EE 194 Project Paper

Xu Liu, Matt Asnes, Behnam Heydarshahi
Tufts University
xu.liu@tufts.edu, behnam@cs.tufts.edu, matthew.asnes@tufts.edu

Abstract

Our final project is comparing performances of CPU and GPU by implementing several sorting algorithms respectively on GPU in sequential way and on GPU in parallel way.

1 Introduction

Since we have hit the power wall and memory wall on the road of improving computers' performance by continually increasing the frequency of processor, we are exploring other ways to keep the performance of computer growing while keep the CPU cool. These ways include making more CPU on single chip, exploiting GPU or AISC's, dynamically adjusting voltage and frequency and so on.
GPU probably will be very helpful for improving integrated performance of the whole computer system. Therefore people usually care about how GPU benefits the overall performance and what kind of programming model helps.

2 Project Description

This project is comparing absolute performances of CPU and GPU on sorting algorithms.

- **Platform** Our programs are developed on the computer in lab 120, which has CPU: intel Core i5 at 3.3 GHz, GPU: Nvidia k620 at 1.124Gh with 2 G GPU memory.
- **Algorithms** We implemented five sequential sorting algorithms : shell, merge, quick, insertion and radix sorting; we also implemented three parallel algorithms: merge, even odd and bitonic sorting. In addition we call parallel radix sorting directly from Nvidia library–thrust as a reference.
- **Tools** We use *nvcc* compile our programs. We use program *time* to measure the consuming time. It will report the total time consuming, time consumed by system and the time consumed by our application. We use another tool *Nvidia Visual Profiler* to profile our program and get details about each module.

3 Experimental Methodology

1. We implement all five sequential sorting algorithms to provide a reference for later comparisons.
2. We write simple cuda program to test the platform and system and obtain detailed system information.

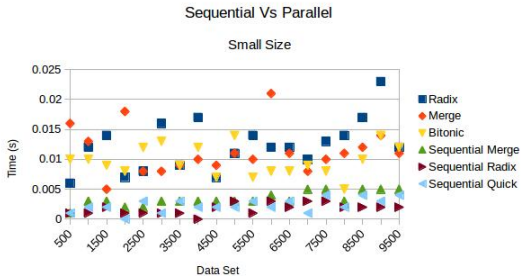

Figure 1: **Comparison between sequential and parallel sorting on small size data set.**

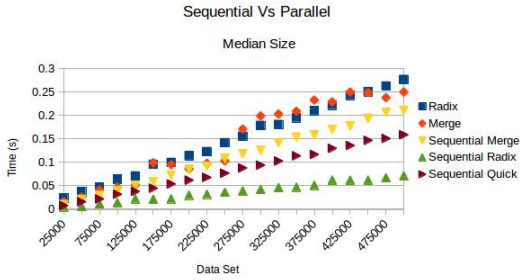

Figure 2: **Comparison between sequential and parallel sorting on median size data set.** *This figure shows, for median size data set (between tens of thousands and million), sequential outperforms parallel.*

3. We develop the three parallel algorithms one each step.
4. We write scripts to measure time consuming of each algorithms on different data set.
5. We plot chart on the test results while comparing the test results.
6. We profile each parallel algorithms to analyze the reason why some are faster than the others.

4 Results and Analysis

Figure 1 shows, for small size data set (less than 10000 elements), there is no obvious trend and also no big difference between different sorting algorithms.
For median and large size data set, sequential sorting outperforms parallel sorting as shown in Figure 2 and 3. This result was not expected by us.

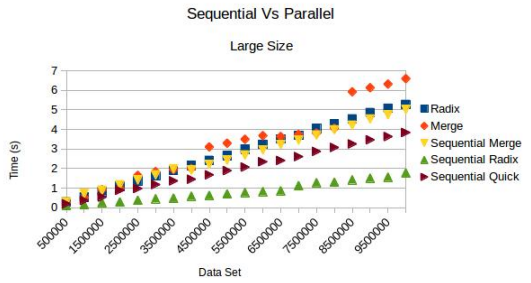

Figure 3: **Comparison between sequential and parallel sorting on large size data set.** *This figure shows, for large size data set (between millions and tens of millions), sequential outperforms parallel.*

	Sequential (seconds)		Parallel(seconds)		
Algorithm	1M	10M	1M	10M	9000
Radix	0.053	0.479	0.555	5.446	
Quick	0.165	1.913	-----	-----	
Merge	0.228	2.446	0.512	7.432	
Even Odd	----	-----	137.46		
Bitonic	----	-----	0.035 (reference)	0.193 (reference)	0.013
Shell	337.49		-----	-----	
Insertion	slow		-----	-----	

Figure 4: **Summary for comparison between sequential algorithms and parallels.**

Figure 4 is the summary of our comparison results. This is the first time we program on GPU and also first time use cuda. We think the reason why the sequential sorting are so fast is that CPU has been optimized well enough by pipeline, branch prediction, cache policy and so on. Well why the parallel sorting are slow? We continue to profile all the parallel to see details.

Figure 5 is the profiling for parallel radix. Parallel radix sort comes from Nvidia thrust library, therefore we are not very familiar with its structure. The profiling shows that it is very compact and exploit the shared memory well. However it still can not outperform sequential radix sort.

Radix is kind of data independent, we can keep multiple threads on data set. It only visit memory one or two times for each element. Actually it possesses a programming model which fit in GPU very well except it may need dynamically allocate memory. data independent. We probably can improve it .

Figure 6 tells us that the last few rounds of parallel merge sort only use few threads and therefore consume most of the time. We think the reason it is slow is that this strategy cannot exploit multiple threads well enough.

In Figure 7, we only cite very small part of the profiling since there are many same module which number is up to the same as

Name	Duration(ns)	GridX	GridY	GridZ	BlockX	BlockY	BlockZ	Registers/Thread	Static Shared Memory
Memcpy HtoD [async]	6182436								
DeviceRadixSortUpsweepKernel	1783983	150	1	1	64	1	1	40	2112
RadixSortScanBinsKernel	6175	1	1	1	512	1	1	38	2624
DeviceRadixSortDownsweepKernel	3337856	150	1	1	64	1	1	80	4688
DeviceRadixSortUpsweepKernel	1704783	150	1	1	64	1	1	40	2112
RadixSortScanBinsKernel	6272	1	1	1	512	1	1	38	2624
DeviceRadixSortDownsweepKernel	3290784	150	1	1	64	1	1	80	4688
DeviceRadixSortUpsweepKernel	1598993	150	1	1	64	1	1	40	2112
RadixSortScanBinsKernel	6112	1	1	1	512	1	1	38	2624
DeviceRadixSortDownsweepKernel	3223776	150	1	1	64	1	1	80	4688
DeviceRadixSortUpsweepKernel	1571409	135	1	1	64	1	1	48	4224
Memcpy DtoD [async]	4160								
Memcpy DtoH [async]	6580384								

Figure 5: **Profiling for parallel radix sort**

Duration	Grid Size	Block Size	Name
6.1212ms	38.147MB	6.0859GB/s	Quadro K620 (0) [CUDA memcpy HtoD]
3.8715ms	(4883 1 1)	(1024 1 1)	mergeInt(int*, int*, int, int)
5.1133ms	(2442 1 1)	(1024 1 1)	mergeInt(int*, int*, int, int)
6.4604ms	(1221 1 1)	(1024 1 1)	mergeInt(int*, int*, int, int)
7.9964ms	(611 1 1)	(1024 1 1)	mergeInt(int*, int*, int, int)
12.207ms	(306 1 1)	(1024 1 1)	mergeInt(int*, int*, int, int)
18.151ms	(153 1 1)	(1024 1 1)	mergeInt(int*, int*, int, int)
	repeat procedure		
45.653ms	(1 1 1)	(153 1 1)	mergeInt(int*, int*, int, int)
92.972ms	(1 1 1)	(77 1 1)	mergeInt(int*, int*, int, int)
174.90ms	(1 1 1)	(39 1 1)	mergeInt(int*, int*, int, int)
295.26ms	(1 1 1)	(20 1 1)	mergeInt(int*, int*, int, int)
490.23ms	(1 1 1)	(10 1 1)	mergeInt(int*, int*, int, int)
789.60ms	(1 1 1)	(5 1 1)	mergeInt(int*, int*, int, int)
1.57609s	(1 1 1)	(3 1 1)	mergeInt(int*, int*, int, int)
2.55756s	(1 1 1)	(2 1 1)	mergeInt(int*, int*, int, int)
2.68451s	(1 1 1)	(1 1 1)	mergeInt(int*, int*, int, int)
6.6332ms	38.147MB	5.6162GB/s	Quadro K620 (0) [CUDA memcpy DtoH]

Figure 6: **Profiling for parallel merge sort.**

the data size. Too many threads cause it slow.

Figure 8 shows the bitonic can exploit as many threads as half of the data size, which make it fast. We think it is the only parallel algorithm with potential to outperform sequential on our platform. Unfortunately we only make it work up to array with size of 9215. Profiling shows that all parallel programs written by us all don't take the advantage of shared memory, which is a direction to improve them.

Figure 9, 10 and 11 show the Even-Odd sorting algorithm are much still as profiling shows it takes to many rounds to guarantee the data sorted. Parallel radix is a little better than parallel merge, since on one hand it possesses better characteristics which fit in GPU, on the other hand it exploit shared memory well.

5 Conclusion

Although Parallelism is a promising direction to take the advantage of Moore's law, the programming model is an significant factor affect whether the multiple threads can be used well. In our project most of sorting algorithm don't fit well the GPU programming model.

Model ideal for GPU is

- Not too small amount of data
- Not too big amount of data

Name	Start Time(ns)	Duration(ns)	GridX	GridY	GridZ	BlockX	BlockY	BlockZ	Registers/Thread	Static Shared Memory
Memcpy HtoD [sync]	148632529	62271								
even_sort(int*, int)	148699344	15712	49	1	1	1024	1	1	8	0
Memcpy DtoH [sync]	1676409706	62336								
odd_sort(int*, int)	148718160	16608	49	1	1	1024	1	1	8	0
even_sort(int*, int)	148737456	15199	49	1	1	1024	1	1	8	0
odd_sort(int*, int)	148755503	15968	49	1	1	1024	1	1	8	0
even_sort(int*, int)	148774127	15360	49	1	1	1024	1	1	8	0
Up to as many as the size of array										

Figure 7: **Profiling for parallel even odd sort.**

Name	Start Time(ns)	Duration(ns)	GridX	GridY	GridZ	BlockX	BlockY	BlockZ	Registers/Thread	Static Shared Memory
Memcpy HtoD [sync]	107087475	5472								
initial(int*, int, int)	107847921	2400	8	1	1	1024	1	1	8	0
create_bitonic(int*, int, int, int)	108207761	6528	8	1	1	1024	1	1	18	0
create_bitonic(int*, int, int, int)	108224241	4704	8	1	1	1024	1	1	18	0
create_bitonic(int*, int, int, int)	108238385	4736	8	1	1	1024	1	1	18	0
Repeat procedure										
bitonic_sort(int*, int, int)	109657326	3552	8	1	1	1024	1	1	13	0
bitonic_sort(int*, int, int)	109668558	3488	8	1	1	1024	1	1	13	0
bitonic_sort(int*, int, int)	109680398	3520	8	1	1	1024	1	1	13	0
Memcpy DtoH [sync]	109701614	7392								

Figure 8: **Profiling for bitonic sort.**

- Independent data
- Intensive calculation with infrequent memory visit
- It is best to have data fit in shared memory

Parallel radix sort and bitonic sort do possess the characteristics of the ideal model. That's why they are relatively faster.

For radix, there is still a drawback which is it may need dynamically allocate memory.

And I think, in our project, the bitonic is the only one parallel algorithm with potential to outperform sequential algorithms. Unfortunately we only make the bitonic sort work up to 9215, I need more time to debug.

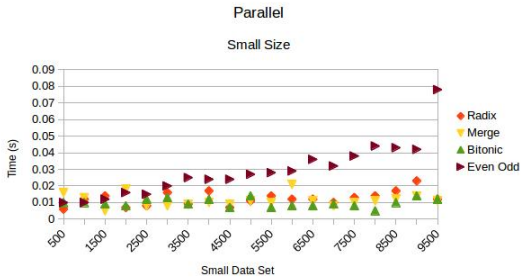

Figure 9: **Comparison between different parallel sorting on small size data set.**

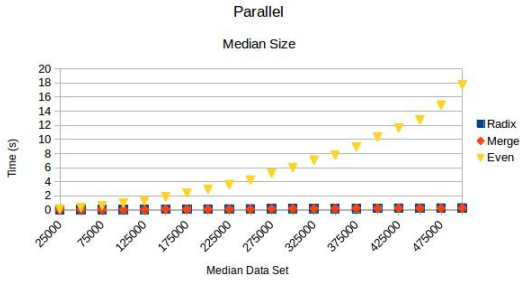

Figure 10: **Comparison between different parallel sortings on median size data set.**

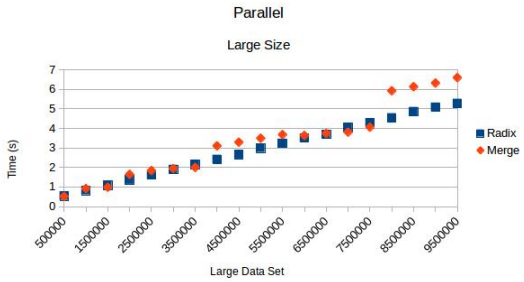

Figure 11: **Comparison between different parallel sorting on large size data set.**