International Journal of
Applied Mathematics,
Electronics and Computers
ISSN: 2147-8228 http://ijamec.atscience.org
Advanced Technology and Science
Original Research Paper

Implementation of Sorting Algorithms with CUDA: An Empirical Study

Ali Yazici *[1], Hakan Gokahmetoglu [2]

Accepted 6th December 2015 **DOI: 10.18100/ijamec.53457**

Abstract: Sorting algorithms have been studied for more than 3 decades now. The aim of this paper is to implement some of the sorting algorithms using the CUDA language in a GPU environment provided by the Nvidia graphics cards. This empirical study is done for comparing the performance of the sorting algorithms in a run-time environment provided by the GPUs and the CUDA programming language. This study considers the implementation of bubble sort, insertion sort, quicksort, selection sort and shell sort algorithms. It is shown in this article that there is a significant amount of speed-up in using CUDA and the Nvidia architecture instead of a sequential code running on standard architectures.

Keywords: *CUDA, sorting algorithms, GPGPU programming, parallel sorting.*

1. Introduction

CUDA programming language [1] is first introduced in 2008 by Nvidia, to run parallel computations on Nvidia devices, then Graphical Processing Units (GPUs), later accelerator devices from Nvidia only added to CUDA programming. Although, Message-Passing Interface (MPI) [2] library is used for massively parallel and cluster systems, it does not deliver any performance to a single PC. Hence, in this paper, it is decided to conduct an empirical study to illustrate how the graphics cards provided by Nvidia and the use of CUDA language can improve the computation times for sorting algorithms.

Theoretically, a SM 2.0 CUDA capable GPU can execute 1024 threads at a compute cycle [3]. That is equivalent of 1024 CPU cores, when doing a data-level computation in parallel, e.g. simple arithmetic operations.

Finally, wall clock values are compared using the increasing output sizes for each sorting algorithm. By doing so, we believe, one can compare the theoretical execution time values to actual results, and decide if CUDA accelerates the execution times for the algorithms designed sequential effectiveness in mind.

Next section gives an overview of sorting algorithms for classical computer architectures. Section 3 surveys some of the parallel sorting algorithms relevant to this study. In Section 4 implementation of the aforementioned sorting algorithms using the CUDA programming language and the GPUs are discussed. In Section 5 test results are given in a comparative manner. Section 6 and 7 are devoted to discussions and conclusions.

2. Sequential Sorting Algorithms

2.1. An overview

Sorting algorithms [4] are generally used for ordering elements in an array. The most conventional way is using alphanumerical ordering, and then the resulting sorted array can be used for merging and searching algorithms. Another use of sorting is increasing the human readability of an output.

Five sequential sorting algorithms with different computational complexities are considered, namely, bubble sort, selection sort, insertion sort, quicksort and shell sort.

Bubble and quicksort are types of exchange sorts where elements in the array are compared pairwise and interchanging the elements only if necessary. Selection sort is a type of selection sorts where an extreme value (e.g. a maximum or a minimum) is chosen and then sorting is done once according to that item. After that many selection operations are made for finalizing the sorting the array. Insertion and Shell sorts are types of insertion sorts where final sorted array (or list) is build one by one, i.e. moving each element of the array in to desirable position until the list is completely sorted in the desired order.

The CUDA implementations of algorithms such as merge sort, distribution sorts or hybrid sort will be considered in a separate study.

2.2. Computational Complexities

Computational complexity [5] is a theory which is about relating the computational steps to execute a code that does some arithmetic operations, with the measuring of these steps done according to the time and memory space taken on a particular machine. Then big-O notation can be used to associate the complexities to time values and compare among the different algorithms.

Generally speaking, sorting algorithms have computational complexities between O(n) and $O(n^2)$. Table 1 below summarizes the computational complexities of the sorting algorithm in question using the big-O notation.

Table 1. Theoretical complexity values for sequential algorithm

Sorting Algorithm	Best case	Avg. case	Worst case	Parallel prediction – p=#of treads
Bubble	O(n)	$O(n^2)$	$O(n^2)$	$O(n^2/p)$
Insertion	O(n)	$O(n^2)$	$O(n^2)$	$O(n^2/p)$
Quick	O(nlogn)	O(nlogn)	$O(n^2)$	O(nlogn/p)
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2/p)$
Shell	O(n)	$O(n^{3/2})$	$O(n^{3/2})$	$O(n^{3/2}/p)$

In Table 2.2, a best case means when there is a minimum effort needed for sorting an array. This is usually when the array is perfectly shuffled and array does not consist of increasing or decreasing series of adjacent numbers. A worst case is opposite of this. An average is in between these two.

[1] *Software Engineering, Atilim University Ankara, Turkey*
* *Corresponding Author: Email: ali.yazici@atilim.edu.tr*

Last column is just a prediction on the effort needed when sorting the array using the parallel version of the code written in CUDA language.

3. Parallel Sorting Algorithms

3.1. Related work

Parallel sorting has been considered by many researchers in the context of different parallel architectures. For example, general organization of some of the basic sorting algorithms for multithreading is considered in [6]. A parallel bucket-sort algorithm is presented in [7] that requires time O(log n) and the use of n processors. A pipelined insertion sort for sorting n numbers with n processes using MPI is given in [8]. In the same article, an inherently parallel sorting method, bitonic sort is discussed which can be effectively implemented in shared memory architectures.

For the GPUs, efficient strategies for parallel radix sorting on GPUs were discussed in [9]. In a technical report by NVIDIA Corporation [10], radix sort and merge sort algorithms are implemented in multicore GPUs using the CUDA language. It is claimed that the merge sort is the fastest published comparison-based GPU sort and is also competitive with multi-core routines.

This study is an attempt to implement the five well-known sorting algorithms mentioned above in multicore GPUs and try to identify the challenges and barriers in using GPUs. The paper is not to provide the fastest implementation but to lay out the basics of GPU processing in terms of some of the sorting algorithms.

3.2. GPU Architecture

A CUDA program consists of phases which one or more of these phases are run on the CPU or GPU [11]. In CUDA, data parallel part of the code called a kernel. A kernel may consist of device functions and their structures that run on the GPU. All CUDA code is compiled with the CUDA compiler and kernel code typically generate a large number of threads (e.g. 10,000) to compute. That can be achieved because GPU threads are considered very light weight compared to CPU threads, when clock cycles to generate and schedule these threads are the subject which, GPUs have special hardware to support that [11].

When a program is executed, kernel code execution is deferred to GPU where large numbers of threads are generated to execute the code in parallel.

Fig. 3.2 [11] shows the CUDA thread model for consecutive runs of CUDA kernel and serial code; on the GPU and on the CPU respectively. The concept of CUDA programming model is to generate thousands of threads that perform in a Single Program Multiple Data (SPMD) manner, each on a small chunk of data in parallel [11]. In the figure the waiving single arrow displays the main thread that is common for all programs and many of those arrows together in the Grid0 box displays; the GPU code is handled with many threads at a time unlike the single main thread the serial code has.

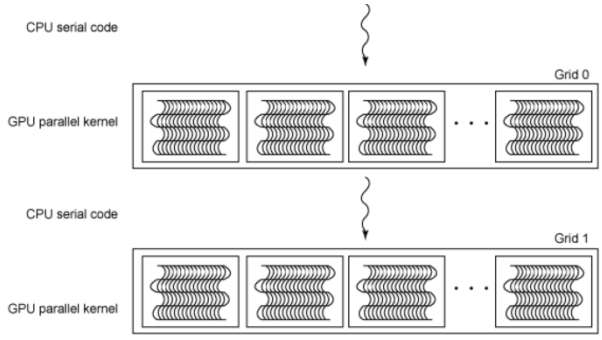


Figure 1 CUDA thread model [11]

3.3. Usage of PyCUDA

In this study, an open source Python library called PyCUDA [12] is used.

1) CUDA language: CUDA makes available the general purpose computation (GPGPU) via using GPUs. An actual task of these processors is making graphics processing but they can also be accessed on Nvidia devices with CUDA kernels for GPGPU. In this paper, CUDA 6.5 version is utilized for computational purposes.

2) Python: Python has many community made library extensions. This study incorporates pyCUDA, matodlib, and numpy libraries.

4. Implementation Details

This section details some of the issues concerning the implementation of the sorting algorithms mentioned above.

First of all, multi-core GPU architecture is very well suited for data-parallel computations. Then, all of the sorting methods considered in this study provide data level parallelism. In other words, array to be sorted is practically decomposed into subarrays and using multithreading approach each subarray(s) is sorted by the GPUs and the results are reduced and gathered to the main thread.

Algorithm 1 and 2 show the pseudo codes for the Bubble sort for a standard architecture and Nvidia CUDA kernel code respectively. This code is in the most general form for the sorting algorithms used in this study.

Algorithm 1 Pseudo code for sequential Bubble sort

```
1: FOR passnum BETWEEN LengthOf(array) AND 0
2:   FOR i BETWEEN 0 AND passnum
3:     IF (array[ ith_element] > array [i+1_th_element])
4:       SWAP(array[ ith_element] WITH array [i+1_th_element])
5:     ENDIF
6:   ENDFOR
7: ENDFOR
```

Algorithm 2 Pseudo code for parallel Bubble sort

```
1: idx = thread_id, N = length_of(array)-1
2: FOR i = idx BETWEEN 0 AND N
3:   FOR j BETWEEN 0 AND N-1-i
4:     IF (array[ j ] > array [ j+1 ])
5:       SWAP(array[ j ] WITH array [ j+1 ])
6:     ENDIF
7:   ENDFOR
8: ENDFOR
```

In the first line of Algorithm 2, variable idx is associated with thread_id, which means that for each run of the kernel, the thread number will be unique. This line is necessary for the correct computation as well as the parallelization of the code, otherwise code will run sequentially. Then, variable N equals to the length of input array. The second line starts with for loop each thread_id associated with the loop variable i, then an inner loop iterates the compare and swap array indexes for each pass of the outer FOR loop. The fourth line compares the elements of the array pairwise, and if any pair happens to be in descending order, then is swapped, in order to place the elements in ascending order.

Notice here that the CUDA code is quite similar to the sequential one. This is necessary because llvm compiler is a C language-based compiler and the default optimization parameter value is –O2. If one tries to alter the code, as such changing the "for loops" with "if" clauses, then all algorithms will be nearly the same algorithm, causing the llvm compiler produces nearly the same machine code. That means, the algorithm of the code is altered in such a way that it does not reflect the actual behaviour.

4.1. Sequential Results

An environment using the latest Ubuntu OS is set up, and all the programs are present in the built-in repositories.

For a test bed, "numpy" was used to generate arrays increasing in size, all holding floating point numbers. The sequential version of algorithms executed using CPU with these arrays generating run times, as shown in Fig. 3 below.

The average run time for all algorithms, omitting the times with 512 elements, is 0.00015 seconds, that is 150μs (microseconds). The measured runtimes are remarkably small and as expected, quicksort seems to the winner among all especially with large size arrays.

The logarithmic scale is used, so bars looking downwards are showing values between 1 second and 1μs, and they are non-negative. The bars above horizontal line are between 1 second and 100 seconds.

In addition, bubble sort, insertion sort and selection sort has run times all exceeding 100 seconds. This is expected, because computational complexities for these algorithms are $O(n^2)$ on the average.

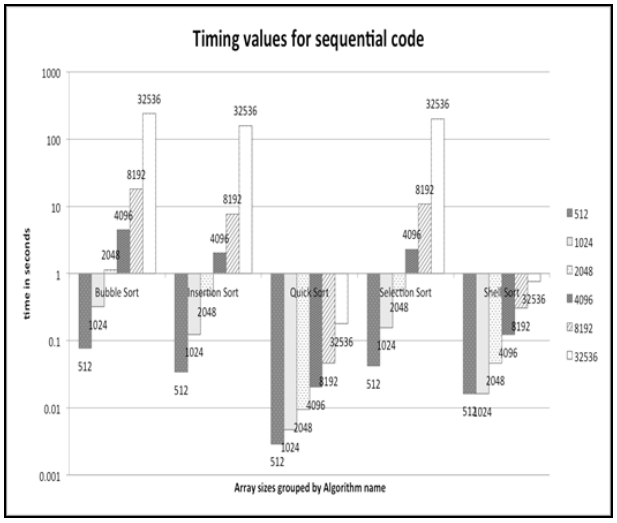
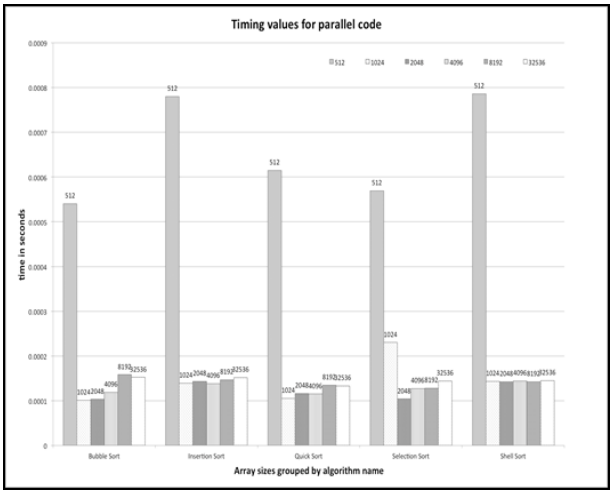


Figure 2 Execution times for sequential code



5. Comparison of the timing values

The timing values for sequential algorithms are in Fig. 3 and parallel counterfeits are in Fig. 2. From these charts it is very clear that CUDA language does what it promises and it gives an advantage in terms of execution times when compared to the sequential counterfeits. Even when considering the quicksort algorithm, where all sequential code execution times are under 1 seconds, the speed-up achieved with its parallel version is 333x times the sequential code.

The results clearly show the CUDA can achieve great speed-ups over sequential code. However this paper was intended for academic work, and we use Python language, which is a JIT (just in time) interpreted language. If we also remember that the sequential code fails to execute with 32,536 elements, there might be some problem with the Python compiler used, although pyCUDA code always compiles with CUDA "llvm", and is not affected by the Python compiler. So we leave this to readers' choice to seek after if Python code can run a little faster.

Table 2. Sequential results vs. parallel timing predictions and speed-ups in seconds

Arr_size	Bubble sort	t_p	S_{up} Bubble	Quick sort	t_p	S_{up} Quick
512	0.08	$0.8x10^{-4}$	142	0.003	$0.3x10^{-5}$	5
1,024	0.3	$0.3x10^{-3}$	314	0.005	$0.5x10^{-5}$	45
2,048	1.1	$0.1x10^{-2}$	1100	0.01	$0.1x10^{-4}$	81
4,096	4.5	$0.4x10^{-2}$	3753	0.02	$0.2x10^{-4}$	178
8,192	18	$0.2x10^{-1}$	7561	0.05	$0.5x10^{-4}$	341
32,536	240	$0.3x10^{0}$	11420	0.2	$0.2x10^{-3}$	1342

Table 2 above shows the timing values in seconds for sequential bubble sort and quick sort algorithms. The column with heading tp, represents the predicted parallel timing values according to the formula; tp=ts/p, where ts is the time for the sequential execution and p is the total number of treads utilized. In Table 2 above, thread count was kept constant at 1,024 threads, for all array sizes.

By looking at this table it is not unexpected that the quicksort and shell sort are the only two algorithms that can execute over 30,000 elements, in competitive times, because the theory implies that these algorithms are more efficient to execute when running with a single thread or process in terms of time and computational space (i.e. memory). The parallel versions, does not seem to show any sign of change in behaviour, when

executed with different element sizes. This may be due to architecture of GPU and CUDA, which involves expensive memory operations in terms of time. Hence, to obtain maximum efficiency from CUDA, the array sizes must be a very large value, e.g. millions. Then CUDA timings of the parallel algorithms would be according to the complexities that of serial timings.

If one compares the tp, values column to parallel results graph (i.e. Fig 2) than it is clear that tp values are almost identical for small array sizes and 10x slower when compared to large arrays. This is quite easy to explain, the difference is due two different devices that are used to find the results, the CPU and GPU. And, there is currently no way in CUDA for running a kernel code on GPU in sequential. The 10x extra speed-up is achieved when we compare the theoretical tp values to actual parallel code timings. This can be explained as GPU(s) having nearly or over 20x speed-ups compared to CPU(s), especially when large data sizes are considered.

Moreover, Sup-Bubble and Sup-Quick columns represent the speed-ups achieved for timing values of serial execution versus parallel executions for bubble sort and quick sort, respectively. Where, Sup = tserial/tparallel, where tserial is execution time for serial execution, and tparallel is execution time for parallel code.

6. Discussions

Python language is used on purpose because Python's numpy library is very powerful at array operations. Furthermore Python has many more extensions that are community maintained, making it superior to coding in plain C/C++ where the coder has to generate every piece of code from scratch.

Results found show that there is almost no difference in running times for parallel versions, but this is mainly because parallel languages are all designed for very large array sizes. This performance can be hardly achieved with the sequential code.

Also, Intel 2nd generation i7 4-core CPU and Nvidia GT540m GPU is used for this experiment, where both has certain limitations. A different test with more powerful hardware can achieve larger array values, to reflect the computational complexity for timing and computational space much better.

In other words, this was the limitation for the hardware used. And, with a GPU with more memory space one expects reduction in the execution times of the algorithms, especially with larger array sizes, which is also depicted by the corresponding theoretical complexity bounds given earlier.

7. Conclusions

In this paper, the results prove there is significant amount of speed-up can be achieved when using CUDA instead of a sequential code. As the results indicate, the sequential sorting algorithms organized for GPUs, can provide high performance results without even substantially changing the sequential codes. However, to get the most out of the GPU processing power, one has to redesign the algorithms to match with the underlying parallel architecture of the graphics cards (GPUs).

Acknowledgements

This study is supported by Atılım University and founded by the Atılım Universtiy scholarship program for graduates. This study is presented in ICAT'15 conference.

References

[1] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of Gpu Computing), 1st. ed., Morgan Kaufmann, 2012

[2] P. Pacheco, Introduction to Parallel Programming, Morgan Kaufmann, 2012

[3] N. Wildt, The CUDA Handbook, A Comprehensive Guide to GPU Programming, Pearson Education, 2013

[4] J. Edosomwan, Sorting Algorithm, LAP Lambert Academic Publishing, 2012

[5] S. Arora and B. Barak, Computational Complexity: A Modern Approach, 1st. ed., Cambridge University Press, 2009

[6] M. Dawra and P. Dawra, IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 4, No 3, July 2012

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

[8] B. Wilkinson and M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd. ed., Pearson Education, 2005.

[9] D. Merrill and A. Grimshaw, Revisiting Sorting for GPGPU Stream Architectures, Technical Report CS2010-03, Department of Computer Science, University of Virginia. February 2010.

[10] N. Satish, M. Harris and M. Garland, Designing Efficient Sorting Algorithms for Manycore GPUs, NVIDIA Technical Report NVR-2008-001, Sep. 2008., NVIDIA Corporation.

[11] D. B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-On Approach (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010

[12] (2014) http://mathema.tician.de/software/pyCUDA/