report by NVIDIA Corporation [10], radix sort and merge sort algorithms are implemented in multicore GPUs using the CUDA language.
5.2. Sequential sorts

Sequential sorts are designed for single threaded computations. Consequently, the algorithm determines the execution times or memory space used for the sequential programs. Later, in Chapter 6, we will discuss a parallel quicksort, and a merge sort. Both of them use the selection sort for an array size less than a threshold value, in other words they are hybrid sorts.

Table IV Theoretical complexity values for sequential algorithm

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Best case</th>
<th>Avg. case</th>
<th>Worst case</th>
<th>Parallel prediction – p=#of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2/p)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2/p)$</td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n\log n/p)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n/p)$</td>
</tr>
</tbody>
</table>

Table IV shows the computational complexities for sequential code as well as the predicted parallel computational complexities. By being the divide and conquer type algorithms both quick and merge sort are highly parallelizable. Meanwhile, partitioning the input data lessens the probabilities of occurrence of data-dependencies, which is the most suitable type of operation for data-level parallelization.

The C language stdlib has a built-in sorting function called the qsort, which implements the use of quicksort for the operation. The function is explained in the paper [13] in 1993, by developers of qsort function. In this section, we will use a very similar approach, which takes an array with $n$ numbers and recursively sorts with quicksort if the array size is larger than $DT$ numbers where $DT$ is the threshold value. When the recursion reaches smaller than $DT$ numbers, sorting is deferred to selection sort. This is necessary because the quicksort has data overhead costs that affect the execution times heavily.
Consider the above partition table formed after the \( n \)th iteration of the hybrid sort. A, B, and Z are represents sub-arrays, and they are all the same size. Then the formula is, \( \text{sizeof}(A) = DT \), where size of DT is chosen manually at any positive value between 0 and the array size. In the following pages, Figures 24 and 27 shows the partitioning model for quicksort and merge sort.

![Figure 23 Time values for HB quicksort vs. built-in](image)

**Figure 23 Time values for HB quicksort vs. built-in**

Figure 23, shows the comparison of timing values for C language “stdlib” library’s built-in sort. In addition, hybrid quicksort algorithm (i.e. HBquick) is one of the sorting algorithms assessed throughout this thesis. The algorithm (CUDA based version) is explained in Chapter 6, Algorithm 8 and Algorithm 9. When both `qsort` and HBquick (CPU) are run with the same input array size, the stdlib’s quicksort is slower. In addition, by comparing the HBquick (CPU) sort (hybrid quicksort with selection sort) and qsort time lines to the linear line (i.e. green line) one can tell that both algorithms have a computational complexity of \( O(n) \) (i.e. slope is 1 for lines). *Table IV*, given previously, shows the computational complexity of the quicksort and insertion sort is \( O(n\log n) \) and \( O(n^2) \), respectively. That means, both the built-in qsort and our function HBquick (CPU) sort has much better computational complexities. Therefore, the Figure 23 shows the reader, achieving a better performance is more related to the design of the algorithm. In this example, both functions are sequential (i.e. runs on single thread). In the rest of this thesis, we will be seeking further
performance increase using the parallel languages. By this, it is meant to gain extra performance where hard limit for decreasing the timing using a single-threaded execution is reached, even with a well-designed algorithm.

5.3. Traditional Quicksort

Tony Hoare, a visiting student to Moscow State University, is the developer of the quicksort algorithm in 1959 [97]. The main idea is dividing a large array into smaller sub-arrays then recursively solving the smaller sub-arrays. Keeping recursive code design but utilizing a partitioning algorithm obviously helps to overcome the unwanted program termination due to reaching the stack size limit. For the same reason, it is also suitable for data-level parallelism, where both stack size (i.e. GPU memory is limited) and concurrent thread numbers are a consideration.

5.4. Hybrid quicksorts

This section gives details about our algorithm HBquick (CUDA), where a parallel quicksort algorithm used with a parallel selection sort algorithm to create parallel HBquick (CUDA) algorithm.

![Figure 24 HBquick (CUDA) sort execution model](image)

Figure 24, shows the idea behind the hybrid sort. Recursive quicksort takes an n number array and partitions the array until reaching threshold then the selection sort sorts the smaller sub-array, and this operation is repeated until the entire array is sorted. Most of the execution speed-up occurs at this point, because the divide-and-conquer algorithms are faster for large arrays. However, this advantage slowly fades
if the array size drops below a few dozens of elements. Therefore, it is best to use both together.

A more detailed explanation as well as the pseudo codes are present in the Section 6.

Figure 25 shows, the timing values for HBquick (CUDA) when the array to sort held at 400K constantly, but the defer threshold (DT for simplicity, which is the sub-array length to swap the sorting to selection sort) changes. It is clear from the graph that, up to DT = 4000, the execution time for the HBquick (CUDA) decreases. That means, affective value for 400K numbers is $4 < DT < 4000$. Moreover, after DT=4000, the performance benefit gained is ceased. Then after DT=4000 the effect reverses (i.e. it slows down the execution, instead of speeding up), this is where the time line is increasing. At DT = 4, the function is slower when compared to the DT values such as 40,400, and 4000 simply because the value 4 is too small to effect the hybrid sort’s execution time.

The results show us in CUDA there is a hardware limit to data-level parallelism. This limit is both related to the code design (i.e. algorithms) and CUDA architecture (i.e. depends on the particular GPU that is used). Since, the architecture is an issue of hardware; in this thesis, we are focusing on the algorithm, to seek feasible choices for better performance.
5.5. Comparisons for the sequential algorithms

Figure 26 shows, selection sort, bubble sort and the insertion sort timings for array sizes consist of two’s powers. The results show a linear increase in all algorithms, which means the timing values, does increase in $O(n^2)$, as the computational complexity implies. For example, for bubble sort array size increases from 4096 to 8192 and time values are 0.443 and 1.76 respectively. Thus, $O(n^2)$ increases in time and computational space. HBquick(CPU) sort (i.e. single threaded version) is given to demonstrate that not all algorithms are slow on CPU.

It is clear from the graph, HBquick, which is also a C language based function that runs on CPU obtain the fastest time. The speed-up achieved with respect to selection sort (CPU) is up to 8000x in wall-clock time.

Therefore, it is proven that any optimized algorithm can result in a speed-up in time almost as much as a parallel language. This should be taken into consideration that it is one of the main subjects of this thesis. Where by comparing parallel-to-parallel language and by comparing parallel-to-sequential language, the most contributing factor is sought to decrease the execution timings. Then, we can understand if the deriving factor for the speed-up to occur is the algorithm or hardware capabilities.
CHAPTER 6

IMPLEMENTATION

This chapter is prepared for introducing the reader to the algorithms that are used in this thesis for wall-clock time comparisons. By doing so, it is expected to give enough means of reason to the reader to be convinced into considering the algorithm is a major contributor for achieving better performance from programming languages as well as the parallel languages.

Table V Description of all Algorithms used in this thesis

<table>
<thead>
<tr>
<th>Name of algorithm</th>
<th>Description</th>
<th>Code sample/Code present in thesis</th>
<th>Available in CUDA</th>
<th>Available in OpenMP</th>
<th>Available in C code</th>
<th>Built-in (B) or generated (G) code for this thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBquick</td>
<td>A parallel hybrid sorting algorithm with quicksort and selection sort</td>
<td>Pseudocode (Algorithm 8) Code sample in Appendix C</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>G</td>
</tr>
<tr>
<td>Hybrid Merge Sort</td>
<td>A parallel hybrid sorting algorithm with merge sort and selection sort</td>
<td>Pseudocode (Algorithm 3-7) Code sample in Appendix C</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>G</td>
</tr>
<tr>
<td>qsort</td>
<td>C language stdlib’s built-in sort function</td>
<td>Code sample in Appendix C</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>B</td>
</tr>
<tr>
<td>Selection sort</td>
<td>A parallel sorting algorithm used in hybrid merge sort hybrid quicksort</td>
<td>Pseudocode (Algorithm 7 and 9) Code sample in Appendix C</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>G</td>
</tr>
<tr>
<td>Merge sort</td>
<td>OpenMP based parallel merge sort algorithm</td>
<td>Code sample in Appendix C</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>G</td>
</tr>
<tr>
<td>cdpSimpleQuicksort</td>
<td>CUDA SDK official sample (quicksort + selection sort)</td>
<td>Free to download from [3]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>B</td>
</tr>
<tr>
<td>cdpAdvancedQuicksort</td>
<td>CUDA SDK official sample (quicksort and bitonic sort for larger arrays)</td>
<td>Free to download from [3]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>B</td>
</tr>
</tbody>
</table>
Table V, above, shows the sorting algorithms used in this thesis. The first column gives the algorithm name, second column gives short description, third column gives if the pseudocode/code is present in written form, fourth to sixth columns give the language versions available, and last column gives information if I write the code or a built-in function that is present to public use. The availabilities columns indicate that there is a code or pseudocode present in this thesis, and it does not mean the particular algorithm is not applicable in that language.

The hardware environment and used software is summarized as following. Intel® Core™ i7-5500U CPU @ 2.40GHz, Nvidia® GeForce™ 850M GPU @ 1.0 GHz with 4 GB (DDR3) memory, 16 GB (DDR3) system memory, Windows 10 operating system, Nvidia Nsight™ Visual Studio™ Edition (Visual Studio 2012), Nvidia CUDA Version 7.0, OpenMP Version 3.0, Microsoft C developer pack.

6.1. Hybrid merge sort (CUDA)

In this section, a parallel version of HBquick sort in CUDA is compared to a parallel implementation of the hybrid merge sort, in sequential form, in OpenMP and in CUDA. A merge sort is discussed in [27, 34, 47, 72] and it is said to be fastest compared to other approaches to sorting. Although this thesis is the first time where a hybrid merge sort with CUDA dynamic parallelism is introduced, as to the best of our knowledge.

Figure 27, below, shows the description of hybrid merge sort (in CUDA), the split function recursively partitions the input array in sub-arrays with sizes equal to the defer threshold. When the left-most sub-array is reached (i.e. sub-array[first-index] = input-array[0] and sub-array[last-index]=input-array[defer threshold]) the split function calls the parallel selection sort (coded with OpenMP) function. The split function then repeats the operation for the second sub-array (i.e. sub-array[first-index] = input-array[defer threshold] and sub-array[last-index]=input-array[2*defer threshold]). Then, the resulting sub-array is merged together. This operation is carried until there is two final sub-arrays exist where joining these two gives the initial input array, but its elements in sorted order.
Figure 27 Hybrid merge sort

Defer threshold limits the split operation to a more meaningful value, where neither selection sort overloaded by a large array size nor the merge sort have to merge sub-array sizes starting from 2 numbers for a sub-array.

Algorithm 3 Split function for hybrid merge sort

1: if (sub-array > defer threshold)
2:    split (sub-array)
3:    merge sort (sub-array)
4: end if
5: else
6:    selection sort (sub-array)

Algorithm 3 shows a very similar algorithm to traditional merge sort, only with added code path to call an external selection sort function. However, the actual
implementation of the split function is made with the CUDA dynamic parallelism, which makes it one of a kind, at least for now. The code sample is present in the Appendix Section.

6.2. OpenMP implementations

CUDA compiler has built-in OpenMP support. In this thesis, OpenMP supplied by CUDA developing package is used. This means, OpenMP code compiles with CUDA compiler, and can be used as a stand-alone language or can be used alongside with CUDA.

Since, the OpenMP uses the C language as a base language, a very similar algorithm to sequential code above used for the OpenMP as well. This means, for OpenMP parallelization algorithm change is unnecessary.

The algorithm 4 uses a similar approach as discussed in [65,78,91]. However, our algorithm differs from the other recursive function based algorithms significantly. Mainly, it is tested and optimized to eliminate all redundant code paths that may take place in parallel languages. The code in this section follows the rules of OpenMP standard version 3.0. Again, the complete C code is presented in the Appendix of the thesis.

Algorithm 4 Split function for hybrid merge sort in OpenMP

1: pragma omp parallel sections
2: pragma omp section
3: if (sub-array > defer threshold)
4:    middleIndex = size_of(sub-array)/2
5:    split (sub-array, firstIndex, middleIndex)
6:    split (sub-array, middleIndex, lastIndex)
7:    merge sort (sub-array, firstIndex, lastIndex)
8: end if
9: else
10: selection sort (sub-array)
6.3. CUDA implementations

There are two algorithms used in this thesis, they are HBquick and hybrid merge sort. HBquick uses CUDA dynamic parallelism, which is described in Chapter 5. However, first the implementation of hybrid merge sort.

Algorithm 5 Split function for hybrid merge sort in CUDA

1: if (sub-array > defer threshold)
2:   split (sub-array)
3:   thread_num = defertreshold
4:   merge sort <<<blocks,thread_num>>>(sub-array)
5: end if
6: else
7:   selection sort (sub-array)

Hybrid merge sort uses the CUDA dynamic parallelism (i.e. GPU managed memory), described in Chapter 5, which enables heterogeneous computing. This means, at any step of computation the control of the memory region should deferred back and forth between the CPU and GPU, without a need for explicit memory copies. That means, without the presence of time expensive copy operations, there is a speed-up occurs in program execution. In fact, using OpenMP based selection sort, proved faster when compared to using a CUDA kernel selection sort. Subsequently, according to CUDA programming model (Chapter 4), parent and child kernels share the GPU resources. In here, merge sort’s split function is the parent kernel, and selection sort is the child kernel.

Therefore, with an external sorting function instead of the parallel selection sort (i.e. in CUDA), there will be more resources left for the parent kernel (i.e. merge sort’s split function), which will speed-up the execution.

Algorithm 6 Merge sort function for hybrid merge sort in CUDA

1: n = sub-array size to sort
2: i = 0, j = n / 2
3: k = unique_thread_number
4: stride = block_dimension * grid_dimension
5: while ( k < n )
6:   if ( j == n) dummy_array[k] = original_array[i++]
7:   else if ( i == n / 2 ) dummy_array[ k ] = original_array[ j++ ]
8:   else if (original_array[ j ] < original_array[ i ] ) dummy_array[ k ] =
9:   else dummy_array[ k ] = original_array[ i++ ]
10:   k += stride
11: endwhile
12: synchthreads()

Algorithm 6 above shows the pseudocode for the CUDA version of the merge sort function of the CUDA hybrid merge sort code. The while block incremented with a stride value instead of 1. The reason for that is explained throughout this thesis (e.g. Algorithm 1). Although the Algorithm 6 seems to carry a strong resemblance to the legacy code, the version used in thesis (i.e. CUDA based, coalesced thread access version), is the first implementation in the literature, as to the best of our knowledge.

Algorithm 7 below shows the pseudocode for the selection sort algorithm. The algorithm is redesigned for this thesis to provide a more suitable algorithm to work in OpenMP. At the beginning the algorithm was written with the OpenMP private first and private last pragmas, however the function became unstable with some input range, and program crashed. Then another version of the selection sort algorithm is designed with #parallel tasks pragmas, however the data dependencies of the sorting algorithm caused erroneous output. Finally, the algorithm took the form as seen in the Algorithm 7, in which idx is the private variable to each thread, and other remaining variables are shared with threads available to the OpenMP. The algorithm here is a combination of selection and insertion sort, but with closer relativeness to selection sort.

---

**Algorithm 7** Selection sort (in OpenMP) function for hybrid merge sort in CUDA

1: int temp=0, p=0, idx=0
2: #pragma omp parallel shared(a,n) private(idx)
3:   for(idx->0 to n)
4:     temp = a[idx]
5:     p = idx
6:     while (p>0 && a[p-1] > temp)
7:       swap(a[p-1], a[p])
8:     endwhile
9:     a[p] = temp
10: endfor
6.4. Parallel hybrid quicksort

Parallel quicksort, is a hybrid sort with use of quicksort and selection sort to run on GPU. Its role in this thesis is to give more examples about CUDA programming model. Both quick and selection sorts are present in the Ali et al. “Implementation of Sorting Algorithms with CUDA: An Empirical Study” [60]. For this thesis, they are used together to create hybrid- quicksort. The implementation of our hybrid quicksort with new CUDA dynamic parallelism is first of its kind in the literature, as to best of our knowledge.

Hybrid quicksort has better timing, meaning better computational complexities than both of these algorithms are timed separately (i.e. quicksort or selection sort, separately). In Section 5.3 Figure 24 shows the inner workings of the HBquick, that is the same algorithm discussed in here.

The quicksort recursively calls itself, until DT (defer threshold) is reached. Selection sort sorts the resulting sub-arrays and quicksort merges the smaller sub-arrays together. This approach creates the best case [60] input, for quicksort (i.e. the input array being already in the desired sorting direction).

Algorithm 8 Quicksort function from HBquick in CUDA (dynamic parallelism)

```
1: HBquick(array2sort, left, right)
2:   if ((right-left) < defer threshold)
3:     selection sort <<<subarray_size/dt, dt>>> (sub-array)
4:   endif
5: else
6:   pivot = sub-array[right]
7:   i = left
8:   j = left
9:   while(sub-array[j]<pivot)
10:     if ( i < j ) swap (sub-array [++i], sub-array [j++])
11:   endwhile
12: endif
13: swap (sub-array [i-1], sub-array [j])
14: HBquick <<<1,1,0,stream1>>> ( a, i )
15: HBquick <<<1,1,0,stream2>>> ( a, n – i )
```

Algorithm 9 below shows the pseudocode for selection sort. The algorithm ensembles the legacy code for sequential selection sort. However, this is the idea of
CUDA language that is keeping the original algorithm. The unique thread number (UTI) is calculated as (threadId.x+blockId.x-blockDim.x) and it is conventional in all CUDA programs. The stride also is a conventional way of keeping the threadIdx unique around many block warps. In line 10, stride added to i instead of 1, to keep the threadIdx’s unique. The kernel profiles from Nsight profiler in Chapter 7 will prove this.

In addition, the algorithm differs from the legacy code in using two swap operations instead of one, in this way; the break statement is eliminated from the code. Things to note here is the CUDA compiler does not give an error for neither CUDA nor the OpenMP when using a break statement. Nevertheless, the unit tests show the redundant code path, so the algorithm is altered accordingly.

Algorithm 9 Selection sort (in CUDA) function from HBquick in CUDA

1: \( i = \text{unique\_thread\_number} \)
2: \( \text{stride} = \text{block\_dimension} \times \text{grid\_dimension} \)
3: \( \text{while } ( i < n ) \)
4: \( \text{temp} = \text{sub-array}[i] \)
5: \( p = i \)
6: \( \text{while } (p > 0 \&\& \text{sub-array}[p-1] > \text{temp}) \)
7: \( \text{swap}(\text{sub-array}[p-1], \text{sub-array}[p]) \)
8: \( p-- \)
9: \( \text{endwhile} \)
10: \( \text{sub-array}[p] = \text{temp} \)
11: \( i + = \text{stride} \)
12: \( \text{endwhile} \)
Figure 28 HBquick sort timing when *defer threshold* is equal to array size to sort

Figure 28 shows the timing when HBquick threshold is equal to the size of the input array. For example, for 4096 elements to sort, DT is equal to 4096. Then it is clear from the graph even when most of the computation is made with selection sort, the program has a computational complexity of $O(n \log(n))$. That is equal to the HBquick which has a computational complexity of $O(n \log n)$, i.e. obtained from Figure 30. Therefore, using a hybrid sort in parallel languages has more benefit, instead of using these algorithms separately.
CHAPTER 7

TESTS AND COMPARISONS

In this chapter, the timing values of all algorithms mentioned in Chapter 6 are given for comparison.

7.1. Defer threshold and Block dimension relations for hybrid merge and HBquick sort

Both the hybrid merge sort and HBquick use sorting algorithms that are warped in blocks, such as $32 < \text{blockDimension} \leq 1024$ and $\text{gridDimension} = \frac{\text{total_array_size}}{\text{blockDimension}}$. Consequently, this section first gives the timing values for changing the blockDimension, then timing values for changing the threshold.

Table VI Time values for constant array size and changing defer threshold

<table>
<thead>
<tr>
<th>Array size</th>
<th>1024</th>
<th>1024</th>
<th>1024</th>
<th>1024</th>
<th>1024</th>
<th>4096</th>
<th>4096</th>
<th>4096</th>
<th>4096</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defer Threshold</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Hybrid Merge sort timing</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.16</td>
<td>0.13</td>
<td>0.09</td>
<td>0.08</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Table VI above shows that for small arrays changing the defer threshold up to the same size as the input arrays can affect run time positively. This behavior is explained in Chapter 6. Although, next table shows, this will change when the array size grows.
Table VII shows the timing for hybrid merge sort. First half of the table (i.e. left to
the grey bar) is where DT is held at constant and the thread numbers (i.e. block
dimensions) changed for merge function and selection sort function inside the hybrid
merge sort code. In the first part, timing values almost does not change with the
changing thread numbers, which proves the algorithm works as expected. In
addition, the kernel profiler shows there is little time difference, between executing
our functions with large grid dimensions and small block dimensions. In fact, CUDA
developers themselves suggest that timing values for launching the same global
function (i.e. kernel) with small grid dimensions or large grid dimensions should only
change in microseconds [99]. However, one should be aware that this holds true until
320 blocks for a kernel warp, after exceeding that point the GPU used for this thesis
cannot concurrently warps kernels, and schedule them. This phenomenon is
explained in detail with the Nsight profile of merge sort function in Sections 7.4 and
7.5.

Second part of Table VII shows the time values for increasing DT. The merge sort
(i.e. Algorithm 5) is a divide-and-sort algorithm, and the values prove the benefit of
using such an algorithm. Because, it is clear from the results that without dividing the
input array, the arithmetic operations, for selection sort to execute, overwhelms the
device limits. Therefore, the best result obtained is when either of the DT or the
block dimension is equal to 1024. This is (block size = 1024) where the warps/SM is
at its peak value of 64 for the particular GPU used in this thesis. Figure 35 in Section
7.5 explains this behavior more in detail.

In conclusion, there is a relation with the input array size and the concurrently
running threads for particular GPU. Furthermore, making an efficient use of it is
achieved with dividing the overwhelmingly large input array to smaller sub-arrays, i.e. hybrid merge sort.

7.2. Parallel sorts from this thesis vs. the other functions

In this section, the parallel and sequential execution time values of code generated for this thesis are compared. In addition, cdpSimpleQuickSort (i.e. an official CUDA SDK example) is used in the comparisons to give the reader more evidence when assessing the results.

In Figure 29, the execution time values (given in Table VIII) for HBquick sort are compared. HBquick in CUDA (i.e. Algorithm 8, in Chapter 6) version. HBquick in OpenMP (i.e. sample code given in Appendix C) version. Hybrid merge sort in CUDA (i.e. Algorithm 6, in Chapter 6) version. Hybrid merge sort in OpenMP (i.e. sample code given in Appendix C) version and cdpSimpleQuickSort official CUDA SDK example.

<table>
<thead>
<tr>
<th>array size</th>
<th>HBquick sort (CUDA)</th>
<th>HBquick sort (OpenMP)</th>
<th>hybrid merge sort (CUDA)</th>
<th>hybrid merge sort (OpenMP)</th>
<th>cdpSimpleQuickSort (CUDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>0.032</td>
<td>0.265</td>
<td>0.125</td>
<td>0.001</td>
<td>0.078</td>
</tr>
<tr>
<td>16384</td>
<td>0.031</td>
<td>0.469</td>
<td>0.281</td>
<td>0.002</td>
<td>0.1553</td>
</tr>
<tr>
<td>32768</td>
<td>0.062</td>
<td>0.453</td>
<td>0.578</td>
<td>0.005</td>
<td>0.315</td>
</tr>
<tr>
<td>65536</td>
<td>0.094</td>
<td>0.468</td>
<td>1.312</td>
<td>0.009</td>
<td>0.612</td>
</tr>
<tr>
<td>131072</td>
<td>0.188</td>
<td>0.484</td>
<td>3.77</td>
<td>0.016</td>
<td>1.213</td>
</tr>
<tr>
<td>262144</td>
<td>0.329</td>
<td>0.546</td>
<td>6.483</td>
<td>0.03</td>
<td>2.6512</td>
</tr>
</tbody>
</table>

Figure 29 Speed-up achieved against CUDA official sample cdpSimpleQuickSort

63
The graph, in Figure 29 shows that the hybrid merge sort in OpenMP is the fastest, also the slowest when the same algorithm (i.e. hybrid merge sort) is written in CUDA. This was an expected result, which is actually the main subject of this thesis (comparing shared memory to data parallel). The difference occurs using the same algorithm simply because CPUs are designed for latency hiding (i.e. fast where data is small), and GPUs are designed for throughput (i.e. fast where data is large).

The green line with triangle, hybrid merge sort, is the slowest, this happens not because there is a design flaw, but capability of the GPU used has a limit with this algorithm. Here, the latest CUDA dynamic parallelism is used but this model was introduced recently and still developing. Since, the number of threads in this GPU that can be concurrently used for dynamic parallelism is low at approximately 10,000 (i.e. 5 SMs x 2048 threads per SM). The calculation is quick at relatively low numbers. However, when thousands of concurrent kernel launches needed, the execution time increases. See Chapter 7.5 for details.

7.3. The quickest algorithms compared

In this section, cdpAdvancedQuicksort (from CUDA sample SDK), HBquick in OpenMP (i.e. sample code given in Appendix C) and HBquick in CUDA (i.e. Algorithm 8, in Chapter 6) versions are compared to HBquick (CPU), in wall-clock time values using the elements sizes ranging from 10M to 40M.

<table>
<thead>
<tr>
<th>Array size</th>
<th>HBquick sort (CPU) in sec.</th>
<th>HBquick sort (CUDA) in sec.</th>
<th>HBquick sort (OpenMP) in sec.</th>
<th>cdpAdvancedQuicksort (CUDA) in sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00E+07</td>
<td>3.91</td>
<td>3.983</td>
<td>0.985</td>
<td>0.953</td>
</tr>
<tr>
<td>2.00E+07</td>
<td>8.36</td>
<td>8.14</td>
<td>1.46</td>
<td>1.972</td>
</tr>
<tr>
<td>3.00E+07</td>
<td>13.16</td>
<td>12.344</td>
<td>1.938</td>
<td>2.959</td>
</tr>
<tr>
<td>4.00E+07</td>
<td>17.64</td>
<td>16.564</td>
<td>2.528</td>
<td>4.078</td>
</tr>
</tbody>
</table>
From Figure 30, we observe that HBquick (CPU) is the slowest, so other time values are compared against it. The CUDA version of HBquick has almost the same execution time with CPU version. Although, this does not means our HBquick sort algorithm causes GPU serialization, because in that case the whole computation executes with a few threads or a single thread and then it would be impossible to reach either of the array size or time value. In addition, if one compares HBquick speed-up with cdpAdvancedQuickSort then it is clear that the speed-up remains linear for increasing input. Therefore, it is safe to say that the time difference is only caused by the fact that the algorithm of the bitonic sort is a bit better than HBquick sort (i.e. $O(\log (n^2))$ versus $O( n \log (n))$, respectively).

Moreover, CUDA sample (i.e. cdpAdvancedQuickSort) is the second fastest function. However, it uses quicksort and bitonic sort together. A quicksort program takes a number as input size and creates an array of arbitrary numbers. If the input sequence is small (i.e. few thousands) the array is sorted with quicksort, although if the input array is more than a few thousands the program calls the bitonic sort immediately and finalize the computation with the bitonic sort. In other words, the computation is not deferred back to the quicksort if started with bitonic sort. Besides, bitonic sort invented by Ken Batcher, the pioneer of parallel computing, and the actual purpose of the use of this algorithm is given in his paper [86]. However, a bitonic sort takes in series of increasing and decreasing numbers, and the output should be in series, and not in just one increasing series of elements as the CUDA sample does. It is not
completely understood, why CUDA sample forms a single series of ascending numbers!

HBquick written in OpenMP, the purple line with asterisk, is the fastest. Therefore, the approach for designing a hybrid algorithm is proved useful, at least in the OpenMP language. Otherwise, this much of speed-up would not be possible with just 4 threads (i.e. CPU processors). The reason is partially covered in Section 7.5.2, where the partitioning model for HBquick (all versions) is given. To be precise, the small thread number in our CPU (i.e. four) is not overwhelmed with possible many small (i.e. as little as 2 in size) sub-arrays, where the sorting (i.e. when array gets partitioned to less than DT in sizes) is deferred to selection sort.

Therefore, the most performance-yielding algorithm is HBquick (OpenMP) in the Figure 30, if one considers the effort needed for altering per line of code for achieving performance. Although, CUDA language takes the first place, when one considers performance per watt and performance per unit cost. Moreover, the latter case is more important for the professional applications. Hence, it is proved to be practical to use a hybrid sorting approach against the traditional (i.e. defer at the end or beginning methods, where cdpSimpleQuickSort, cdpAdvancedQuicksort are these kinds of sorts, respectively) methods.

7.4. Functionalities of CUDA profiler

In Hybrid merge sort, default stream region shows concurrent and overlapping kernel executions. In the Figure 31, there are 4 CUDA kernels, these are split function,
The merge function, array copy function and selection sort. The red bars show the array-to-array data copy kernel. The Figure 31 also shows the partition idea of the algorithm, from left to right the operations grouped into columns and a red bar (final sorted array for particular partition copied from temporary memory to main array) links them. Then, each group grows, as the data partition size grows to end of the execution of the program.

In addition, Figure 37 in Appendix B shows the memory copy operation entirely managed by the GPU. Including the current CUDA model, every array is initialized in the global memory, and in the run time the desired portion of the array is copied back-and-forth between GPU’s discrete memory and global memory. By doing so, the memory I/O operations are clearly accelerated because there is no need for the transferring entire content of an array to GPU and from system memory. However, neither Linux nor Windows version of Nsight is able to trace this operation while a kernel uses dynamic parallelism. Consequently, a different version of hybrid merge sort is used for only displaying the memory operation according to the UMA model, since it is not shown by profiler in Figure 31.

Figure 32 Insertion sort profile shows warped operation
Figures 32 and 33 (larger images exist in Appendix B), shows the kernel profile for instruction execution, this is a menu directly accessible from Nsight profiler. This code is automatically profiled and shows line-by-line the optimization problems that might occurred from thread branch divergence or predicated off threads. In both figures, the code has no such problems, which means code needs no further optimizing and the program runs efficiently.
7.5. Nsight performance counters for merge sort function

The Figure 34 (larger image exists in Appendix B), below, shows the Nvidia Nsight kernel profiler results for hybrid merge sort program.

![Figure 34 Merge sort profile with size=300K and DT=1024](image)

The column names and their meanings are like the following. *Device name* shows the GPU model the program run on. In this thesis only one GPU is used, and model name is Geforce GTX 850M. *Blocks* means, block count used for executing the particular *kernel*, the actual profiler results has 550 *grid* (blocks x threads) launches. Then for simplicity, only one occurrence of each block count (i.e. grid) is included here. Because as it is proposed in this Section, a normal *kernel* function’s execution time should not change extremely, unless the particular GPUs block execution limit is reached. *Threads per Block* means, thread count for each *kernel*. In our function design, this number is the deterministic factor for block count, as such the \[ \text{Blocks} = \frac{\text{size_of(sub-array)}}{\text{Threads per Block}} \] threads per block is equal to 1024. *Threads* means, total number of threads used for executing the particular *kernel*. This is the reverse of Blocks formula, given above. *Duration* means, the execution time for each *grid* (i.e. blocks of threads) to run the kernel on the particular row. *Occupancy*
means, the percentage of GPU utilization predicted by the compiler at the compile time. The deterministic factors for the occupancy value are calculated from the values in Figure 35 that are compiler statistics for a particular kernel function.

Figure 35 Occupancy statistics for TopDownMerge kernel

Also in the Figure 35 (larger image exists in Appendix B), first graph (i.e. Varying Block Size) shows the maximum warp count that can be achieved using the block size (i.e. threads per block) for the particular GPU. The same graph means, if and only if the block sizes of 64, 128, 256, 512, 768 and 1024 threads are used, then the concurrent warps for the current kernel can reach to 64 warps (that is the architecture limit, and common to all GPUs of this era). The third graph shows the shared memory amount used. Moreover, the last graph shows the achieved occupancy for the particular kernel. Therefore, to achieve maximum occupancy the block size should be chosen appropriately to the GPU used.

There are two other columns to explain in the Figure 34. Achieved occupancy, means the actual usage percentage of the GPU for the particular kernel. Warps launched are the total number of warps needed to launch for the particular kernel. Warps launched have a value as such: Threads = Threads_per Block \times Blocks, Warps Launched = Threads / Warp Size, where Warp Size = 64 and it is determined from the first graph from the Figure 35, valid values are 32 through 64.
7.6. Limit to faster kernel execution: Warp launch count

In the previous section (Section 7.5), we studied the column name meanings in Figure 34, in this Section another kernel profile (i.e. CopyArray, a deep array-to-array copy function) will be given to explain the concurrent warp count limit to execute on the particular GPU (i.e. GeForce GTX 850M).

![Profiler stats for CopyArray kernel](image)

Figure 36 (larger image exists in Appendix B), likewise Figure 34, shows the limiting factors for the faster kernel execution. However, before going there, the table in the Appendix A.1 shows the capabilities of the GPU used in this thesis. One of the rows (i.e. Number of concurrently active threads) shows the number 10240, which is found by the equation (Maximum number of threads per multiprocessor x the SM count), then 2048 x 5 = 10240) for our GPU. This number also related to warp launched formula that is explained in Section 7.5. If one inspects the Figures 34 and 36 closely, then when the threads count reaches the value 10240 calculated above, the particular GPU is reached the limit of concurrently executing the kernels. After

<table>
<thead>
<tr>
<th>Device Name</th>
<th>Function Name</th>
<th>Blocks</th>
<th>Threads per Block</th>
<th>Threads</th>
<th>Duration (ms)</th>
<th>Performance Counters (1) - Warps Launched</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>2</td>
<td>1024</td>
<td>2048</td>
<td>17.888</td>
<td>64</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>4</td>
<td>1024</td>
<td>5072</td>
<td>37.44</td>
<td>96</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>5</td>
<td>1024</td>
<td>3120</td>
<td>17.503</td>
<td>160</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>10</td>
<td>1024</td>
<td>10240</td>
<td>27.072</td>
<td>320</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>20</td>
<td>1024</td>
<td>20480</td>
<td>47.359</td>
<td>640</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>40</td>
<td>1024</td>
<td>40960</td>
<td>89.888</td>
<td>1280</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>80</td>
<td>1024</td>
<td>81920</td>
<td>163.712</td>
<td>2560</td>
<td>100%</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>CopyArray</td>
<td>160</td>
<td>1024</td>
<td>163840</td>
<td>321.376</td>
<td>5120</td>
<td>100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Registers per Thread</th>
<th>Achieved Occupancy [ID: Active Threads]</th>
<th>Performance Counters [ID: Active Warps]</th>
<th>Performance Counters [ID: Active Warps]</th>
<th>Occupancy [ID: Occupancy Per Block]</th>
<th>Occupancy [ID: Occupancy Per Block]</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>16%</td>
<td>240782</td>
<td>0</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>24%</td>
<td>395160</td>
<td>0</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>45%</td>
<td>722909</td>
<td>722670</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>88%</td>
<td>2510661</td>
<td>2511070</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>90%</td>
<td>4371175</td>
<td>4376300</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>88%</td>
<td>861651</td>
<td>8613250</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>77%</td>
<td>16269359</td>
<td>16464000</td>
<td>16184</td>
<td>Warps</td>
</tr>
<tr>
<td>13</td>
<td>75%</td>
<td>29912865</td>
<td>30395780</td>
<td>16184</td>
<td>Warps</td>
</tr>
</tbody>
</table>
this point, the warp scheduler in the GPU runs the remaining warps (i.e. more than 160 warps) pre-emptively instead of concurrently. This phenomenon can be proved also checking the durations for the particular kernels. When the limit of 10240 threads (or 160) warps are reached the kernel run times almost doubles in time. Therefore, over 160 warps, the remaining warps are scheduled to run later, when the GPU has enough resources.

In this section, we saw the veritable reason affecting the execution times of the kernels, which is warp limit. For example, the total execution time for the whole program is the summation of the durations of each of these kernels. If one considers the same kernel requires to be launched with thousands of large grids, such as our example in Figures 35 and 36. Then it is clear that even a few micro-second will become a couple of seconds of latency in the end, similar to the rolling snowball effect.

7.7. Answers to the Research Questions for this thesis

In Chapter 2, Section 2.2.3, we found that some important aspects were simply missing from the literature. They were initialization of the test data, new CUDA memory model, scalability issues in CUDA and OpenMP threads, and test cases for sorting algorithms. Therefore, it will be wiser to assess these subjects using the codes in this thesis, or create some code for this section, such as the initialization of data part.

This work is intended to improve the quality of this thesis, by using the batch information according to the field research papers.

7.7.1. Data initialization

This was the most disregarded subject, possibly because simplicity of mentioning, in the literature papers. In most of the papers, the focus was on the sorting algorithms and their performance in terms of time. Although, in paper [60], it is stated that the Python library *numpy* is used to create random 32-bit floating point number set of data between 1024 elements and 32.768 elements. However, the same paper uses a thin wrapper for CUDA and Python integration, and in this thesis, Python integration will not be used. Therefore, this Section is devoted for code initialization.
As we have already discussed in Section 4.2.7, new CUDA dynamic parallelism allows the library function calls from within the kernel functions. Since, there is a built-in library for CUDA that is called cuRAND library; it is easier to achieve an initialized array in CUDA. The following pseudo code (i.e. Algorithm 10) shows generating a random number array with cuRAND.

**Algorithm 10** cuRAND random number generator for CUDA

1: curandCreateGenerator()
2: curandGenerateUniform(deviceArray, arraySize)

However, the same function cannot be used for OpenMP, if the CUDA “llvm” compiler is not chosen for compiling OpenMP. In this thesis, OpenMP is always compiled with “llvm” but if one assumes this was not the case, and then a pseudocode for the above operation would look like the Algorithm 11.

**Algorithm 11** random number generation for OpenMP

1: pragma omp parallel
2: srand()
3: pragma omp for
4: for(i → 0 to i → arraySize)
5: dataArray ← rand() % arraySize

### 7.7.2. New memory management model in CUDA

Another, research question was the usage of new memory model. Although most of the papers in our pool are belonged to post 2010 era, the memory model in CUDA has been just changed, in 2015, with the introduction of CUDA dynamic parallelism. Not surprisingly, the new model was not present in any of the papers. Therefore, explaining the difference between the new memory model and the old memory techniques in CUDA decided to be part of this thesis.

The old memory model is involved the explicit memory copying between the global and device memory, it has already been discussed in the previous chapters. However, for a sorting algorithm, the actual speed-up was being occurred where the local memory is used in CUDA.
Accordingly, in old memory model it was the case of making as much as operations before deferring the control back to the CPU. In addition, the local memory (i.e. registers, and not card memory) have been being used for making small but faster kernel calculations. However, the new dynamic parallelism, automatically distributes the available registers among each thread. Which also means, a kernel code that successfully runs on a new GPU, with using local memory, will not work on an older GPU.

An example for the above paragraph is present in the previous section, Section 7.5, where the Nsight profiler shows the register usage, and automatic memory management for the HBquick sort program, which was introduced in the Chapter 6.

Briefly, the new memory model makes coding in CUDA easier. For the reader’s attention the properties of GPU, which is used in this thesis, is present in Appendix A.1.

### 7.7.3. Scalability issues in shared memory languages

This is actually a very general subject to all shared memory systems, and might be the one of the reasons why there is increasing demand for data-level parallelism instead of shared memory systems. That is why; finding a direct answer for this question seems a little out of the scope of this thesis.

The so-called scalability issue is only present using shared memory systems. In CUDA, kernel code has to be written in such way that every thread executes the kernel code at least once. Otherwise, it is called branch divergence, this is both mentioned in Section 3.7 and Section 7.4.

For OpenMP based sorting algorithms, which are used in this thesis, the appropriate information obtained from measuring wall-clock execution times of these algorithms is present in Section 7.3. The test results show a speed-up proportional to the thread count for our algorithms, however, the CPU used for the tests has only 4 actual threads. This means, a system with more actual threads may reveal opposite results, such as slow-down instead of speed-up when the algorithms are run with more than 4 threads.
7.7.4. Testing the outputs

In our pool of papers, another issue was the absence of the mentioning of the word test in its actual meaning. Thus far, nearly all of the papers mentioned to test as, “testing” the program for desired execution time. For this reason, the actual meaning of test (i.e. testing the output validity) for a sorting algorithm will be examined, in order to figure out the reasons for the absence of the testing phrase from papers.

Actually, to test a parallel function could be quite challenging. The reasons for that involves but not limited to the following:

1. Test Array size in device or system memory.
2. Time needed to test the function.
3. Accurately understand the error codes returned by GPU in runtime.

In most of the literature papers, it is mentioned that a sorting algorithm coded with a parallel language is usually superior to a sequential (i.e. single thread) code in terms of time and memory space [40]. Therefore, it will be very problematic to use the traditional code for testing purposes of the parallel code. Of course, a simple algorithm like the Algorithm 12 is easy to code but it has flaws in itself. These flaws are incompatibility of some C language statements in both CUDA and OpenMP. For example, break clause cannot be used in both parallel languages. On Windows, the code line is redundant in runtime, and on Linux, the code does not compile! Therefore, if the array size was too long, and if the error occurred in just the beginning, the test function does not terminate at the occurrence of the first error, which obviously means unnecessary computations are made.

---

**Algorithm 12** CUDA or OpenMP test case with intentional redundant code

1: for(i → 0 to i → arraySize)
2:    if ( array[i] > array [ i+1] )
3:        print “Array not sorted on”, i
3:        break
4:    endif
5: endfor
The answer for this question is simple, although only using dynamic parallelism. The Algorithm 13 shows the test case only for the CUDA based sorting algorithms. Again break clause cannot be used in CUDA language. Therefore, a simple loop control is constructed using a Boolean variable that is isSorted, and a recursive function testFunc. The only consideration here is the stack size, but the stack size can grow up to device memory size. Moreover, it seems, an industrial sized application is needed to overwhelm it.

**Algorithm 13** CUDA kernel code for testing sorted array

1: testFunc (array, isSorted)
2: k ← unique thread number
3: if (array[i] > array [i+1])
4:    isSorted = 0
5: print “Array not sorted on”, k
6: testFunc<<<1,1>>>(array, isSorted)

In addition to anything above, in CUDA language it is the programmers’ responsibility to write a function for printing the error codes returned by GPU in the runtime. Otherwise, the program just terminates normally without giving a clue. In other words, in CUDA the error handling mechanism is not automatic. Algorithm 14 below show the appropriate code to print run time occurred error codes on a console window, and it should be written somewhere in the code to compile. This simple function is very useful where CUDA is compiled normally but execution is halted because of a runtime error. For example, an untested range of input can cause the program to give a runtime error.

**Algorithm 14** A function for runtime error checking in CUDA

1: #define cucheck(call)
2:    cudaError_t cucheck_err = (call)
3: if (cucheck_err != cudaSuccess)
4:    print FILE, LINE, error_string
7.7.5. **Test cases for the quicksort and merge sort**

The proposed algorithms in this thesis use a combination of merge sort with selection sort and quicksort with selection sort. While the selection sort and merge sort are only effected in execution times when using different ranges of inputs. The quicksort algorithm uses a partitioning logic, which might affect the program output when using different ranges and types of inputs. Because, the algorithm of the quicksort uses, the previous value of pivot element, to sort current partition, then the same pivot value is used as the next partitioning value for the algorithm. This is clear that if there are repeating elements in different locations of the input array, then the quicksort algorithm might fail.

In Section 7.1, the algorithms for creating different types of inputs are given. In this section, some test cases will be created using these techniques.

**7.7.5.1. Test cases for merge sort**

The merge sort algorithm makes use of partitioning the input array in to equal sizes of parts. In our proposed algorithm, the sub-arrays smaller than a threshold value is sorted with a different algorithm, although, this does not affect the partitioning logic. In other words, the merge sort algorithm’s behavior is not dependent on the input. Therefore, the test cases should involve different types of input (i.e. integer, floating-point numbers), and different ranges of input.

Test case 1 (TC1): The input set is uniformly distributed integer numbers, which are already sorted in the opposite direction of the expected sorting direction (i.e. decreasing numbers, from the array size to zero, with no repeated numbers). In addition, this case is considered as a worst case for computational complexity for a single-threaded (i.e. sequential) computation. For the reason that, when the input array is already sorted in opposite direction, there will be n occasions of compare and swap operations where n is the size of the input array. However, in CUDA data-parallelism, the kernels should always be launched with same number of threads; therefore, this necessity eliminates the time difference that might occur due to the increased computational complexity. Nevertheless, the OpenMP version will be affected from increased compare-swap operations in terms of time, but it is more of a regression testing (i.e. non-functional testing, e.g. performance test) consideration.
For example, consider below, the 6 numbers to sort, for simplicity, the array is sorted in descending order for creating a *worst case* scenario for sorting.

![Diagram of sorting process]

**Figure 37 Test case 1 for hybrid merge sort**

If one compares Figure 37, above, to Figure 27 in Chapter 6.1, then it is clear that the program behavior does not change according to the order of the input elements. However, in this algorithm the same behavior causes the increased number of swap operations due to elements being already sorted in the opposite direction compared to desired sorting direction.

Therefore, TC1 is a performance test, and not a robustness test for the merge sort algorithm.

Test case 2 (TC2): The input array consists of integer numbers, which are not uniformly distributed, and there is unknown quantity of repeating numbers. (i.e. the array obtained with, C language built-in rand() function, seeded with the maximum
integer number the system used is capable). This is considered as the average case in computational complexity for all of our sort functions. In both CUDA and OpenMP versions of the merge function, double pointers (i.e. opposite of the in-place sort) are used for merging parts, so output is not affected by the repeating input.

Therefore, TC2 tests performance because the same reason TC1 is so.

Test case 3 (TC3): The input array consists of floating point numbers, which are not uniformly distributed, and there is unknown quantity of repeating numbers (i.e. obtained with the cuRAND library’s default random generated floating-point numbers function, Algorithm 10). This case needs change in the code, such that the functions are modified for accepting the different types of inputs (i.e. integer, floating-point, etc.). This is almost the same test case with TC2, however the input is generated by a parallel library function. This means, the merge sort algorithm can be tested for robustness testing (i.e. testing output correctness).

The TC3 is both a performance test and a robustness test. This means, being capable of executing more input types makes a program less prone to errors, thus more robust. In addition, TC3 tests merge function for correctness of the significance of the decimal digits of the output elements. That is testing the algorithm if an unwanted truncating happens, due to bad design, caused by the assign operations to mismatched variable types.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Expected output</th>
<th>Achieved output in CUDA</th>
<th>Achieved output in OpenMP</th>
</tr>
</thead>
</table>
| TC1       | • Array is sorted.  
           | • No repeating numbers introduced, other than one’s already present in the input.  
           | • Spacing between input elements is not altered. | Pass            | Pass         |
| TC2       | • The rules above, plus the significance of the decimal part is not lost (i.e. possible truncate errors). | Pass            | Pass         |
| TC3       | • The rules above, plus the significance of the decimal part is not lost (i.e. possible truncate errors). | Pass            | Pass         |
7.7.5.2. Test cases for quicksort

The quicksort algorithm is designed for sequential computing in mind. For that reason most of the community accepted (i.e. commonly known versions) quicksort algorithms use one or more decision points for sorting the current partition, and then using that same decision point (also called pivot) for partitioning point of the next iteration. This behavior, causes an unstable algorithm (for some cases), where the output is effected directly by the input order of the elements. Therefore, it is clear that testing this function for robustness is more important than doing a regression test.

Test case 1 (TC1): The Figure 38 below, shows the hybrid quicksort behavior for a small set of numbers, but some elements are repeated and the whole array is in unsorted order.

Figure 38 Hybrid quicksort with 8 numbers

In Figure 38, the quicksort algorithm is called with 8 numbers, for simplicity, in the first recursion the sub-array’s left is equal to 0 and right is equal to 7. Then the
**partitioning function** is called (instead of Selection Sort, because DT is 2), for simplicity number 4 is at the very back of the array because **partition function** always selects the last value as pivot. The figure clearly shows that if the last value were a zero instead of four, the function would need another recursion for the sort operations to begin. This does not result in a failure because the last element is always swapped with the first iterator. The OpenMP version is also affected by the varying order of input elements and will be tested by the same inputs.

Therefore, TC1 should consist of same sized arrays that are populated with random numbers. In addition, there should be repeating numbers to test the algorithm for robustness.

Test case 2 (TC2): As it is stated in testing merge sort Section, in CUDA the block sizes in the same grid should be same, and not changing (i.e. executing same kernel). However, in this Section hybrid quicksort uses CUDA *dynamic parallelism* which means the block size is determined by the special hardware in the GPU (using SM 5.0 and above GPUs) for every recursion of the GPU function (or kernel). For this reason, the changing sub-array sizes do not affect the program stability.

Therefore, a better case would be changing the defer threshold number and look for the errors. Since they might occur in the intersection points with **partition function** where the sorting is altered to selection point.

Test case 3(TC3): This is the same test as given in the previous section, testing merge sort. Since, it is enough for TC1 and TC2 to cover completely the input range where the quicksort might generate an erroneous output. TC3 should look for a robustness test with different types of input. First two test cases were in integer format.

Therefore, TC3 uses floating-point numbers, generated with cuRAND library, for robustness testing of the hybrid quicksort. A reminder for reader, the OpenMP can be compiled using CUDA’ *nvmm* compiler, so the cuRAND generated array is also available to OpenMP functions.
<table>
<thead>
<tr>
<th>Test case</th>
<th>Expected output</th>
<th>Achieved output in CUDA</th>
<th>Achieved output in OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>• Array is sorted.</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Varying input does not change the program output.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Changing defer threshold does not affect output.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC2</td>
<td>• The rules above, plus the significance of the decimal part is not lost (i.e. possible truncate errors).</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The tests for hybrid merge sort reveals; the *pivot point*, selection method in the *partition* function could affect the timing results. There is nothing wrong for choosing the last array element as the pivot, however if that element is bigger than all the other elements in that sub-array, that pivot element is replaced with the element the first iterator pointing. Then, the pivot function is called with same left and right value again.

Therefore, a parallel reduction sum (Algorithm is given in Appendix C) could be used to find the sum of elements in the particular sub-array. Then, the summation found can be divided with the element count in the sub-array revealing a median value. However, that median value might not be present in the array at all. For example, consider the subarray, bellow; the median is 45 (truncated). Another fact about the *reduction sum* is, it can only be computed with a single block, then the median value can be calculated only when the launched block returns control to launcher (possibly parent kernel (quicksort) in our case). Therefore, many extra computations needed for finding the median, almost eliminates the benefits gained from using it, against waiting the partition function to swap the pivot values one by one (causing extra launches with same left and right values).

<table>
<thead>
<tr>
<th>0</th>
<th>20</th>
<th>1</th>
<th>5</th>
<th>60</th>
<th>80</th>
<th>99</th>
<th>100</th>
</tr>
</thead>
</table>

So another algorithm, sum scan can be used for finding the maxima and minima in the array. However, swapping in the above array the minima 0 with maxima 100 does not change behavior. A remainder for the reader, the current input needs 4 calls to *partition* function instead of 2 (the example in Figure 38) calls on average.
However, even if it was $\theta$, the function is still needed that 2 extra calls to partition function (with same left and right values).

To conclude, an external kernel launch for neither reduction sum nor sum scan is not feasible, if we consider both algorithms to find the perfect pivot for the \textit{partition} function.
CHAPTER 8

CONCLUSIONS AND FUTURE WORKS

In this thesis, the parallel sorting subject is covered using two common parallel languages in the literature, which were CUDA and OpenMP. While the preference of examining these languages were not unintentional, the information obtained from researching field related papers is showed that most of the effort making those papers are spent to comparing the parallel versions of the algorithms to their single-threaded counterparts. However, there should be a difference between speed-up and code writing effort is considered not just with the sequential languages but also with the parallel languages themselves. Therefore, in this thesis it is deemed suitable to compare parallel languages against each other and against to their sequential versions. In this thesis, first five chapters give background information about the parallel languages. In Chapter 2, a small systematic literature review is made with a pool of around seventy papers. Then the results from the SLR study showed that the information about testing the programs or information about metrics other than speed-up is being missed completely. Therefore, in Chapter 7, this missing information is conveyed using the data available from this thesis. Then, it is found out that providing these data was very easy. Thus, it has been concluded that the information about the codes for initialization or testing of the parallel algorithms missed simply because of the choice of the authors. In Chapter 6, the algorithms that are suitable for comparison are introduced and in the next chapter the timing results are obtained by comparing the algorithms. Then the algorithms are examined for the reasons that might cause the slowdown to happen. It was found that in CUDA language there is a hard limit of concurrently executing threads, even when these threads were grouped in warps, for a GPU. Therefore, a future study can re-evaluate the timing results found in this thesis when a more advanced hardware exists in the
The time comparisons also show that there is a huge difference between the CUDA and OpenMP results, with the same algorithms written in OpenMP being faster. This result can be explained with the higher clock frequencies of the CPU compared to GPU, because CPUs are designed for hiding latency. However, this is where CUDA statement holds true, because developers of CUDA language never claims that CUDA language can beat a CPU when comparison is made with latency. Moreover, what is claimed by CUDA language is providing much higher throughput then a CPU and providing it for a very long time. In addition, if one considers that a GPU has much lower power consumption, it should be more favorable to build a cluster of GPUs instead of a cluster of CPUs. Then, those two entities can be fairly compared. In addition, the results obtained from this thesis show that OpenMP based algorithm works as intended because their close relativeness to the C language. However, CUDA is much harder to both code and debug, simply because being introduced recently.

This study revealed the data-level parallelism has a promising future for even using the algorithms, which arise from many data dependent memory I/O operations. Although, the timing results in Chapter 6 reveals the OpenMP based algorithms have significant performance efficiency, that assumption only holds true if the comparisons are made using only the speed-up in wall-clock time, in mind. However, in this thesis, information about other metrics for assessing parallel languages against each other is given; these were memory efficiency, throughput and computations per watt efficiency. Then, it is clear that the data-level parallelism has significant benefit when compared to the task-level parallelism. In addition, current CPUs has multiple identical cores on the same chip, making them the head starters when the computation involves many operations, where hiding the latency almost impossible (e.g. a sorting algorithm where a computation uses the output of the previous computation). Therefore, future improvements to the GPU hardware can follow the same approach today, of making less cores (or SMs in GPU) but making them more heavy weight in terms of computational capabilities. In addition, in new architectures of Nvidia devices, SM count decreases but the number of SP (streaming processors) in the SMs increases. That means, in the future the data-level parallelism will have much better results, even with the data dependent computations, when compared to task-level parallelism. In addition, in Section 7.7.4.1 it is shown that different test
inputs could change the output of the same function significantly, most of them erroneous. Therefore, a thin wrapper for unit testing the CUDA code is a necessity. A future work to make this happen could positively affect coding in CUDA.
REFERENCES


https://docs.google.com/spreadsheets/d/1-N6SrSuCLSOMtYDccE0GveY0RGHw_eFwiNyrtxtBpn4/edit?usp=sharing


Moore, Nicholas. "Kernel specialization for improved adaptability and performance on graphics processing units (GPUs)." (2012).


D. S. Hirschberg, Communications of ACM, 21(8), 1978


APPENDIX A

GENERAL INFORMATION ABOUT TEST ENVIRONMENT

A.1 The general properties for the GPU used

<table>
<thead>
<tr>
<th>Device 0:</th>
<th>&quot;GeForce GTX 850M&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Driver Version / Runtime Version</td>
<td>7.0 / 7.0</td>
</tr>
<tr>
<td>CUDA Capability Major/Minor version number:</td>
<td>5</td>
</tr>
<tr>
<td>Total amount of global memory:</td>
<td>4096 MBytes (4294967296 bytes)</td>
</tr>
<tr>
<td>(5) Multiprocessors, (128) CUDA Cores/MP:</td>
<td>640 CUDA Cores</td>
</tr>
<tr>
<td>GPU Max Clock rate:</td>
<td>902 MHz (0.90 GHz)</td>
</tr>
<tr>
<td>Memory Clock rate:</td>
<td>900 Mhz</td>
</tr>
<tr>
<td>Memory Bus Width:</td>
<td>128-bit</td>
</tr>
<tr>
<td>L2 Cache Size:</td>
<td>2097152 bytes</td>
</tr>
<tr>
<td>Maximum Texture Dimension Size (x,y,z):</td>
<td>1D=(65536), 2D=(65536, 65536)</td>
</tr>
<tr>
<td>Maximum Layered 1D Texture Size, (num) layers</td>
<td>1D=(16384), 2048 layers</td>
</tr>
<tr>
<td>Maximum Layered 2D Texture Size, (num) layers</td>
<td>2D=(16384, 16384), 2048 layers</td>
</tr>
<tr>
<td>Total amount of constant memory:</td>
<td>65536 bytes</td>
</tr>
<tr>
<td>Total amount of shared memory per block:</td>
<td>49152 bytes</td>
</tr>
<tr>
<td>Total number of registers available per block:</td>
<td>65536</td>
</tr>
<tr>
<td>Warp size:</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of threads per multiprocessor:</td>
<td>2048</td>
</tr>
<tr>
<td>SM count:</td>
<td>5</td>
</tr>
<tr>
<td>Number of concurrently active threads:</td>
<td>10240</td>
</tr>
<tr>
<td>Maximum number of threads per block:</td>
<td>1024</td>
</tr>
<tr>
<td>Max dimension size of a thread block (x,y,z):</td>
<td>(1024, 1024, 64)</td>
</tr>
<tr>
<td>Max dimension size of a grid size (x,y,z):</td>
<td>(2147483647, 65535, 65535)</td>
</tr>
<tr>
<td>Concurrent copy and kernel execution:</td>
<td>Yes with 1 copy engine(s)</td>
</tr>
<tr>
<td>Run time limit on kernels:</td>
<td>Yes</td>
</tr>
<tr>
<td>Integrated GPU sharing Host Memory:</td>
<td>No</td>
</tr>
</tbody>
</table>

A.2 General system info

<table>
<thead>
<tr>
<th>System info</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i7-5500</td>
</tr>
<tr>
<td>GPU</td>
<td>Gtx 850M</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 10</td>
</tr>
<tr>
<td>Cuda version</td>
<td>CUDA 7.0</td>
</tr>
<tr>
<td>OpenMP version</td>
<td>OpenMP 3.0</td>
</tr>
<tr>
<td>Development IDE</td>
<td>Nsight v4</td>
</tr>
</tbody>
</table>

APPENDIX B

EXTRA PROFILER RESULTS

B.1 Nsight profiler view to show UMA operations

Figure 39 UMA automatic memory management
B.2 Large versions of the images from Section 7.4-7.5

Appendix B 1 The larger image for Figure 32

Appendix B 2 The larger image for Figure 33
<table>
<thead>
<tr>
<th>Device Name</th>
<th>Function Name</th>
<th>Threads per Block</th>
<th>Threads</th>
<th>Duration (us)</th>
<th>Occupancy</th>
<th>Achieved Occupancy</th>
<th>Achieved Occupancy</th>
<th>Performance Counters [1]: Sm Active Warps</th>
<th>Performance Counters [1]: Active Cycles</th>
<th>Performance Counters [1]: Warps Launched</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>2</td>
<td>1024</td>
<td>2048</td>
<td>47.987</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.20</td>
<td>0</td>
<td>39274</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>3</td>
<td>1024</td>
<td>2072</td>
<td>47.615</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.20</td>
<td>0</td>
<td>48316</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>5</td>
<td>1024</td>
<td>5120</td>
<td>47.649</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.50</td>
<td>2783500</td>
<td>88499</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>10</td>
<td>1024</td>
<td>10240</td>
<td>64.512</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.99</td>
<td>10066320</td>
<td>157295</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>20</td>
<td>1024</td>
<td>20480</td>
<td>186.973</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.84</td>
<td>19177335</td>
<td>3116660</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>40</td>
<td>1024</td>
<td>40960</td>
<td>357.497</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.66</td>
<td>40114653</td>
<td>4999409</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>80</td>
<td>1024</td>
<td>81920</td>
<td>659.640</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.58</td>
<td>76438530</td>
<td>2406969</td>
</tr>
<tr>
<td>GeForce GTX 850M</td>
<td>TopDownMerge</td>
<td>160</td>
<td>1024</td>
<td>163840</td>
<td>1276.472</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0.97</td>
<td>134590320</td>
<td>2502761</td>
</tr>
</tbody>
</table>

**Appendix B.4 The larger image for Figure 35**

**Appendix B.5 The larger image for Figure 36**
APPENDIX C
SOURCE CODES

REDUCTION SUM ALGORITHM
The reduction sum is the fastest possible way of adding the values of elements of an array together. However, it is still not fast enough as our quick sort kernel, uses dynamic parallelism, because it is designed for old architectures and does not have a newer version yet. This function is an official example that comes with the CUDA development package examples.

```c
__device__ __managed__ int sdata[65536];

template<unsigned int blockSize>
__global__ void reduce(int *in, int *out, unsigned int n){
    // __shared__ int sdata[];
    unsigned int tid=threadIdx.x;
    unsigned int i=blockIdx.x*(blockSize*2)+tid;
    unsigned int gridSize=blockSize*2*gridDim.x;
    sdata[tid]=0;
    while (i<n) {
        sdata[tid]+=in[i]+in[i+blockSize];i+=gridSize;
        __syncthreads();
        if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
        if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
        if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }
    }
    if (tid < 32){
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
        if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
        if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
        if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
    }
}
```

************Start of the file for hybrid merge sort function (in CUDA)
**************************

HYBRID MERGE SORT FUNCTION (IN CUDA)
/*Hybrid merge sort in CUDA, a program that sorts either integers or single precision floating point numbers in ascending order. Author: Hakan GOKAHMETOGLU, written for the thesis document, for Msc. Degree in Software Eng. In Atilim University */
#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <time.h>
#include <omp.h>
#include <cuda.h>
#include <curand.h>

#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { \
    printf("Error at %s:%d\n", __FILE__, __LINE__);\n    return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { \

printf("Error at %s:%d\n", __FILE__, __LINE__); return EXIT_FAILURE;}

//Forward function declarations
template <typename T>
void TopDownSplitMergeFunc(T *A, int iBegin, int iEnd, T *B);
template <typename T>
__global__ void TopDownMerge(T *A, const int iBegin, const int iEnd, const int iMiddle, const int iEnd, T *B);
template <typename T>
__global__ void CopyArray(T *B, const int iBegin, const int iEnd, T *A);
template <typename T>
__global__ void test1(T *A, int n);
template <typename T>
void ss(T *a, const unsigned n);
float tm();

int dt, mb, ib;

//device variables to hold threshold and block size values
__device__ unsigned int th, bl;
unsigned int thrd, blck;

//swaps the values of two pointers of any type
template <typename T>
inline void swapss(volatile T *a, volatile T *b)
{
    T temp; temp = *a; *a = *b; *b = temp;
}

//selection sort in OpenMP
template <typename T>
void ss(T *a, const unsigned n)
{
    T temp; //local variable, holds value to swap
    unsigned int p = 0; // local variable, index variable
    unsigned int idx = 0; // local variable, index variable

    // the index variables are declared outside the parallel region
    // according to the OpenMP syntax
    // only Linux systems give warning to declare like this!
    #pragma omp parallel shared(a, n, temp, p) private(idx)
    {
        idx = 0;
        #pragma omp parallel for
        {
            for (idx = 0; idx < n; idx++)
            {
                temp = a[idx];
                p = idx;
                while (p > 0 & & a[p-1] > temp)
                {
                    swapss(&a[p-1], &a[p]);
                    p--;
                }
                a[p] = temp;
            }
        }
    }

    // (top down) merge sort function's split function
    // refactored to defer the sorting for array sizes to sort below a threshold.
    // for more details see: en.wikipedia.org/wiki/Merge_sort
    // it was possible to implement this function with dynamic parallelism (i.e. a recursive function)
    // although, the selection sort (ss) function is an external function and cannot be launched from a global function. Then, the reasons for designing this function as an external
    // function is explained in the thesis document, Chapter 6.
    template <typename T>
    void TopDownSplitMergeFunc(T *A, int iBegin, int iEnd, T *B)
    {
        // decision path for deferring the sorting to selection sort or
// keep partitioning the array, until the dt (defer threshold) is reached.
if((iEnd-iBegin) < dt) {
    ss(A+iBegin,(iEnd-iBegin));  //OpenMP selection sort
}  else {
    //the middle point of array to merge
    int iMiddle = (iEnd + iBegin) / 2;
    //split making a partition where middle is the end point.
    TopDownSplitMergeFunc(A, iBegin, iMiddle, B);
    //split making a partition where middle is the start point.
    TopDownSplitMergeFunc(A, iMiddle, iEnd, B);
    //calculate the grid and block sizes for kernels below
    thrd = MAX(mb, 64);  blck = MAX(((iEnd-iBegin)+(thrd-1))/thrd,1);
    // merge the two half runs
    TopDownMerge<<<blck,thrd>>>(A+iBegin, iBegin, iMiddle, iEnd, B+iBegin);
    cudaDeviceSynchronize();
    //copy the merged array back to A
    CopyArray<<<blck,thrd>>>(B+iBegin, iBegin, iEnd, A+iBegin);
    //blocks the first thread finishes executing the kernel from informing
cudaDeviceSynchronize();
    //device that its free to run other kernels!
}

__device__ int getId(void){return blockIdx.x*blockDim.x+threadIdx.x;}
__device__ int getStride(void){return blockDim.x*gridDim.x;}

//Kernel for merging the sub-arrays beginning from
template <typename T>
__global__ void TopDownMerge(T *A, const int iBegin, const int iMiddle, const int iEnd, T *B)
{
    unsigned int i=0;
    //index pointing to the first index of the arrays to merge
    unsigned int j=(iEnd-iBegin)/2;
    //index pointing to the mid-point
    unsigned int n = (iEnd-iBegin);
    //variable n is the array length to merge
    unsigned int *a = (unsigned int*)A;
    //pointer for array to sort
    unsigned int *bk = (unsigned int*)B;
    //pointer for temporary array
    //each thread executes the loop once
    for (unsigned int k = getId();k < n; k++)
    {
        if(j == n) {'bk+(k) = *(a+(i++));'}
        else if(i == n/2) {'bk+(k) = *(a+(j++));'}
        else if(*(a+j) <= *(a+i)) {'bk+(k) = *(a+(j++));'}
        else {'bk+(k) = *(a+(i++));'}
    }
    __syncthreads();
    //synchronize threads before next warp
}

//copies elements back from temp array B to array to sort.
//uses warps (the difference is explained in the thesis in Chapter 4)
template <typename T>
__global__ void CopyArray(T *B, const int iBegin, const int iEnd, T *A)
{
    unsigned int *b = (unsigned int*)B;
    //pointer for temp array
    unsigned int *a = (unsigned int*)A;
    //pointer for actual array
    int n = (iEnd-iBegin);
    //array length to copy
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;  //thread index number
    unsigned int stride = blockDim.x * gridDim.x;
    //index stride for warped operation

}
```c
while (i < n) {
    *(a+i) = *(b+i);  // copy from temp to actual
    i += stride;      // add stride for warped operation
}
```

```
// tests the output for increasing series of numbers,
// explained in Chapter 7, Algorithm 11
```

```
template <typename T>
__global__ void test1(T *A, int n)
{
    bool sorted = 1;
    int *ai = A + 1;
    int *aj = A + 0;

    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;
    if (i < n) {
        while (sorted) {
            if (*(ai+i) < *(aj+i)) {
                sorted = 0;
                printf("array not sorted at %d\n", i);
            }
            i += stride;
        }
    }
}
```

```
// initial values for validating OpenMP
int realCores = 1, nestDepth = 2, numThread = 1;
int main(int argc, char** argv) {
    int n = 1024, i;  // number of input array to sort
    int nHdeferTreshHold = 32;  // threads count for merge function
    int nHmergeBlockDim = 32;
    char sprs = 'x';  // show/hide the results
    char isRandom = 'b';  // initialize input array (random or backwards)

    printf("usage: Arraysize | Defer to selection sort threshold | Merge sort block dim | s for hiding output | b for backwards, any key for random input");
    if (argc > 1) n = atoi(argv[1]);
    if (argc > 2) nHdeferTreshHold = atoi(argv[2]);
    if (argc > 3) nHmergeBlockDim = atoi(argv[3]);
    if (argc > 4) sprs = *argv[4];
    if (argc > 5) isRandom = *argv[5];

    // control the OpenMP availability, and displays the total number of threads in the system, and the actual core number
    #pragma omp parallel
    {
        numThread = omp_get_max_threads();
        realCores = omp_get_num_threads();
        omp_set_nested(nestDepth);
        omp_set_num_threads(numThread);

        #pragma omp single
        printf("Calculating with %d number of threads with %d real CPU(s)\n", numThread, realCores);
    }

    // displays if device is busy, also the stack and heap size
    // for the program. If there was a job on the GPU, that is not terminated properly this will show it
    size_t pend, heap, sta;
    cudaDeviceGetLimit(&pend, cudaLimitDevRuntimePendingLaunchCount);
    cudaDeviceGetLimit(&heap, cudaLimitMallocHeapSize);
    cudaDeviceGetLimit(&sta, cudaLimitStackSize);
```
printf("pending jobs %d \n heap size = %d \n stack size = %d \n", pend, heap, sta);

//input array A and temporary array B
int *A_h, *B_h; float *farr;
//array sizes to allocate on memory
size_t s = sizeof(int) * n;

//new CUDA dynamic parallelism model, GPU manages the variables
CUDA_CALL( cudaMallocManaged((int**)&A_h, s) );
CUDA_CALL( cudaMallocManaged((int**)&B_h, s) );
CUDA_CALL( cudaMallocManaged((float**)&farr, sizeof(float) * n) );
curandGenerator_t gen;
float *devData, *devB;
/* Allocate n floats on device */
CUDA_CALL( cudaMemcpy(nullptr, devData, n*sizeof(float)));
CUDA_CALL( cudaMemcpy(nullptr, devB, sizeof(float) * n) );

if (isRandom=='b')
{
    for (i = 0; i < n; i++) A_h[i] = n - 1 - i;

    printf("\n starting parallel sort with %d elements and \n defer threshold = %d \n", n, hdeferTreshHold);
dt = hdeferTreshHold; mb = hMergeBlockDim;
tm(); //start-stop timer, starts here
TopDownSplitMergeFunc(A_h, 0, n, B_h); //call to split function
    //timer stops here
    printf("time for parallel execution tm = %f sec.\n", tm());
    //parallel test for sorted array
test1<<<n/64,64>>>(A_h,n-1);
    //print screen the input array
}
else{
    /* Create pseudo-random number generator */
    CURAND_CALL( curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT) );
    /* Set seed */
    CURAND_CALL( curandSetPseudoRandomGeneratorSeed(gen, 1234ULL) );
    /* Generate n floats on device */
    CURAND_CALL( curandGenerateUniform(gen, devData, n) );
    CUDA_CALL( cudaMemcpy(farr, devData, n * sizeof(float), cudaMemcpyDeviceToHost) );
    printf("\n starting parallel sort with %d elements and \n defer threshold = %d \n", n, hdeferTreshHold);
dt = hdeferTreshHold; mb = hMergeBlockDim;
tm(); //start-stop timer, starts here
TopDownSplitMergeFunc(farr, 0, n, devB); //call to split function
    //timer stops here
    printf("time for parallel execution tm = %f sec.\n", tm());
}

//free arrays from the memory
cudaFree(A_h);
cudaFree(B_h);
cudaFree(farr);
cudaFree(devB);
cudaDeviceReset(); //needed for profiling the app. with Nsight profiler
exit (0);

//start-stop timer uses the C time.h
float tStart=0.000000000f;
float tEnd=0.000000000f;
float tm() {
    tEnd = clock();
    float t = (tEnd - tStart) / 1000.00000000f;
    tStart = tEnd;
    return t;
}
HBQUICK SORT (IN CUDA)

//Hybrid quick sort in CUDA, a program that sorts either integers or single precision floating point numbers in ascending order. Author: Hakan GOKHAMETOGLU, written for the thesis document, for Msc. Degree in Software Eng. In Attilim Universiety */
#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <time.h>
#include <cuda.h>
#include <curand.h>

//template <typename T> __device__ void swap( T& a, T& b );
template <typename T> __device__ void ss( T* a, const unsigned int n);
template <typename T> __global__ void quicksort( T* a, const int l, const int r);  //template <typename T> __global__ void test1( T* A, const int n, bool sorted);
template <typename T> __device__ int partition( T* a, const int l, const int r);

float tm();
#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { printf("Error at %s:%d\n", __FILE__, __LINE__); return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { printf("Error at %s:%d\n", __FILE__, __LINE__); return EXIT_FAILURE;}} while(0)

//device managed variable for defer threshold.
__device__ __managed__ int dt;

//custom swap function. Global functions cannot call system functions (for ex. std::swap())
template <typename T> __device__ void swap( T& a, T& b ) {
    T t = a; a = b; b = t;
}
__device__ int getId(void){return blockIdx.x*blockDim.x+threadIdx.x;}
__device__ int getStride(void){return blockDim.x*gridDim.x;}

//selection sort algorithm, pseduo code is given in Chapter 6
//template <typename T> __device__ void ss( T* a, const unsigned int n) {
}
//partition function, separated from main function, quick
//sort, for reading simplicity

template<typename T>
__device__ int partition (T *a, const int l, const int r)
{
    T x = a[r];
    int i = (l - 1);

    for (unsigned int j = getId() + 1; j <= r - 1; j+=getStride())
    {
        if (a[j] <= x)
        {
            swap (a[++i], a[j]);
        }
        j++;
    }
    swap (a[i + 1], a[r]);
__syncthreads();
    return (i+1);
}

//Global function(kernel) for quick sort, uses CUDA dynamic parallelism
//and CUDA streams

template<typename T>
__global__ void quicksort(T *a, const int l, const int r)
{
    const int len = r - l; //the length of current partition
    if ( len <= dt ){ //decision path to defer or not
        ss(l+1,len+1); //to the device function selection
        cudaDeviceSynchronize(); //sort
    } else
    {
        int par = partition(a, l, r); //partition device function, returns the
        cudaDeviceSynchronize(); //appropriate point to start a new partition

        if (l < par) //start a new stream with current left and
            //current partition point as the right value
        {
            cudaStream_t s;
            cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
            quicksort<<< 1, 1, 0, s >>>(a, l, par-1); //starts a new CUDA stream
            cudaStreamDestroy(s);
        }

        if (par < r) //start a new stream with current right and
            //current partition point as the left value
        {
            cudaStream_t s1;
            cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
            quicksort<<< 1, 1, 0, s1 >>>(a, par+1, r);
            cudaStreamDestroy(s1);
        }
    }
}

//initial values for validating OpenMP
int realCores=1,nestDepth=2,numThread=1;
int main(int argc, char** argv){
    int n = 1024,1; //number of input array to sort
                    //the integer number, to determine when will the sorting
                    //is deferred to the selection sort
    hdeferTreshHold = 32; //dt=hdeferTreshHold;
    char isRandom = 'b'; //initialize input array (int or floating)
    printf("usage: Arraysize | Defer to selection sort treshHold | \
          Merge sort block dim | s for hiding output | b for backwards, any key for random input\n");
    if(argc>1)n = atof(argv[1]);
if(argc>2)hdeferTreshHold = atoi(argv[2]);
if(argc>3)isRandom=argv[3];

//input array A and temporary array B
int *A_h;float * farr;
//array sizes to allocate on memory
size_t s = sizeof(int) * n;
size_t ss = sizeof(float) * n;

//new CUDA dynamic parallelism model, GPU manages the variables
CUDA_CALL(cudaMallocManaged((void**)&A_h,s));
CUDA_CALL(cudaMallocManaged((void**)&farr,ss));
curandGenerator_t gen;
if (isRandom==’b’)
{
    srand(32768);
    for (i = 0; i < n; i++) A_h[i] = rand() % n;

    printf("\\n starting parallel sort with %d elements and \n defer threshold = %d \n",n,hdeferTreshHold);
    tm(); //start-stop timer, starts here
    quicksort<<<1,1>>>(A_h,0,n); //call to quicksort function
    CUDA_CALL(cudaDeviceSynchronize());
    //timer stops here
    printf("time for parallel execution tm = %f sec.\n",tm());
    //parallel test for sorted array
    test1<<<1,1>>>(A_h,n,true);
    CUDA_CALL(cudaDeviceSynchronize());
}
else
{
    /* Create pseudo-random number generator */
    CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT));
    /* Set seed */
    CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen, 1234ULL));
    /* Generate n floats on device */
    CURAND_CALL(curandGenerateUniform(gen, farr, n));
    CUDA_CALL(cudaDeviceSynchronize());

    printf("\\n starting parallel sort with %d elements and \n defer threshold = %d \n",n,hdeferTreshHold);
    tm(); //start-stop timer, starts here
    quicksort<<<1,1>>>(farr,0,n-1); //call to quicksort function
    CUDA_CALL(cudaDeviceSynchronize());
    //timer stops here
    printf("time for parallel execution tm = %f sec.\n",tm());
    //parallel test for sorted array
    test1<<<1,1>>>(farr,n,true);
    CUDA_CALL(cudaDeviceSynchronize());
}
//free arrays from the memory
CUDA_CALL(cudaFree(A_h));
CUDA_CALL(cudaFree(farr));
CUDA_CALL(cudaDeviceReset());//needed for profiling the app. with Nsight profiler
exit (0);

//start-stop timer uses the C time.h
float tStart=0.000000000f;
float tEnd=0.000000000f;
float tm() {
    tEnd = clock();
    float t = (tEnd - tStart) / 1000.00000000f;
    tStart = tEnd;
    return t;
}

//tests the output for increasing series of numbers,
//explained in the Chapter 7, Algorithm 11
```c
__device__ int f=0;

template <typename T>
__global__ void test1(T *A, const int n, bool sorted)
{
    if (sorted) {
        T *ai = A + 1;
        T *aj = A + 0;

        while (sorted && f<n-1){
            if(*ai++ < *(aj++)){
                sorted=0;
                printf("array not sorted at %d\n",f);
            }
            f++;
        }
    }
}

HYBRID MERGE SORT (IN OPENMP)

/*Hybrid merge sort in OpenMP, a program that sorts either integers or single precision floating point numbers in ascending order. Author: Hakan GOKHAMETOGLU, written for the thesis document, for Msc. Degree in Software Eng. In Attilim University */

#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <omp.h>
#include <cuda.h>
#include <curand.h>
#define CURAND_CALL(x) do{
    if((x)!=CURAND_STATUS_SUCCESS) {
        printf("Error at %s:%d\n", __FILE__, __LINE__);
        return EXIT_FAILURE;}} while(0)
#define N 67188864
#define ch 1024
int i, numThread=4, realCores=2;
int A[N];
int B[N];
template <typename T> void TopDownMergeSort(int A[], int B[], int n);
template <typename T> void TopDownSplitMerge(int A[], int iBegin, int iEnd, int B[]);
template <typename T> void TopDownMerge(int A[], int iBegin, int iMiddle, int iEnd, int B[]);
template <typename T> void CopyArray(int B[], int iBegin, int iEnd, int A[]);
float tm();
template <typename T> void test(int n);
inline void swapss(volatile T *a, volatile T *b )
{
    T temp;temp = *a; *a=*b; *b=temp;
}
template <typename T>
void TopDownMergeSort(T A[], T B[], int n)
{
    TopDownSplitMerge(A, 0, n, B);
}
template <typename T>
void ss(T *A, const unsigned n)
T temp;  //local variable,holds value to swap
unsigned int p=0;    //local variable,index variable
unsigned int idx=0;    //local variable,index variable

//the index variables are declared outside the parallel region
//according to the OpenMP syntax
//only Linux systems give warning to declare like this!
#pragma omp parallel shared(a,n,temp,p) private(idx)
{
  idx = 0;
  #pragma omp parallel for
  {
    for(idx=0; idx < n; idx++)
    {
      temp = a[idx];
      p=idx;
      while (p > 0 && a[p-1] > temp)
      {
        swapss(&a[p-1], &a[p]);
        p--;
      }
    } // end of for loop
    a[p] = temp;
  } // end of parallel for
}

// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set)
template<typename T>
void TopDownSplitMerge(T A[], int iBegin, int iEnd, T B[])
{
  if((iEnd - iBegin) < 1024) // if run size == 1
    ss(A+iBegin, (iEnd-iBegin)); //return; consider it sorted
  else
  {
    int iMiddle = (iEnd + iBegin) / 2;  // iMiddle = mid point
    #pragma omp parallel sections
    {
      #pragma omp section
      TopDownSplitMerge(A, iBegin, iMiddle, B); // split / merge left half
      #pragma omp section
      TopDownSplitMerge(A, iMiddle, iEnd, B); // split / merge right half
    } // end of parallel sections
    TopDownMerge(A, iBegin, iMiddle, iEnd, B); // merge the two half runs
    CopyArray(B, iBegin, iEnd, A); // copy the merged runs back to A
  } // end of if
}

// left half is A[iBegin :iMiddle-1]
// right half is A[iMiddle:iEnd-1]
template<typename T>
void TopDownMerge(T A[], int iBegin, int iMiddle, int iEnd, T B[])
{
  int j=0;
  int i0 , i1;
  // While there are elements in the left or right runs
  #pragma omp parallel shared(A,B,iBegin,iEnd, iMiddle, i0, i1) private(j)
  {
    i0 = iBegin; i1 = iMiddle;
    #pragma omp parallel for
    for ( j = iBegin; j < iEnd; j++)
    {
      // If left run head exists and is <= existing right run head.
      #pragma omp parallel sections
      {
        #pragma omp section
        if(i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1])){
          B[j] = A[i0];
          #pragma omp atomic
          i0 = i0 + 1;
        } // end of if
        #pragma omp section
        else{
          B[j] = A[i1];
          #pragma omp atomic
        } // end of else
      } // end of parallel sections
    } // end of parallel for
  } // end of parallel region
} // end of TopDownMerge

110
i1 = i1 + 1; }
}

```cpp
template<typename T>
void CopyArray(T B[], int iBegin, int iEnd, T A[])
{
    int k = 0;
    #pragma omp parallel shared(A,B,iBegin,iEnd) private(k)
    {
        #pragma omp parallel for
        for( k = iBegin; k < iEnd; k++)
            #pragma omp critical
            A[k] = B[k];
    }
    int main(int argc, char **argv){
        curandGenerator_t gen;
        int n = 4096;
        int nestDepth = 2;
        char bck = 'b';
        if (argc > 1) n = atoi(argv[1]);
        if (argc > 2) bck=argv[2];
        #pragma omp parallel
        {
            numThread = omp_get_max_threads();
            realCores = omp_get_num_threads();
            omp_set_nested(nestDepth);
            omp_set_num_threads(numThread);
            #pragma omp single
            printf("Calculating with %d number of threads with %d real CPU(s)\n", numThread, realCores);
        }
        if (bck=='b')
        {
            tm();
            #pragma omp parallel shared(A) private(i)
            {
                #pragma omp parallel for
                for (i = 0; i < n; i++)
                {
                    A[i] = n - 1 - i;
                }
            }
            printf("time for parallel initialization is %f\n",tm());
            tm();
            TopDownMergeSort(A, B, n);
            printf("time for parallel execution is %f\n",tm());
            tm();
            test(n);
            printf("time for parallel test is %f\n",tm());
        }
        else
        {
            float *devData;
            float *farr = (float*)malloc(n*sizeof(float));
            float *fbrr = (float*)malloc(n*sizeof(float));
            cudaMemcpy((void**)&devData, n*sizeof(float));
            CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT)); /* Set seed */
            CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen, 1234ULL)); /* Generate n floats on device */
            CURAND_CALL(curandGenerateUniform(gen, devData, n));
            cudaMemcpy(farr, devData, n * sizeof(float), cudaMemcpyDeviceToHost);
            tm();
            TopDownMergeSort(farr, fbrr, n);
        }
    }
```
printf("time for parallel execution is %f\n",tm());
tm();
test(n);
printf("time for parallel test is %f\n",tm());
}

float tStart=0.000000000f;
float tEnd=0.000000000f;
float tm() {
    tEnd = clock();
    float t = (tEnd - tStart) / CLOCKS_PER_SEC;
    tStart = tEnd;
    return t;
}

//parallel test function written in OpenMP
void test(int n){
    int i = 0;
    bool sorted = 1;
    /*for testing the array sorted*/
    #pragma omp parallel shared(A) private(i)
    {
        #pragma omp parallel for
        {
            //pragma omp critical
            if(A[i] < A[i-1]){
                printf("array not sorted at %d\n",i);
                break;
                sorted = 0;
            }
        }
    }
    if(sorted)
        printf("Array sorted\n");
}

/****************************start of the file for hybrid quick sort (HQuick sort) function (in OpenMP)  
/****************************/

HBQUICK SORT (IN OPENMP)

/*Hybrid quick sort in OpenMP, a program that sorts either integers or single precision floating point numbers in ascending order. Author: Hakan GOKAHMETOGLU, written for the thesis document,  
for Msc. Degree in Software Eng. In Atilim Univesirty */
#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <time.h>
#include <omp.h>
#include <cuda.h>
#include <curand.h>

#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\n    return EXIT_FAILURE;}} while(0)

template <typename T> void qs(int *a, int l, int r);

112
template<typename T>
void ss(T *a, const int n);  

template<typename T>  
int partition(T *a, const int l, const int r);  

float tm();  

int i, numThread=16, realCores=2, deferTreshold=4;  
int dt=8;  

template<typename T>
inline void swapss(volatile T *a, volatile T *b)
{
    T temp; temp = *a; *a = *b; *b = temp;  
}

template<typename T>
void ss(T *a, const int n){
    T temp; temp = a[0];
    unsigned int p=0;
    unsigned int idxx=0;  
    #pragma omp parallel shared(a,n,temp,p) private(idxx)
    {
        idxx = 0;
        #pragma omp parallel for  
        {
            for(idxx=0; idxx < n; idxx++)
            {
                temp = a[idxx];
                p=idxx;
                while (p > 0 && a[p-1] > temp)
                {
                    swapss(&a[p-1], &a[p]);
                    p--;
                }
                a[p] = temp;
            }
        }
    }
}

template<typename T>
int partition(T *a, const int l, const int r){
    const T x = a[r];
    int i = (l - 1);
    int j = 1;
    while (j <= r - 1 )
    {
        if (a[j] <= x)
        {
            swapss(&a[i++], &a[j]);
        }
        j++;
    }
    swapss(&a[i + 1], &a[r]);  
    return (i+1);  
}

template<typename T>
void qs(T *a, int l, int r){  
    if((r-1) <= dt)  
    {
        ss(a+l,(r-1)+1);  
    }
    else  
    {
        const int p = partition(a,l,r);  
    
    #pragma omp parallel sections num_threads(4)
    {
    
}
```c
#pragma omp parallel section
{
  if(l<p)
    qs(a,l,p-1);
}

#pragma omp parallel section
{
  if(p<r)
    qs(a,p+1,r);
}

template <typename T>
T compare (const void * a, const void * b)
{
  return (int)( *((T*)a) - *((T*)b) );
}

int main(int argc, char **argv )
{
  int arraySize = 1024;
  int nestDepth = 2;
  char bck='b';
  if(argc>1)arraySize=atoi(argv[1]);
  if(argc>2)dt=atoi(argv[2]);
  if(argc>3)bck=argv[3][0];

  size_t t = sizeof(int)*arraySize;
  int *a = (int*)malloc(t);
  int *b = (int*)malloc(t);

  curandGenerator_t gen;
  numThread = omp_get_max_threads();
  realCores = omp_get_num_threads();
  omp_set_nested(nestDepth);
  omp_set_num_threads(numThread);

  printf("Calculating with %d number of threads with %d real CPU(s)\n", numThread, realCores);

  if (bck=='b')
  {
    tm();
    #pragma omp parallel shared(a) private(i)
    {
      #pragma omp parallel for
      for (i = 0; i < arraySize; i++)
      {
        a[i] = arraySize - 1 - i;
        b[i] = arraySize - 1 - i;
      }
    }
    tm();
    #pragma omp parallel
    #pragma omp single
    qs(a,0,arraySize-1);
    printf(" qs time = %f \n",tm());
    qsort (b, arraySize, sizeof(int), compare);
    if (memcmp(b,a,t)==0)
    {
      printf("pass, array sorted\n");
    }
    else
    {
      size_t tf = sizeof(float)*arraySize;
    }
  }
```
float *devData;
float *fa = (float*)malloc(arraySize*sizeof(float));
float *fb = (float*)malloc(arraySize*sizeof(float));
cudaMalloc((void **)&devData, arraySize*sizeof(float));
CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT));
/* Set seed */
CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen, 1234ULL));
/* Generate n floats on device */
CURAND_CALL(curandGenerateUniform(gen, devData, arraySize));
cudaMemcpy(fa, devData, arraySize * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(fb, devData, arraySize * sizeof(float), cudaMemcpyDeviceToHost);
#pragma omp parallel
#pragma omp single
qs(fa,0,arraySize-1);
printf(" qs time = %f \n",tm());
qs(fa,0,arraySize, sizeof(float), compare);
if (memcmp(fb,fa,tf)==0)
{
 printf("pass, array sorted");
}
free(fa);
free(fb);
cudaFree(devData);
free(a);
free(b);
return 0;
}
float tStart=0.000000000f;
float tEnd=0.000000000f;
float tm() {
    tEnd = clock();
    float t = (tEnd - tStart) / 1000.00000000f;
    tStart = tEnd;
    return t;
}
/**************end of the file for hybrid quick sort(HBquick sort) function (in OpenMP)**************/

HOW TO BUILD THE CODES

OpenMP with CUDA compiler using VS2012 on Windows

Platform must be chosen as x64

Project → (project_name) properties → Configuration Properties → Linker → Input → additional dependencies → curand.lib; cudadevrt.lib; cudart.lib

CUDA with CUDA compiler using VS2012 on Windows

Platform must be chosen as x64

Project → (project_name) properties → Configuration Properties → CUDA C/C++ → Common → Generate Relocatable Device Code → yes

Project → (project_name) properties → Configuration Properties → CUDA C/C++ → Device → Code Generation → compute5,sm5
Project \(\rightarrow\) (project_name) properties \(\rightarrow\) Configuration
Properties \(\rightarrow\) Linker \(\rightarrow\) Input \(\rightarrow\) Additional
Dependencies \(\rightarrow\) curand.lib; cudadevrt.lib; cudart.lib;
APPENDIX D
SURVEY DOCUMENTS

Figure 40 shows the preview image for the spreadsheet document for SM study that has been made in the Chapter 2, Section 2 of this thesis.

Direct link is, https://docs.google.com/spreadsheets/d/1-N6SrSuCLSOMtYDccE0GveY0RGHw_eFwiNyrxtxBpn4/edit?usp=sharing

Or the shortened link, https://goo.gl/r4VNm3