8-3-2012

Efficient Algorithms for Sorting on GPUs

Seema M. Munavalli
sn2710@gmail.com

Efficient Algorithms for Sorting on GPUs

Seema Mahesh Munavalli

B.E., Visveswaraiah Technological University, Belgaum, 2003

A Thesis

Submitted in Partial Fulfillment of the

Requirements of the Degree of

Master of Science

at the

University of Connecticut

2012

APPROVAL PAGE

Master of Science Thesis

Efficient Algorithms for Sorting on GPUs

Presented by

Seema Mahesh Munavalli, BE.

Major Advisor _____

Dr. Sanguthevar Rajasekaran

Associate Advisor _____

Dr. Reda Ammar

Associate Advisor _____

Dr. Chun-Hsi Huang

University of Connecticut

2012

ABSTRACT

Efficient Algorithms for Sorting on GPUs

Seema Mahesh Munavalli

Major Advisor: Dr. Sanguthevar Rajasekaran

Sorting is an important problem in computing that has a rich history of investigation by various researchers. In this thesis we focus on this vital problem. In particular, we develop a novel algorithm for sorting on Graphics Processing Units (GPUs). GPUs are multicore architectures that offer the potential of affordable parallelism.

We present an efficient sorting algorithm called Fine Sample Sort (FSS). Our FSS algorithm extends and outperforms sample sort algorithm presented by Leischner[2], which is currently the fastest known comparison based algorithm on GPUs. The performance gain of FSS is mainly achieved due to the quality of the samples selected. By quantitative and empirical approach, we found out the best way to select the samples, which resulted in an efficient sorting algorithm. We carried out the experiment for different input distributions, and found out that FSS outperforms sample sort by at least 26% and on an average by 37% for data sizes ranging from 40 million and above across various input distributions.

DEDICATION

This work is dedicated to God, my parents, my husband Mahesh Munavalli, my advisor Dr. Sanguthevar Rajasekaran, Dr. Reda Ammar, faculty and staff at university of Connecticut. I would like to thank each one of the above for their patience, support and encouragement.

ACKNOWLEDGEMENTS

I would like to thank my Advisor Dr. Sanguthevar Rajasekaran for his guidance, support and encouragement throughout the course of this research. I would also like to thank my committee members, Dr. Reda Ammar and Dr. Chun-Hsi Huang for their support.

I would specially thank Dr. Sanguthevar Rajasekaran and Dr. Reda Ammar for providing me funding at UConn.

Finally I would like to thank my parents, my husband Mahesh Munavalli, friends, staff and faculty at UConn for their constant support and understanding.

Table of Contents

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the world of computing there exist many challenging problems. One among them is sorting. Efficient sorting is crucial, as many applications depend on them. Some applications of sorting follow:

- Binary search in databases.
- Many problems in computer graphics and computational geometry.
- Motif search in computational biology; and so on.

Hence, it is of utmost importance to design efficient sorting algorithms for emerging architectures, which can exploit architectural features. One such emerging architecture is the Graphics Processing Unit (GPU).

Graphics Processing Units (GPUs) are massively parallel many-core processors. They work on SIMD (Single Instruction Multiple Data) model. CUDA (Compute Unified Device Architecture) is the computing engine in Nvidia graphics processing units, which allows developers to code algorithms for execution on GPUs, through variants of industry standard programming languages such as C and C++. Reasonable cost and massively parallel computation capability have resulted in an explosion of research directed towards expanding the applicability of GPUs to a wide variety of high-performance computing applications such as sorting.

.

Utilizing the advantages of the architectural attributes offered by GPUs, we were able to develop and implement a parallel sorting algorithm named Fine Sample Sort (FSS). Our experimental study demonstrates that FSS outperforms sample sort developed by Leischner [2], which is currently the fastest known comparison-based GPU sorting algorithm.

Performance of FSS depends crucially on the quality of splitters selected. By empirical and quantitative approach, we were able to find the best way to select splitters. These splitters reduced the number of sort phases needed to ultimately reach a bucket size that can be locally sorted in the shared memory of GPUs.

For the performance analysis we have experimented with commonly accepted set of input distributions. The various distributions considered are normal distribution, Poisson distribution, uniform distribution, and descending order data set. Our results indicate that Fine Sample Sort outperforms sample sort by at least 26% and on an average by 37% for data sizes ranging from 40 million and above across various input distributions.

CHAPTER 2

GRAPHICS PROCESSING UNIT (GPU) AND COMPUTE UNIFIED DEVICE ARCHITUCTURE (CUDA)

Introduction

There are many challenging computing problems that are impractical to be solved in a reasonable amount of time with single processors. These applications can be solved in a reasonable amount of time using architectures that support many-core processors. One such architecture is Graphics Processing Unit. In this chapter we will study about GPU architecture, along with CUDA (Compute Unified Device Architecture) which is a software environment for GPUs.

2.1 Graphics Processing Unit (GPU)

A Graphics Processing Unit (GPU) is a specialized electronic circuit, which is massively parallel with many-core processors. GPUs belong to single instruction, multiple data (SIMD) class of parallel computers that perform the same operation on multiple data simultaneously to exploit data level parallelism.

2.2 The Need for GPUs

There are many advantages of GPU's. To list a few:

- Traditionally, most graphics operations, such as transformations between coordinate spaces, lighting and shading operations have been performed on the CPU. There is a need to offload many of these operations from the CPU (primarily arithmetic and logic) to specialized graphics hardware (based on vector & matrix processing).

- Parallelism is the future of computing and the GPU architecture is well suited for parallel applications.
- It exhibits a high performance at a low cost.

2.3 GPU Computing

The model for GPU computing is to use a CPU and GPU together in a heterogeneous co-processing mode. The sequential part of the application runs on the CPU and the computationally-intensive part runs on GPU. From the user's view the application runs faster because it is using the high-performance of the GPU to boost performance. A typical mode of operation of a GPU follows:

1. Copy data from the main CPU memory to GPU memory
2. CPU sends processing instructions to the GPU
3. GPU cores execute instructions in parallel
4. Copy the result from GPU memory to main memory

Figure 1 provides a summary of the CUDA.

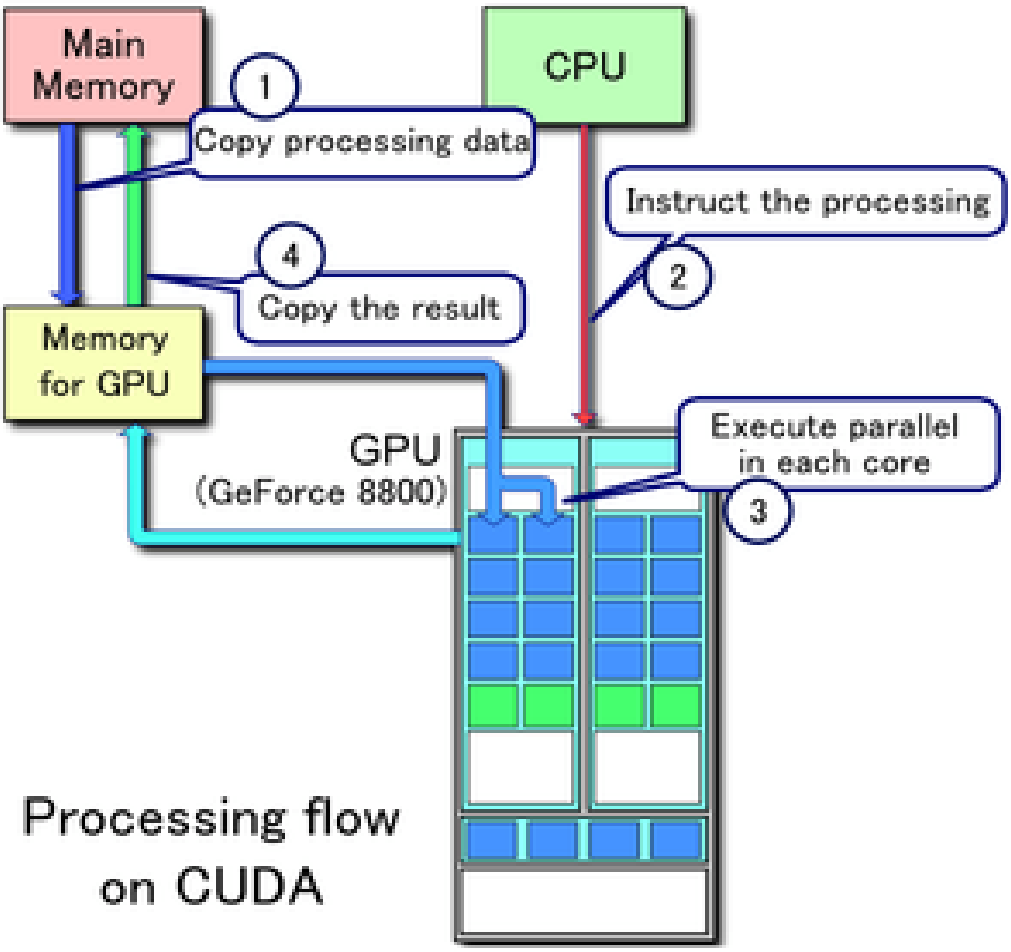

Figure 1: Processing flow on CUDA (Compute Unified Device Architecture)[1]

2.4 GPU Architecture

Graphics processing units (GPUs) are massively parallel many-core processors. Below are the specifications of NVIDIA's GTX 480 and NVIDIA's Tesla GPU's:

NVIDIA's GTX 480 GPUs: These GPUs have 480 scalar processing cores (SPs) per chip. These cores are partitioned into 15 Streaming Multiprocessors (SMs). Each SM comprises of 32 SPs. Each SM shares a 48KB local memory (called shared memory) that

[1] http://en.wikipedia.org/wiki/File:CUDA_processing_flow_%28En%29.PNG

may be utilized by the threads running on this SM. GTX 480 has a 1536MB global memory. Figure 2 presents some details.

NVIDIA's Tesla GPUs: These GPUs have 240 scalar processing cores (SPs) per chip [4]. These cores are partitioned into 30 Streaming Multiprocessors (SMs). Each SM comprises of 8 SPs. Each SM shares a 16KB local memory (called shared memory) and the 240 on-chip cores also share a 4GB off-chip global (or device) memory. Figure 3 shows the various components of TESLA GPUs.

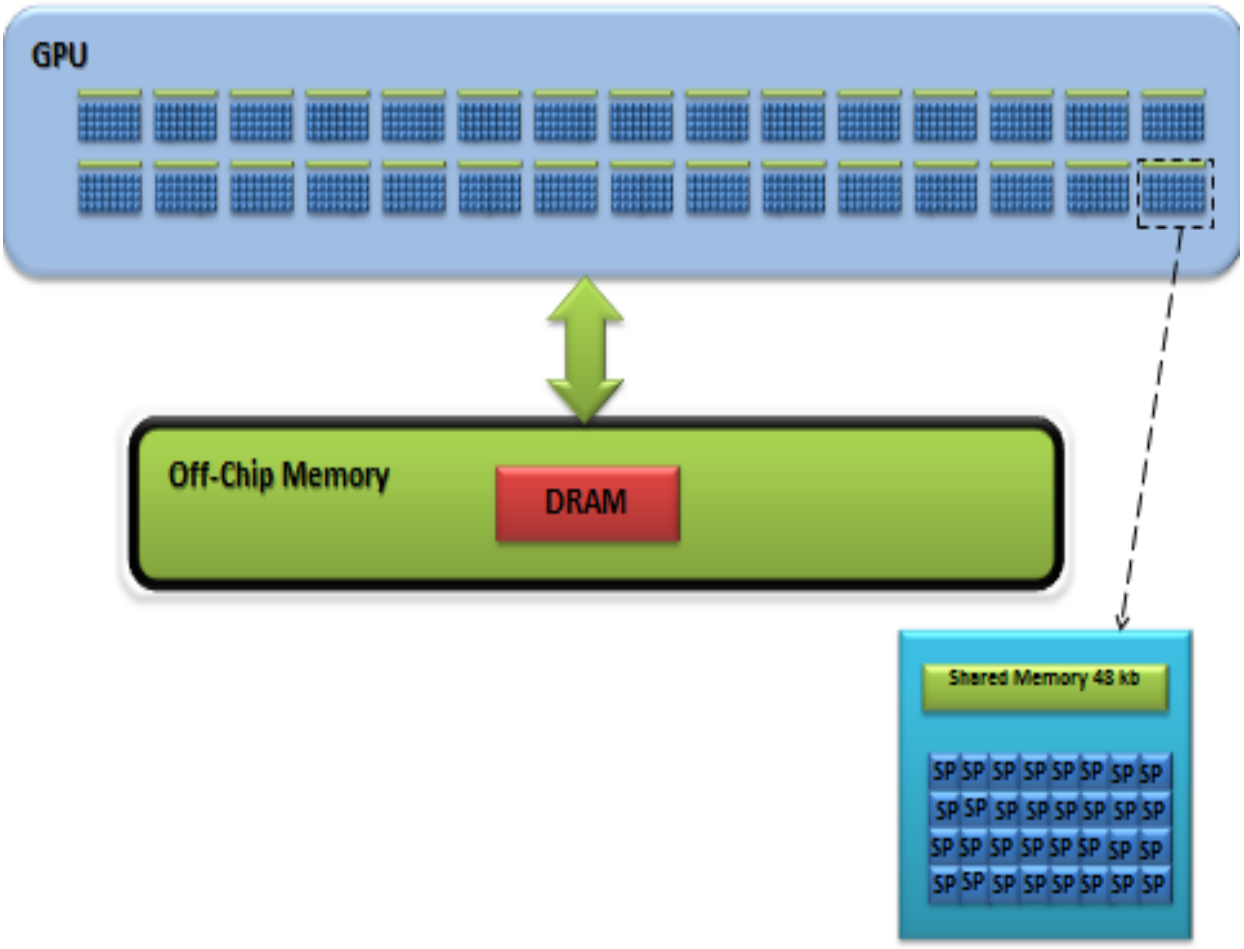

Figure 2 : NVIDIA's GTX 480 GPUs

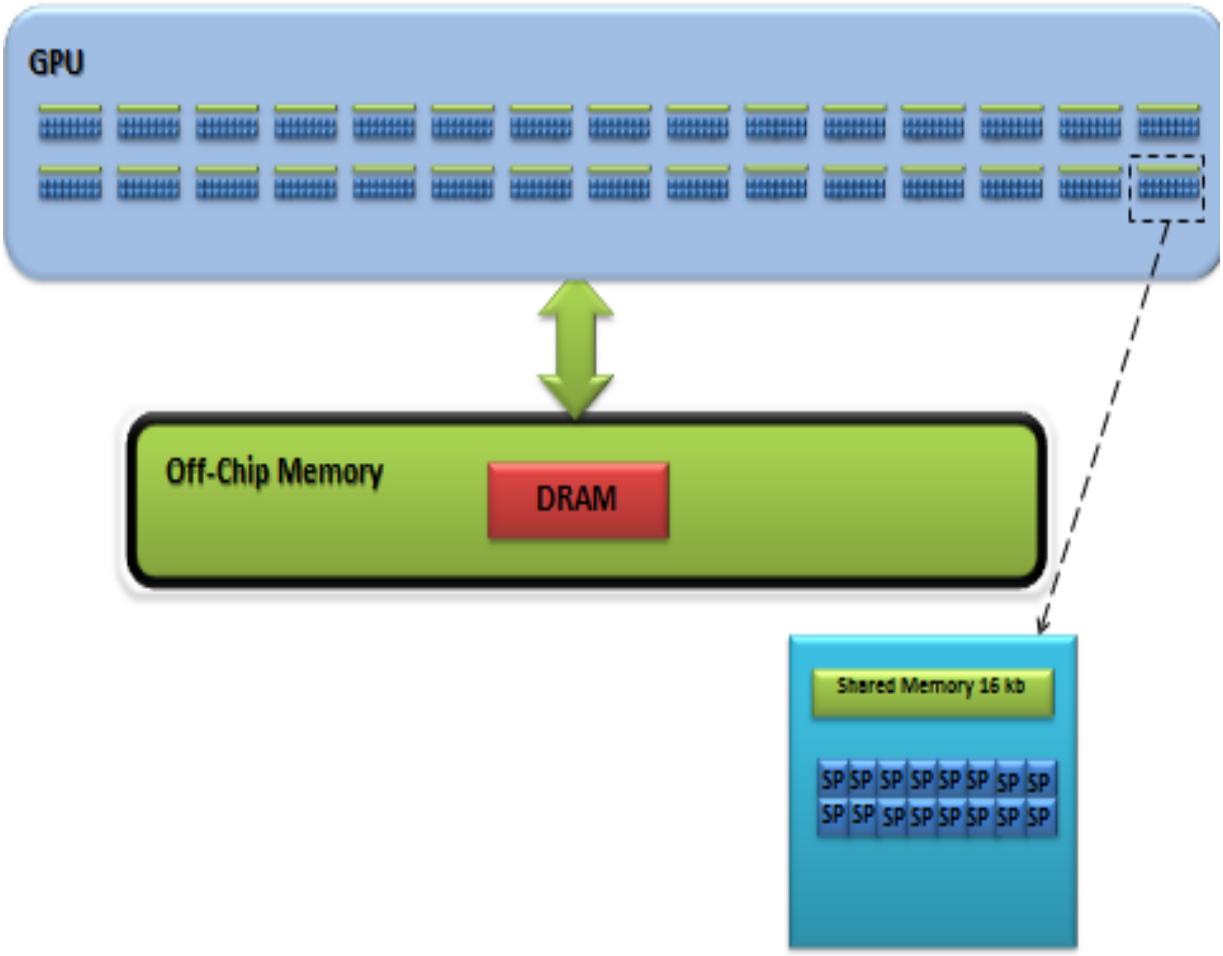

Figure 3: NVIDIA's Tesla GPUs

2.5 CUDA (Compute Unified Device Architecture)

CUDA is a scalable parallel programming model and a software environment for parallel computing. This is an extension of the familiar C/C++ environment. It follows the heterogeneous serial-parallel programming model. CUDA also maps well to multicore CPUs.

2.5.1 Kernel

Kernel is a normal C function that is called on the host machine (CPU) and runs on device machine (GPU). One kernel is executed at a time. Invocation of a kernel executes many threads, through which parallel computing is achieved.

2.5.2 Thread

This is an execution of a kernel with a given index. Each thread uses its j index to access elements in, such that the collection of all threads cooperatively processes the entire data set.

CUDA threads have the following properties:

- CUDA threads are extremely lightweight
- They have very little creation overhead
- Instant switching is supported

2.5.3 Block

This is a group of threads. We can coordinate the threads using the _syncthreads() function that makes a thread stop at a certain point in the kernel until all the other threads in its block reach the same point.

2.5.4 Warp

Warp is a group of 32 threads, to be executed in SIMD fashion by a CUDA SM.

2.5.5 Grid

Group of blocks together form a grid.

Figure 4 displays the relationships among the different components of CUDA. Figures 5 and 6 portray some execution features of CUDA.

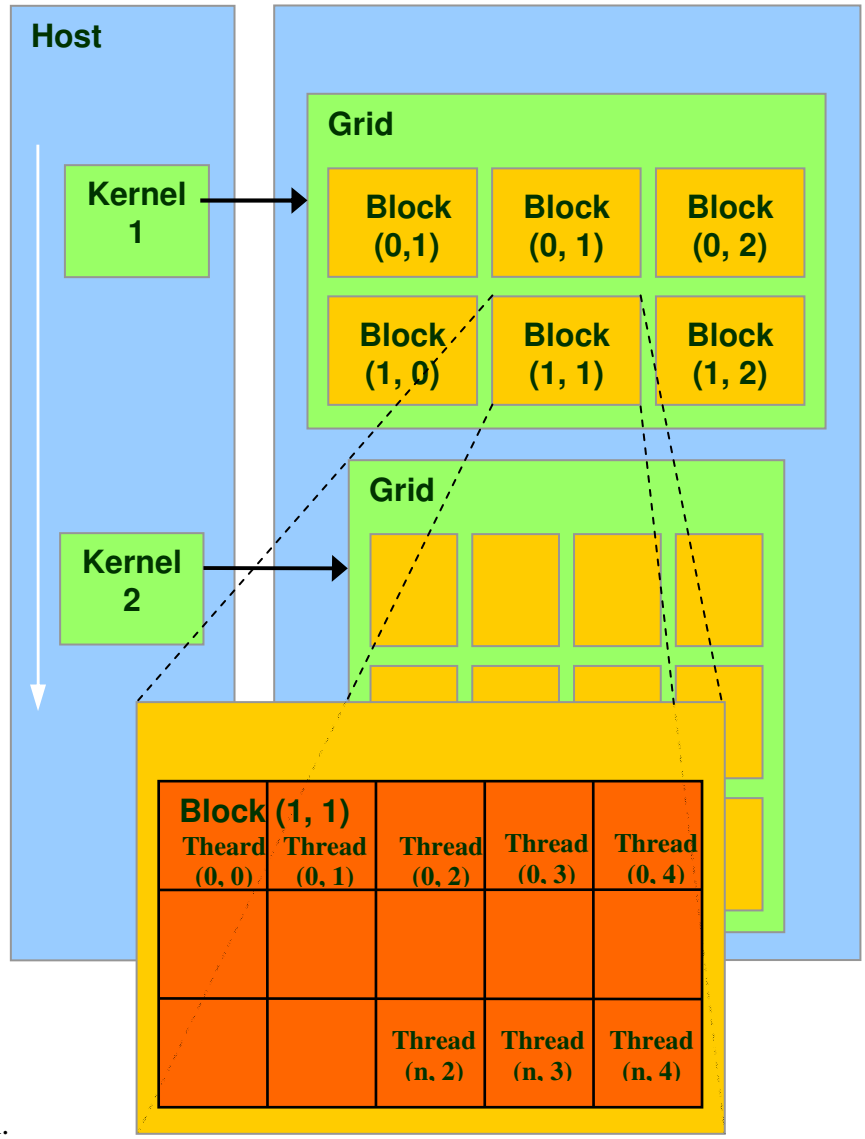

Figure 4 : Organization of threads, thread block and grid.[2]

2.5.6 Arrays of Parallel threads

A CUDA kernel is executed by an array of threads

---

[2] Bandyopadhyay, S. and Sahni, S., GRS - GPU Radix Sort for Large Multifield Records, International Conference on High Performance Computing (HiPC), 2010

- All threads run the same code
- Each thread has an ID that it uses to compute memory addresses and make control decisions

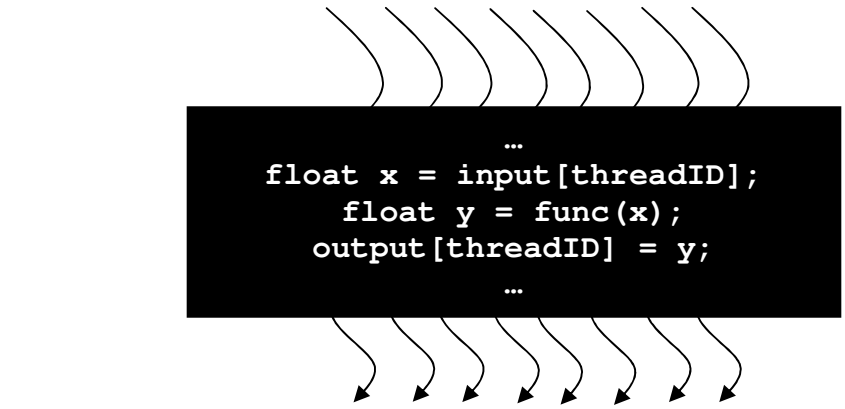

Figure 5 : Computation of address via thread.[3]

2.5.7 Thread Batching

In CUDA, kernel launches a grid of thread blocks. These thread blocks are executed on SMs and threads within each block are executed on scalar processors. The threads within a block can cooperate via shared memory, but threads in different blocks cannot cooperate.

[3]http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf

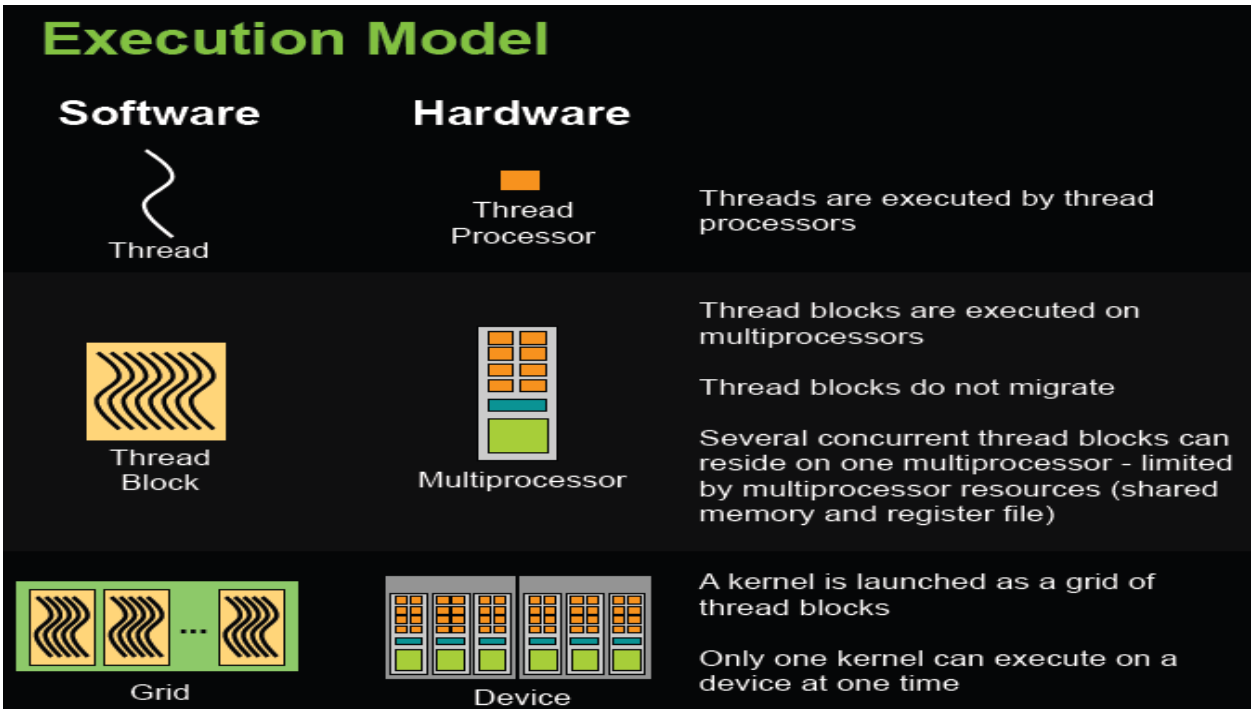

Figure 6: The Execution Model[4]

2.5.8 Managing Memory

CPU and GPU have separate memory spaces. There is no direct provision to access GPU memory. In order for any application to utilize the GPUs, the data should be first stored in CPU memory. Host (CPU) code manages to transfer data to device (GPU) memory and then parallel computations are performed on the GPU. It is the responsibility of the host code to manage allocation and de-allocation of memory of the GPUs.

---

[4]http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf

2.5.9 Thread Life Cycle in hardware

1. Launch CUDA kernel as below

   KernelFunction<<<dimGrid, dimBlock>>>(…);

   Then thread blocks are serially distributed to all the SMs with potentially >1 thread blocks per SM.
2. The thread blocks in each SM are launched as warps of threads by hardware after invocation of kernel. SM schedules and executes warps that are ready to run on the scalar processors.
3. Parallel computations are performed by threads in the GPU.
4. Kernel exits after the completion of parallel computation.

2.5.10 Thread Scheduling/Execution

Warps are scheduling units in an SM. Each thread block is divided into 32-thread warps and the threads within the warps are executed on scalar processors. For example:

If 3 blocks are assigned to an SM and each Block has 256 threads, the number of warps=8 (256/32). This implies that for 3 blocks, we have 24 warps. At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.

SM Warp Scheduling

- SM hardware implements zero-overhead warp scheduling
- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy

- All the threads in a warp execute the same instruction, when selected.

2.6 Factors impacting the performance of applications

- Non-common instructions within a warp are serialized. So avoiding thread divergence within a warp should be considered.
- Global memory is very expensive than access to registers or shared memory. Data to be used several times should be read once from the global memory and stored in registers or shared memory for future use.
- When the threads of a half-warp access global memory, this access is accomplished via a series of memory transactions. The number of memory transactions equals the number of different 32-byte (64-byte, 128-byte, 128-byte) memory segments that the words to be accessed lie in, when each thread accesses an 8-bit (16-bit, 32-bit, 64-bit) word. Given the cost of a global memory transaction, it pays to organize the computation so that the number of global memory transactions made by each half warp is minimized.
- Shared memory is divided into banks in a round robin fashion using words of size 32 bits. When the threads of a half warp access shared memory, the access is accomplished as a series of one or more memory transactions. Let S denote the set of addresses to be accessed. Each transaction is built by selecting one of the addresses in S to define the broadcast word. All addresses in S that are included in the broadcast word are removed from S. At a time only one address from

each of the remaining banks is removed from S. The set of removed addresses is serviced by a single memory transaction. Since the user has no way to specify the broadcast word for maximum parallelism, the computation should be organized so that, at any given time, the threads in a half warp access either words in different banks of shared memory or they access the same word of shared memory.

2.7 Good and Bad candidates for a GPU

Good candidates for a GPU

Data-parallel computations that involve more arithmetic operations compared to memory operations attain maximum performance on GPUs. This is because the volume of very fast arithmetic instruction can hide the relatively slow memory accesses. For example, extraction of endmembers in a hyper spectral image is a good candidate for GPU, as we can exploit data parallelism.

Bad candidates for a GPU

In particular, task-parallel computations which execute different instructions on the same or different data cannot efficiently utilize the hardware on a GPU as it often ends up running sequentially.

Conclusion

In this chapter we explored multicore GPUs and CUDA software environment through which we can devise efficient parallel algorithms that are based on SIMD principle.

CHAPTER 3

PREVIOUS WORK ON SORTING ON GPUs

Introduction

Since sorting is one of the most widely studied and challenging problems in computer science, emerging architectural capabilities of graphics processors brought considerable attention to sorting on GPUs. In this chapter, we will explore prevailing comparison based sorting algorithms on GPUs and Lemma.

3.1 Prior Works

Cederman, et al. [8] have adapted quick sort for GPUs. Their adaptation first partitions the sequence to be sorted into subsequences, sorts these subsequences in parallel, and then merges the sorted subsequences in parallel.

Satish, et al. [3] have developed an even faster merge sort. In this merge sort, two sorted sequences A and B are merged by a thread block to produce the sequence C, when A and B have less than 256 elements each. Each thread reads an element of A and then does a binary search on the sequence B with that element to determine where it should be placed in the merged sequence C. When the number of elements in a sequence is more than 256, A and B are divided into a set of subsequences by using a set of splitters. The splitters are chosen from the two sequences in such a way that the interval between successive splitters is small enough to be merged by a thread block.

GPU sample sort was developed by Leischner, et al. [2]. Sample sort is reported to be about 25% faster than the best comparison based sorting algorithm, merge sort of [3], and on average more than 2 times faster than GPU quicksort.

3.2 Fine Sample Sort

The basic idea behind the Fine Sample Sort algorithm is the following Lemma, which was developed by Reif and Valiant [13].

Lemma: Let T be an ordered set and S1 be a random sample of T of size $n^2 2^{m/3}$. Sort S1 and select elements in positions n^2, $2n^2$, …, $(2^{m/3} -1)\ n^2$. Let these keys be in the list S2. Keys in S2 partition T. Let q be the maximum size of any of these parts. Then,

$Pr[q > (1 + n^{-1/3})|T|/2^{m/3}] < 2^{-c1n}$

for some constant $c1 > 0$ and

$Pr[q < (1 - n^{-1/3})|T|/2^{m/3}] < 2^{-c2n}$

for some constant $c2 > 0$ under the assumption that $n^4 2^{m/3} = o(|T|)$ and that $n^2 2^{m/3} = o(|T|^{2/3})$.

Proof: The probability that an element in T is sampled in S1 is $p = \frac{S1}{|T|} = (n^2 2^{m/3}) / |T|$

Split T into segments of size d. Let X = B(d,p) be a binomial random variable counting how many elements in a region of length d from T are present in S1. The mean of X is $\mu = dp = dn^2 2^{m/3}/|T|$. The probability that one of the final partitions is greater than d is the same as the probability that a region of size d contains less than n^2 elements from S1.

Prob[part > d] = Prob[X < n2]. In order to apply the Chernoff bounds we

compute the following:

$(1 -\varepsilon)\ \mu = n^2 \Rightarrow 1- \varepsilon = |T|/d2^{m/3} = (d2^{m/3}-|T|)/\ d2^{m/3}$

We apply the Chernoff bound:

$Prob[X < n^2] < \exp(-\varepsilon^2\ \mu/2)$

$=\exp(-((d2^{m/3}-|T|)/\ d2^{m/3})^2\ (dn^2 2^{m/3})/2|T|)$

$=\exp(-(nd2^{m/3}-|T|)^2 n^2)\ /2d|T|2^{m/3})$

The probability that there exists any part of length d with less than n^2 elements of S1 is $<=|T|/d(Prob[X < n^2])$.If q is the maximum size of the final parts, then:

$Prob[q > d] <=(|T|/d)\ \exp(-(nd2^{m/3}-|T|)^2 n^2)\ /2d|T|2^{m/3})$

Now, if we use $d = (1 + {}^{n-1/3})|T|/2^{m/3}$ we have:

$Prob[q > (1 + n^{-1/3})|T|/2^{m/3})] <= (2^{\ m/3}/(1+ n^{-1/3}))\exp(-((n(1+n^{-1/3})|T|-|T|)^2 n^2)/\ 2(1+n^{-1/3)|}|T|^2))$

$<= (2^{\ m/3})\exp(-((n(1+n^{-1/3})-1)^2 n^2)/\ 2(1+n^{-1/3}))$

$<= (2^{\ m/3})\exp(-(n^2(1+n^{-1/3})^2 n^2)/\ )/\ 2(1+n^{-1/3}))$

$<=(2^{\ m/3})\exp(-n^4(1+n^{-1/3})/2)$

$<= (2^{\ m/3})\exp(-n^4(1+1)/2)$

$<=(2^{\ m/3})\exp(-n^4)$

We have used the fact that $n^{-1/3} < 1$ for $n > 1$. As long as

$m/3 - n^4/2 < -n$ we have the bound required in the problem. □

Conclusion

In this chapter we explored previous works done on comparison based sorting algorithms and a Lemma by Reif and Valiant. With this prevailing knowledge of existing lemma and previous existing comparison based sorting algorithms on GPUs, we have devised an efficient comparison based sorting algorithm.

CHAPTER 4

FINE SAMPLE SORT ALGORITHM

Introduction

In this chapter we will look at the sequential version of Fine Sample Sort (FSS) algorithm, followed by Fine Sample Sort algorithm on GPUs along with the data used for the experiments and their results.

4.1 Sequential Fine Sample Sort algorithm overview

```
SampleSort(e[1, . . . .n], 128)
 begin
    //M is the data size that fits in the shared memory of GPU
      if  (n < M )
            {
               return SmallSort(e)
             }
       else
             {
             Choose a random sample S of size 128[0.005(n 1/3)]
             Sort the random sample.
             From the sorted list pick elements that are in positions  0.005(n 1/3),
              2[0.005(n 1/3), 3[0.005(n 1/3)],…, 127[0.005(n 1/3)]

             for 1 ≤ i ≤ n do
```

{

Find j ∈ {1, . . . , 128}, such that

b1 = {e[i] ∈ e |e[i]<= s1 },

b_j ={e[i] ∈ e | sj−1 < e[i] ≤ sj }, for 2<= j < =127;

b_{128} = {e[i] ∈ e | e[i] >s127} }

return Concatenate(SampleSort(b_1, 128), . , SampleSort(b_{128}, 128)

}

end

4.2 Fine Sample Sort algorithm for GPUs

In order to efficiently map our computational problem to a GPU architecture, we decompose it into data-independent sub problems that can be processed in parallel by blocks of concurrent threads. We divide the input into p = (Input size)/ (No of elements to be processed in a block) tiles, i.e., we divide the input into p = ⌈n/ (t · ℓ) ⌉ tiles. By choosing 256 threads per block and assigning 8 elements per thread, balance is achieved between the parallelism exposed by the algorithm and memory latency.

A high-level description of FSS follows:

Phase 1: Choose the size of the splitters to be 128[0.005(input size)$^{1/3}$]

Phase 2: Sort the chosen splitters

Phase 3: From the sorted list of splitters select elements that are in positions

0.005($n^{1/3}$), 2[0.005($n^{1/3}$)], 3[0.005($n^{1/3}$)], …, 127[0.005($n^{1/3}$)]. These selected elements form the splitter set.

Phase 4: Each block loads the splitters set into its fast private shared

memory of GPU and each thread block computes the bucket indices for all the elements in its tile. It counts the number of elements in each bucket and stores this per-block k-entry histogram in the global memory.

Phase 5: Perform a prefix sum over the k × p histogram tables stored in a column-major order to compute global bucket offsets in the output.

Phase 6: Since storing the bucket indices in global memory is not faster than just recomputing them, each thread block again computes the bucket indices for all the elements in its tile, computes their local offsets in the buckets and finally stores elements at their proper output positions using the global offsets computed in the previous step.

Sorting of buckets is delayed until the whole input is partitioned into buckets of size at most M (for some relevant value of M), so that the buckets fit in the shared memory. For buckets of size less than M we use quicksort by Cederman and Tsiga [8]. To improve load-balancing, buckets are scheduled for sorting ordered by size.

4.3 Experimental study

The implementation was done using an Intel core i7 processor @3.20 GHz with 16 GB RAM on 64 bit windows 7 professional operating system. We used NVidia GTX 480 GPU for our experiment. The NVIDIA GeForce GTX 480 has a compute capability of 2.0. It was released in March 2010. It has 32 K registers per multiprocessor. The data bus for the GTX 480 GPU is PCI-E 2.0, whose maximum bandwidth is 16 GB/s. Theoretical maximum global memory bandwidth of the GTX 480 GPU is 177.4 GB/s. DRAM access has high latency due to limited bandwidth, but on-chip memory access is much faster.

The GTX 480 GPU has 32 shared memory banks and the bandwidth of shared memory is 1.344TB/s and the global memory size is 1535 MB.

For the analysis of runtime we have used the following commonly accepted set of distributions and compared our FSS algorithm with sample sort, which is currently the best known comparison based sorting algorithm for GPUs.

Uniform distribution – When we employed a uniformly distributed random input in the range [10M-100M] FSS was 47% faster and on an average 55% faster for data sizes of 40M and above.

Poisson distribution- When the data was Poisson distributed in the range [10M-100M] FSS was at least 26% faster and on an average 37% faster for data sizes of 40M and above.

Normal distribution- When the input data had a Normal distribution in the range [10M-100M], FSS was at least 49% faster and on an average 57% faster for data sizes of 40M and above.

Descending ordered – When the data was already sorted in descending order in the range [10M-100M], FSS was at least 48% faster and on an average of 59% faster for data sizes of 40M and above.

4.4 Experimental Results

Our experimental results are summarized in Figures 7, 8, 9, and 10. Graphs are plotted for random samples of different sizes to emphasize how the selection of sample size matters in phase 1 of the algorithm.

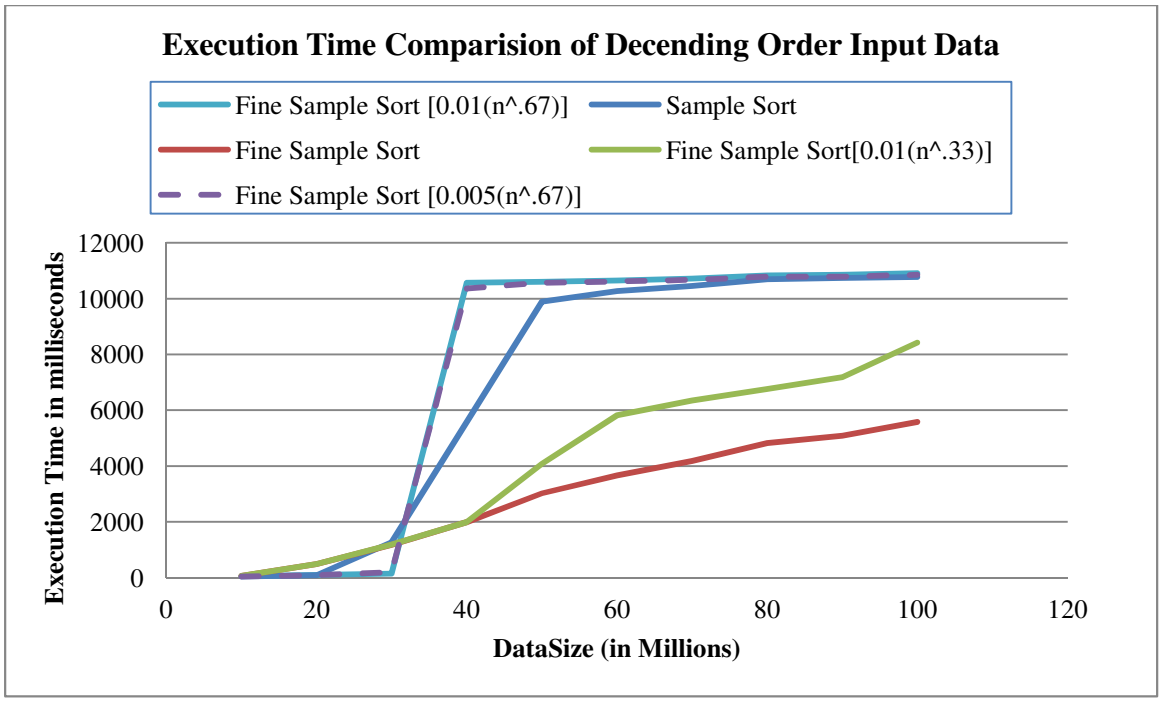

Figure 7: Execution time comparison for descending order input data.

Table 1: Run times of the sample sorting algorithm for descending order input (Ref: Figure 7)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	42.79	92.93	1269.73	5583.61	9895.57
2	42.61	93.57	1269.93	5584.06	9894.52
3	42.43	93.62	1270.18	5583.68	9892.54
4	42.74	93.61	1270.22	5584.33	9894.55
5	42.45	93.59	1270.07	5583.79	9892.28
6	42.43	92.58	1270.09	5584.71	9893.76
7	42.61	93.68	1270.23	5583.86	9894.08
8	42.47	93.51	1270.25	5584.76	9892.60
9	42.56	93.06	1270.07	5582.36	9892.96
10	42.59	93.18	1270.26	5588.22	9893.71
Average run time (milliseconds)	**42.57**	**93.33**	**1270.10**	**5584.34**	**9893.66**

Table 2: Run times of sample sorting algorithm for descending order input (Ref: Figure 7)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	10265.90	10453.00	10689.30	10744.80	10780.30
2	10265.60	10454.70	10690.20	10745.80	10780.80
3	10265.20	10454.80	10689.80	10746.40	10778.70
4	10266.30	10453.80	10696.60	10749.10	10780.50
5	10265.80	10455.40	10690.00	10746.00	10781.00
6	10265.30	10455.20	10689.70	10753.30	10779.90
7	10268.50	10455.70	10691.60	10745.00	10779.70
8	10265.50	10453.60	10690.30	10746.70	10779.90
9	10267.10	10454.90	10690.30	10746.10	10780.60
10	10266.40	10453.80	10688.80	10745.20	10780.30
Average run time (milliseconds)	**10266.16**	**10454.49**	**10690.66**	**10746.84**	**10780.17**

Table 3 : Run times of Fine Sample Sort algorithm for descending order input (Ref: Figure 7)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	67.62	483.77	1183.44	1985.75	3021.18
2	68.22	484.03	1182.51	1986.59	3024.74
3	68.04	484.01	1182.18	1986.92	3024.56
4	67.58	483.91	1183.04	1986.95	3024.62
5	67.86	484.19	1181.99	1986.83	3024.59
6	67.72	484.08	1182.19	1986.43	3024.47
7	67.76	484.49	1182.35	1986.93	3023.94
8	67.60	484.04	1181.89	1986.66	3023.21
9	68.07	484.02	1182.61	1986.76	3024.74
10	67.58	484.20	1182.62	1987.12	3024.04
Average run time (milliseconds)	**67.81**	**484.07**	**1182.48**	**1986.69**	**3024.01**

Table 4 : Run times of Fine Sample Sort algorithm for descending order input (Ref: Figure 7)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	3669.14	4179.98	4824.77	5081.92	5577.71
2	3671.05	4178.55	4820.40	5084.20	5579.29
3	3671.00	4179.42	4820.09	5083.84	5579.17
4	3669.95	4179.22	4820.88	5084.12	5588.15
5	3669.46	4179.38	4820.65	5083.93	5579.64
6	3671.11	4179.20	4820.64	5083.03	5586.93
7	3670.13	4179.64	4819.30	5084.23	5579.64
8	3670.16	4179.16	4820.23	5081.81	5580.53
9	3670.16	4179.16	4819.77	5084.07	5580.13
10	3670.11	4179.31	4819.64	5084.27	5587.49
Average run time (milliseconds)	**3670.23**	**4179.30**	**4820.64**	**5083.54**	**5581.87**

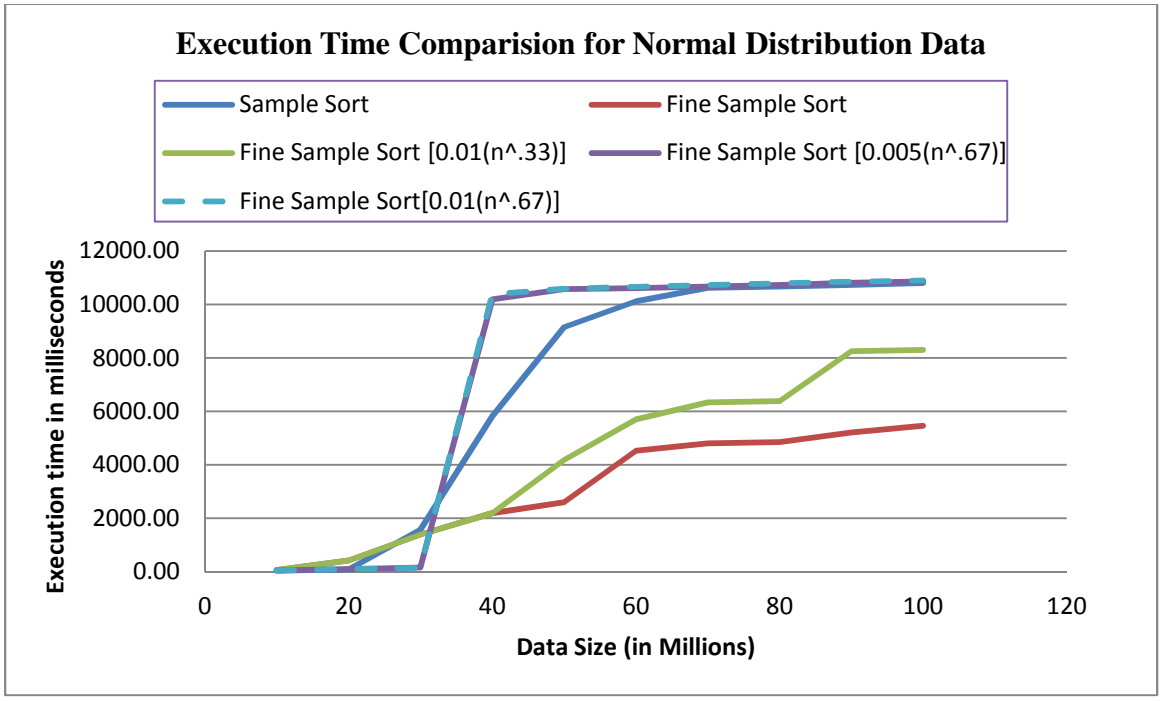

Figure 8: Execution time comparison for normal distribution input

Table 5 : Run times of Sample Sort algorithm for Normal distribution data (Ref: Figure 8)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	44.61	93.09	1576.21	5823.58	9153.79
2	44.44	92.29	1576.80	5819.87	9160.33
3	43.93	91.54	1577.00	5819.61	9147.42
4	44.05	91.30	1576.89	5819.34	9147.14
5	44.43	91.77	1577.17	5821.46	9157.63
6	43.99	91.83	1576.59	5822.14	9146.96
7	44.04	91.51	1578.07	5825.59	9147.58
8	43.83	91.50	1578.10	5822.13	9151.41
9	43.70	90.82	1577.06	5820.16	9150.86
10	43.74	90.93	1577.18	5823.29	9147.93
Average run time (milliseconds)	**44.08**	**91.66**	**1577.11**	**5821.72**	**9151.11**

Table 6 : Run times of Sample Sort algorithm for Normal distribution data (Ref: Figure 8)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	10112.20	10622.30	10670.80	10735.40	10812.00
2	10118.60	10626.60	10670.20	10735.60	10807.50
3	10129.70	10624.80	10669.50	10735.60	10806.50
4	10118.50	10624.70	10667.60	10735.20	10807.40
5	10118.60	10626.50	10674.70	10734.90	10807.20
6	10120.20	10624.90	10667.40	10742.20	10808.30
7	10121.00	10628.20	10680.10	10737.90	10807.40
8	10118.90	10625.40	10666.60	10735.60	10813.00
9	10122.00	10625.10	10669.70	10740.90	10814.20
10	10120.80	10628.30	10667.00	10735.50	10807.60
Average run time (milliseconds)	**10120.05**	**10625.68**	**10670.36**	**10736.88**	**10809.11**

Table 7 : Run times of Fine Sample Sort algorithm for Normal distribution data (Ref: Figure 8)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	70.87	427.66	1394.89	2195.57	2602.12
2	70.55	427.61	1395.11	2196.95	2604.17
3	71.21	427.23	1394.88	2196.91	2603.98
4	70.68	428.28	1394.72	2197.21	2604.22
5	70.70	428.75	1393.15	2197.42	2603.35
6	70.68	427.44	1394.52	2198.93	2604.01
7	70.68	427.30	1395.89	2195.40	2602.31
8	71.09	427.45	1394.65	2196.82	2604.44
9	70.77	427.26	1395.15	2196.72	2604.38
10	70.93	428.85	1394.15	2196.72	2603.75
Average run time (milliseconds)	**70.82**	**427.78**	**1394.71**	**2196.87**	**2603.67**

Table 8 : Run times of Fine Sample Sort algorithm for Normal distribution data (Ref: Figure 8)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	4532.19	4803.02	4850.82	5198.49	5464.93
2	4532.43	4806.47	4849.27	5309.32	5467.41
3	4534.46	4807.75	4850.03	5201.26	5465.36
4	4532.69	4806.85	4848.29	5201.41	5468.26
5	4534.75	4811.28	4856.22	5202.10	5466.67
6	4534.85	4805.83	4849.16	5199.67	5467.87
7	4534.00	4807.73	4852.07	5202.15	5467.44
8	4537.65	4806.54	4851.05	5200.69	5466.00
9	4533.71	4806.99	4849.75	5201.40	5465.73
10	4535.02	4807.56	4849.76	5200.01	5466.39
Average run time (milliseconds)	**4534.18**	**4807.00**	**4850.64**	**5211.65**	**5466.61**

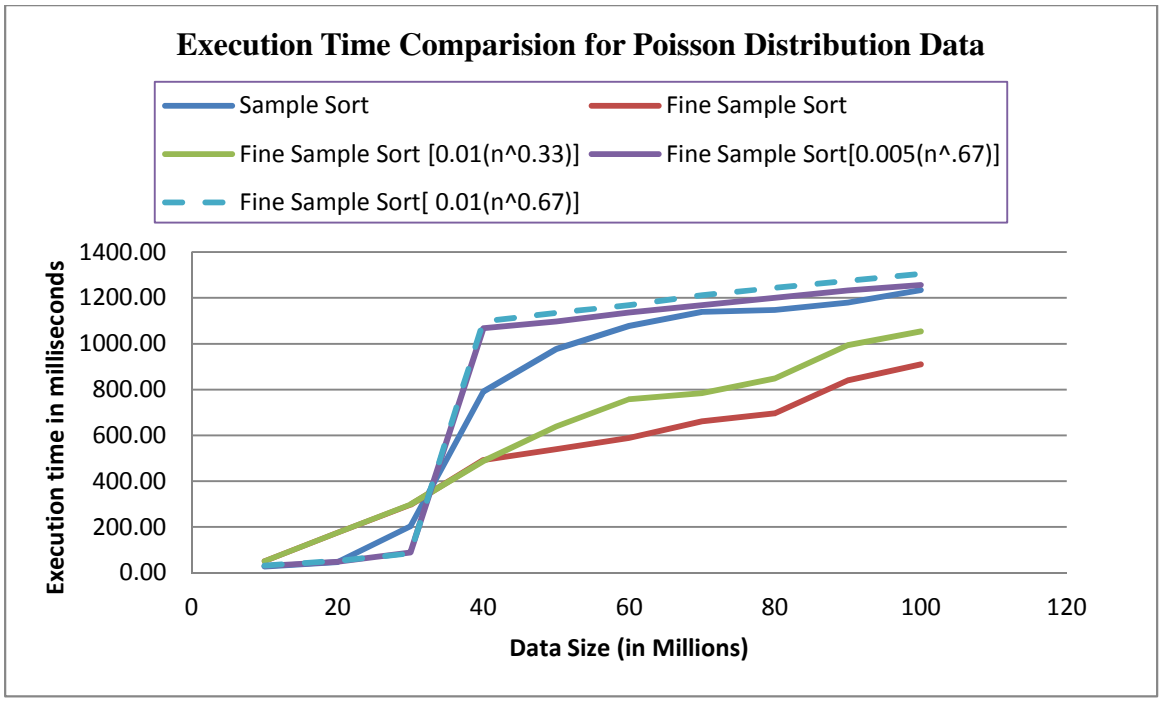

Figure 9: Execution time comparison for Poisson distributions input

Table 9 : Run times of Sample Sort algorithm for Poisson distribution data (Ref: Figure 9)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	26.36	47.27	203.53	788.28	976.93
2	26.22	46.65	203.53	792.63	976.82
3	26.38	46.92	204.49	788.84	979.63
4	26.41	47.54	203.74	790.11	978.64
5	28.13	47.51	203.74	788.54	975.86
6	28.10	47.56	203.51	790.50	973.27
7	28.19	47.64	203.05	791.26	977.29
8	28.15	47.80	203.41	788.46	980.29
9	28.22	48.03	204.02	792.23	973.03
10	27.21	47.87	203.96	794.62	977.30
Average run time (milliseconds)	**27.34**	**47.48**	**203.70**	**790.55**	**976.91**

Table 10 : Run times of Sample Sort algorithm for Poisson distribution data (Ref: Figure 9)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	1078.04	1134.42	1143.54	1198.32	1237.55
2	1072.45	1139.05	1144.23	1177.11	1232.42
3	1076.11	1137.45	1145.50	1186.17	1233.42
4	1078.20	1137.81	1163.95	1171.09	1232.92
5	1078.89	1134.42	1144.51	1171.74	1237.33
6	1077.21	1141.24	1143.99	1174.72	1233.26
7	1077.94	1141.59	1144.73	1173.74	1235.36
8	1075.17	1144.03	1146.76	1185.26	1235.60
9	1081.06	1142.21	1145.18	1185.71	1234.90
10	1079.85	1138.06	1145.64	1170.88	1232.51
Average run time (milliseconds)	**1077.49**	**1139.03**	**1146.80**	**1179.47**	**1234.53**

Table 11 : Run times of Fine Sample Sort algorithm for Poisson distribution data (Ref: Figure 9)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	50.70	174.17	298.21	490.21	536.90
2	49.80	176.23	297.74	490.19	542.14
3	50.90	176.94	297.34	493.62	540.76
4	50.35	175.06	294.96	492.38	540.07
5	50.78	175.99	297.06	492.28	539.81
6	50.03	172.68	299.08	490.30	538.60
7	50.15	172.15	296.21	496.58	535.48
8	50.33	175.14	294.97	492.67	541.20
9	49.95	172.15	296.49	495.86	540.06
10	50.95	175.92	293.87	493.89	538.19
Average run time (milliseconds)	**50.39**	**174.64**	**296.59**	**492.80**	**539.32**

Table 12 : Run times of Fine Sample Sort algorithm for Poisson distribution data (Ref: Figure 9)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	589.58	660.91	697.79	836.16	918.35
2	588.45	663.50	691.63	844.52	906.00
3	586.38	667.44	693.97	839.16	902.95
4	588.99	655.35	695.05	849.60	900.06
5	588.59	662.80	696.18	838.71	912.99
6	587.33	661.58	692.70	835.27	908.52
7	586.25	660.89	695.89	830.86	907.08
8	587.94	659.36	702.15	844.00	914.84
9	587.88	658.81	696.26	844.10	917.47
10	592.92	661.12	701.27	840.96	907.28
Average run time (milliseconds)	**588.43**	**661.18**	**696.29**	**840.33**	**909.55**

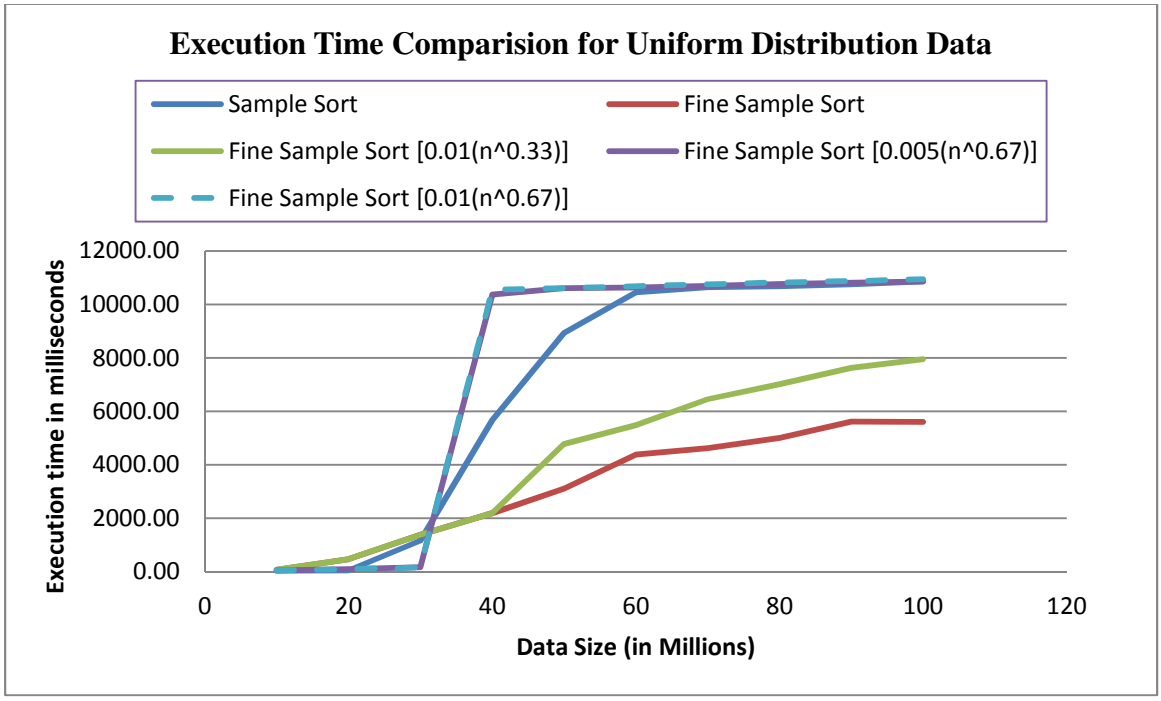

Figure 10: Execution time comparison for uniform distributions input

Table 13: Run times of Sample Sort algorithm for Uniform distribution data (Ref: Figure 10)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	47.21	53.01	1095.49	5727.87	8881.92
2	47.31	52.83	953.45	5711.22	8931.22
3	47.01	52.81	1146.08	5716.37	8881.70
4	46.97	53.29	1246.25	5715.89	8903.40
5	47.37	53.05	1143.63	5710.42	8892.15
6	48.16	52.19	1247.40	5381.63	8886.28
7	47.10	52.09	1143.30	5721.21	8880.73
8	47.33	52.04	1243.99	5712.86	8888.62
9	46.64	52.29	1244.63	5719.62	8890.06
10	46.06	52.18	1244.99	5710.79	9337.15
Average run time (milliseconds)	**47.11**	**52.58**	**1170.92**	**5682.79**	**8937.32**

Table 14: Run times of Sample Sort algorithm for Uniform distribution data (Ref: Figure 10)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	10439.50	10647.30	10707.10	10771.20	10853.90
2	10499.90	10639.50	10552.60	10766.80	10864.00
3	10467.80	10637.90	10707.50	10771.50	10865.40
4	10458.60	10637.90	10708.10	10771.80	10870.10
5	10443.90	10637.90	10716.50	10775.70	10854.10
6	10447.20	10655.90	10711.50	10770.10	10853.20
7	10440.00	10649.50	10557.50	10614.30	10853.50
8	10447.10	10652.20	10714.80	10771.30	10854.80
9	10451.60	10642.20	10718.80	10771.30	10854.80
10	10454.30	10640.10	10707.70	10770.10	10855.30
Average run time (milliseconds)	**10454.99**	**10644.04**	**10680.21**	**10755.41**	**10857.91**

Table 15: Run times of Fine Sample Sort algorithm for Uniform distribution data (Ref: Figure 10)

	Data Size →				
Experiment	**10000000**	**20000000**	**30000000**	**40000000**	**50000000**
1	86.84	554.63	1395.00	2195.01	3106.12
2	77.42	373.47	1184.24	2195.12	3102.84
3	63.16	428.96	1688.39	2194.22	3094.40
4	77.29	459.10	1399.16	2192.70	3097.84
5	88.04	427.99	1509.07	2195.50	3098.26
6	87.26	491.12	1238.46	2195.26	3121.47
7	68.66	402.22	1396.74	2196.82	3106.72
8	63.35	490.65	1454.48	2192.99	3095.72
9	63.29	661.62	1452.44	2212.76	3097.54
10	64.57	458.89	1086.71	2196.71	3121.56
Average run time (milliseconds)	**73.99**	**474.86**	**1380.47**	**2196.71**	**3104.25**

Table 16: Run times of Fine Sample Sort algorithm for Uniform distribution data (Ref: Figure 10)

	Data Size →				
Experiment	**60000000**	**70000000**	**80000000**	**90000000**	**100000000**
1	4376.03	4626.53	5141.50	5750.65	5844.02
2	4382.42	4637.30	4731.39	5421.51	5407.47
3	4377.10	4639.16	4333.24	5647.08	5507.25
4	4387.39	4636.91	4945.46	5868.19	5614.53
5	4386.74	4632.84	5047.17	6217.58	5957.20
6	4383.23	4629.93	5254.70	5097.79	5508.65
7	4395.39	4632.78	4733.57	5421.20	5620.24
8	4377.82	4628.92	5580.11	5987.12	6075.01
9	4377.91	4625.33	5373.82	5535.56	4870.45
10	4414.84	4631.99	4937.96	5315.05	5731.22
Average run time (milliseconds)	**4385.89**	**4632.17**	**5007.89**	**5626.17**	**5613.60**

Table 17: Comparison of % difference in average runtimes over 10 runs of sample sort and Fine Sample Sort algorithms for descending order input

Data Size	Sample Sort	Fine Sample Sort	% Difference
40000000	5584.34	1986.69	64
50000000	9893.66	3024.01	69
60000000	10266.16	3670.23	64
70000000	10454.49	4179.30	60
80000000	10690.66	4820.64	55
90000000	10746.84	5083.54	53
100000000	10780.17	5581.87	48
Average % difference =			**59**

Table 18: Comparison of % difference in average runtimes over 10 runs of sample sort and Fine Sample Sort algorithms for normal distribution input

Data Size	Sample Sort	Fine Sample Sort	% Difference
40000000	5821.72	2196.87	62
50000000	9151.11	2603.67	72
60000000	10120.05	4534.18	55
70000000	10625.68	4807.00	55
80000000	10670.36	4850.64	55
90000000	10736.88	5211.65	51
100000000	10809.11	5466.61	49
Average % Difference =			**57**

Table 19 : Comparison of % difference in average runtimes over 10 runs of sample sort and Fine Sample Sort algorithms for Poisson distribution input

Data Size	Sample Sort	Fine Sort	% Difference
40000000	790.55	492.8	38
50000000	976.91	539.32	45
60000000	1077.49	588.43	45
70000000	1139.03	661.18	42
80000000	1146.8	696.29	39
90000000	1179.47	840.33	29
100000000	1234.53	909.55	26
Average % Difference =			**38**

Table 20 : Comparison of % difference in average runtimes over 10 runs of sample sort and Fine Sample Sort algorithms for uniform distribution input

Data Size	Sample Sort	Fine Sort	% Difference
40000000	5682.79	2196.71	61
50000000	8937.32	3104.25	65
60000000	10454.99	4385.89	58
70000000	10644.04	4632.17	56
80000000	10680.21	5007.89	53
90000000	10755.41	5626.17	48
100000000	10857.91	5613.6	48
Average % Difference =			**56**

Conclusion

In this chapter we have given details on the steps involved in Fine Sample Sort along with the experiment results. Our experimental results indicate that FSS is very competitive in practice. In particular, it is faster than the best known prior GPU sorting algorithm based on comparisons.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

In this thesis, we have explored the features of graphics processing units. Considering the architectural attributes of the GPU, along with the advantages of random sampling techniques, we were able to devise Fine Sample Sort (FSS), an efficient comparison based sorting algorithm, which outperforms the currently best known comparison sorting algorithm, i.e., sample sort by Nikolaj Leischner[2]. Comparison of FSS against sample sort was done for different input distributions. The results show that for uniformly distributed inputs, FSS is 47% faster and on an average 55% faster. For Poisson distributed data. FSS is at least 26% faster and on an average 37% faster. For normally distributed data, FSS is at least 49% faster and on an average 57% faster. For input data that is already sorted in descending order, the speedup of FSS is at least 48% and on an average the speedup is 59%. The above statements hold for data sizes of 40 million and above.

5.2 Future Work

Since GPUs offer the potential of affordable parallelism, there exists a wide scope to utilize the attributes it provides, in order to develop complex and efficient parallel algorithms, which are based on the SIMD model. We feel that our FSS algorithm can be further improved in terms of memory usage by improving the way the histograms of the blocks are calculated. In FSS we calculate the histograms of all the blocks and store these histograms and this requires a large amount of global memory space when we are dealing with large input data sizes. One way to improve this algorithm in terms of memory usage

is to maintain a single histogram table, which has 128 entries at any time. Every block, after calculating its histogram in local memory, can update its values in the histogram table stored in the global memory.

REFERENCES

[1] http://www.engr.uconn.edu/~rajasek/

[2] N. Leischner, V. Osipov and P. Sanders, GPU sample sort, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010

[3] N. Satish, M. Harris and M. Garland, Designing Efficient Sorting Algorithms for Manycore GPUs, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2009.

[4] Bandyopadhyay, S. and Sahni, S., GRS - GPU Radix Sort for Large Multifield Records, International Conference on High Performance Computing (HiPC), 2010

[5] NVIDIA CUDA Programming Guide, NVIDIA Corporation, version 3.0, Feb 2010.

[6] CUDPP: CUDA Data-Parallel Primitives Library, http://www.gpgpu. org/developer/cudpp/, 2009.

[7] Horowitz, E., Sahni, S., and Mehta, D., Fundamentals of data structures in C++, Second Edition, Silicon Press, 2007.

[8] Cederman, D. and Tsigas, P., GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors, ACM Journal of Experimental Algorithmics(JEA), 14, 4, 2009.

[9] http://courses.engr.illinois.edu/ece498/al/Syllabus.html

[10] http://classx.stanford.edu/ClassX/system/users/web/pg/view_subject.php?subject=NVIDIA_ICME_SPRING_2010_2011

[11] http://developer.nvidia.com/

[12] http://en.wikipedia.org/wiki/GeForce_400_Series

[13] J.H. Reif and L.G. Valiant, A Logarithmic Time Sort for Linear Size Networks, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MASS., 1983, pp. 10-16.