Contrasting Parallelized with Sequential Sorting

Anurag Dutta[1] and Manjari Saha[*2]

[1] Undergraduate, Department of Computer Science and Engineering, Government College of Engineering and Textile Technology, Serampore
[2] Assistant Professor, Department of Computer Science and Engineering, Government College of Engineering and Textile Technology, Serampore

*Corresponding Author

cmanjari@gmail.com

Abstract. Sorting is one of the important Data Management Techniques that is quite commonly operated on Data Structures. It has been proved that best, any sorting algorithm can do, in asymptotic versions is $\varphi \log_2 \varphi$, when the iterations are made to run in sequential manner. But what if we parallelize them? Architectural advancements, like Amdahl's Law have made use of the concept of Parallelism, and have successfully revolutionized the computing power of the microprocessors. In this paper, we will try to push the limit of computing, by making use of Graphical Processing unit and will briefly compare the efficiency of the algorithms acquainted with both the techniques, sequential as well and parallel in terms of running time. In the first section, we have included some of the preexisting works on this topic, and have mentioned about scopes of further improvement. From the next section onwards, we have slowly put together all the pillars required to support our work.

Keywords: Parallel Computing, Sorting Algorithms, GPU, Microprocessors

1 Introduction

Sorting Algorithms are the set of algorithms that arranges some data elements in monotonically increasing or decreasing order of its magnitude. If, we try to get a wide, aloof definition of this, we can start by mentioning, that it's an algorithm that searches for a possible permutation, where elements of each index are monotonically arranged, from the set of all Permutations $\mathcal{P}(A)$. It is guaranteed that there always exists an instance $\mathcal{P}_i \in \mathcal{P}(A)$, such that in $\mathcal{P}_i(A)$, $A_i \geq A_{i+1}$ or $A_i \leq$

$A_{i+1} \forall i \in n$, where n is the cardinality of the data set. So far, in nearly half a century of advances in computer science, the fastest algorithm developed so far, manages to scoop out the necessary Permutation in logarithmic time, amortized with linear time multiplicatively. Though there are many such algorithms which has been successful in hitting the bottom of the jar, to name a few, Merge Sort, Quick Sort, Heap Sort are some of the great instances, but none have crossed the minima mark in negative sense. Infact, it can be proved mathematically,

Let us try to understand sorting in terms of searching. Each element that are being sorted is eventually searching its accurate place in the set. So, let we are interested in evacuating the element α_i at its original place. Now, to get to its original place, we will have to search at n places in maximum. Now, once it is dropped at its original place, we will move to the next element of α_i, and to evacuate it to its actual place, we will have to search at most $n-1$ places and in this the number of places to be searched varies like $n-3$, $n-4$, $n-5$, $n-6$, …, 1. Now there are a lot of searching algorithms like linear search, binary search, interpolation search[1], but as we know, the fastest searching algorithm devised so far is binary search which searches at logarithmic time.
So, total time

$$\mathcal{T}(n) = log_2 n + log_2(n-1) + log_2(n-2) + \cdots + log_2 1$$

$$\mathcal{T}(n) = \sum_{i=0}^{n-1} log_2(n-i) = log_2\left(\prod_{i=0}^{n-1}(n-i)\right) = log_2(n!)$$

If we consider the Stirling's Approximation[2][3],

$$\mathcal{T}(n)|_{min} = log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\left(\frac{1}{12n+1}\right)}\right)$$

$$\Rightarrow \mathcal{T}(n)|_{min} = log_2\left(\sqrt{2\pi} n^{\left(n+\frac{1}{2}\right)}\right) + log_2\left(e^{\left(\frac{1}{12n+1}\right)-n}\right)$$

$$\Rightarrow \mathcal{T}(n)|_{min} \approx \left(n+\frac{1}{2}\right) log_2(n) + n \approx n\, log_2(n)$$

But this scenario is true iff no architectural changes are incorporated in the working of the algorithms. But, if we change the course of action, of the computing unit, or Infact change the unit of computation, we will surely fetch good stuff out of it.

2 Graphical Processing Unit (GPU)

In the previous section, we have mentioned that, advances haven't been possible from the last 5 decades on the sorting algorithms, and have logically supported it. Like, we can consider it as intractable but, since we have got no logical proof in support of that, we have proved that the converse is not true. But, towards the end, we have mentioned that, with architectural advances, we could probably pass the minima. Now, in the general sorting, or the sequential sorting, we are forced to run the n iterations of searching, which we have mentioned in the introduction, in sequel. But, if by any means we could make use of the parallelism to perform the iterations in batches, we could get the work, infact to be precise, the sorting action done in a much smaller time. Well, fortunately, we have got computational devices that could do the same, and those devices or hardware are nothing by the Graphical Processing Unit. Like, they can perform multiple tasks or better, multiple iterations of the same task simultaneously. It's like some non-ambidextrous person in work, being replaced by one ambidextrous.
In this section, let's have a brief review on how this ambidexterity – analogue hardware can fasten our work. GPUs are processing unit similar in action to the Central Processing Unit. The major difference between this two hardware is the computational power based on core count. In general, the core count of GPUs, $\varkappa_{GPU} > \varkappa_{CPU}$, the core count of CPUs. Let's for instance, take the example of two mid specs hardware,

- `Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz   1.50 GHz`
- `NVIDIA® GeForce® MX330 GPU`

The core count (total) of the CPU is 4, with a boost of 2.[4] At max, the CPU can handle 8 threads[1] at simul. Whereas, the core count (total) of GPU is 384[5] (unified). At max, it can handle such a huge number of threads. It's very obvious from the facts and figures that, GPUs are a more powerful, yet expensive computational device. GPUs have been quite popular in the last few decades due to the need of fast processing, high graphics computing, like the video games, we see around us, these days, demands a lot of hardware supported computing.

While, CPUs are a set of singular cores, GPUs are a set of multiple cores. GPUs consist of smaller components called as Stream Multiprocessors (SM). Each SM consists of many Stream Processors (SP) on which actual computation is done.

[1] It is a single instance of computing.

Some useful terminologies relating GPUs[6] are

- **Thread:** It is a single instance of computing. One or more threads are processed on a Stream Processor.
- **Block:** It is a group of Threads. One or more blocks are processed on a Stream Multiprocessor.
- **Grid:** It is a group of Blocks. One Grid is generated for one Kernel and on one GPU. Also, only one kernel can be executed at one time instance.
- **Warp:** Number of threads in a block running simultaneously on a Stream Multiprocessor is called a Warp

Apart from these, GPUs invoke, with them, a wide variety of different types of memory. Let's have a look on them.

- **Local Memory:** Each Stream Processors make use of the Local Memory. All variables declared inside a `kernel` are stored in this Memory
- **Shared Memory:** This is the memory which is common to all the threads present inside each block. This Memory helps in reducing the latency – memory access time. When we have to use shared memory for a variable, it should be prefixed with keyword `__shared__` during its declaration.
- **Global Memory:** Main memory of the GPU. To make use of this memory, `cudaMalloc()` function is taken into account.
- **Constant Memory:** It is generally used to store constants – that won't change their value during execution. When we have to use constant memory for a variable, it should be prefixed with keyword `__constant__` during its declaration.
- **Texture Memory:** This memory is again used to reduce latencies. It's domain of action is quite unique, and quite interesting to study. It takes in account of the notion of Spatial Locality – wherein one value is pinched, while the surrounding is also procured because these surrounding values have higher chances of being rereferred.

Now, again, there must be some medium that would commensurate the medium between the task and the computational unit. One such is the parallel computing toolkit, namely CUDA – Compute Unified Device Architecture, developed by

Nvidia. According the CUDA Developers, CUDA is designed to work with programming languages such as C, C++, and Fortran. For, our work, we will make use of the same.

3 CUDA Parallelism

In this section, we will offer a brief o'erview of the CUDA Toolkit. Since, the toolkit, is made open – sourced by it's developers, we will replicate some of the contents directly, from the official site itself, though they will be encapsulated with apt. referencing.

According, to NVIDIA, [7][8]

> CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.
>
> In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers' program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.
>
> The CUDA Toolkit from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.
>
> The first GPUs were designed as graphics accelerators, becoming more programmable over the 90s, culminating in NVIDIA's first GPU in 1999. Researchers and scientists rapidly began to apply the excellent floating-point performance of this GPU for general purpose computing. In 2003, a team of researchers led by Ian Buck unveiled Brook, the first widely adopted programming model to extend C with data-parallel constructs. Ian Buck later joined NVIDIA and led the launch of CUDA in 2006, the world's first solution for general-computing on GPUs.
>
> Since its inception, the CUDA ecosystem has grown rapidly to include software development tools, services and partner-based solutions. The CUDA Toolkit includes libraries, debugging and optimization tools, a compiler and a runtime library to deploy your application. You'll also find code samples, programming guides, user manuals, API references and other documentation to help you get started.

> CUDA accelerates applications across a wide range of domains from image processing, to deep learning, numerical analytics and computational science.

All these statements conclude to the fact, that CUDA Toolkit improves the performance of computational attributes by utilizing the power of the GPUs.

4 Sorting Algorithms

As, we had mentioned in the introduction itself, that Merge Sort is the best that sorting algorithms could do, taking the advent of sequentially accredited sorting.

Merge sort is a very good sorting technique which follows the technique of divide and conquer[9]. Let's have a look on its course of action.

Let we have been given a set of n elements in a list (data structure with language independency), such that

$$L(n) = \{\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{n-1}\}$$

Under this algorithm the list is divided into equally sized sub parts and merged step by step in a recursive[10] manner to bring it to sorted format. It is often referred to as the best sorting technique when we are required to sort a linked list.

The Pseudocode for this algorithm will be

```
function merge(list_of_elements, low_index, mid_index, high_index)

    size_vault1 ← mid_index - low_index + 1

    size_vault2 ← high_index - (mid_index + 1) + 1

    vault1[size_vault1]

    vault2[size_vault2]

    for i = 0 to i = size_vault1 - 1

        vault1[i] = list_of_elements[low_index + i]
```

```
end for

for i = 0 to i = size_vault2 - 1

    vault2[i] = list_of_elements[mid_index + 1 + i];

end for

i, j ← 0, 0

k ← low_index

while i < size_vault1 and j < size_vault2

    if vault1[i] > vault2[j]

        list_of_elements[k] = vault2[j]

        increment j, k

    else

        list_of_elements[k] = vault1[i];

        increment i, k

    end if

end while

while j < size_vault2

    list_of_elements[k] = vault2[j];

    increment j, k

end while

while i < size_vault1

    list_of_elements[k] = vault1[i];

    increment i, k

end while
```

```
end

function merge_sort(list_of_elements, low_index, high_index)

    if low_index < high_index

        mid_index ← low_index + (high_index - low_index) / 2

        merge_sort(list_of_elements, low_index, mid_index)

        merge_sort(list_of_elements, mid_index + 1, high_index)

        merge (list_of_elements, low_index, mid_index, high_in-
dex)

    end if

end
```

The Akra – Bazzi Method[11] is one of the techniques to analyze the asymptotic behavior of recurrences, which are of the form

$$T(x) = g(x) + \sum_{i=1}^{k} \alpha_i T\big(\beta_i x + h_i(x)\big)\, \forall x \geq x_0$$

The asymptotic behavior of $T(x)$ is found by determining the value of ρ for which

$$\sum_{i=1}^{k} \alpha_i (\beta_i)^{\rho} = 1$$

and plugging in that value of ρ in the recurrence format equation, we will get

$$T(x) \in \Theta\left(x^{\rho}\left(1 + \int_1^x \frac{g(u)}{u^{\rho+1}}\, du\right)\right)$$

For, Merge Sort, the recurrence is

$$T(n) = \begin{cases} 0 & \forall n = 1 \\ T\left\lfloor \frac{n}{2} \right\rfloor + T\left\lceil \frac{n}{2} \right\rceil + n - 1 & \forall n > 0 \end{cases}$$

and the computed closed form is like $\Theta(n \log n)$.

Well, this merge sort is one of the advanced sorting algorithms, but there exist certain naïve, not so good algorithms that take exponentially high time. One such is the Permutation Sort. In permutation sort, we simply generate all permutations of the elements present in the list, and on a later note, we check the monotonicity of each permutation, if the tendency of monotonicity is constant, we declare that permutation as the sorted list.

The Pseudocode for this algorithm will be

```
function permutation_sort(list_of_elements)

    current_list ← generate next_permutation(list_of_elements)

    if current_list: monotonic

       return current_list

    else

       return permutation_sort(current_list)

    end if

end
```

Now, generating all permutations takes execution time of the order

$$\varphi\left(\prod_{i=0}^{n-1}(n-i)\right)$$

where, $\varphi(\ )$ is the appropriate asymptotic notation.
Here, in this paper, we will allow the Merge Sort, to make use of the computational power of the CPU Microprocessor, while, we will make use of the computational power of the GPU for the Permutation Sort.

5 Contrast between the Sorting Algorithms in Action

As mentioned in the section above, we will implement the Merge Sort Algorithm by making use of the Computational Power of the CPU, while we will be making use of the GPU Power for the Permutation Sort, and we will be performing these

two by using the NVIDIA CUDA Toolkit. The contrast will be laid on the basis of both – worst and best – case scenario respectively.

5.1 Best Case Scenario

For both, Merge and Permutation Sort, we will have the best case, or attain the best – case scenario, iff the elements of the list $L(n) = \langle \alpha_i \rangle$, that is intended to be sorted is already arranged in increasing order of their magnitude, i.e., $\alpha_i \leq \alpha_j \forall j > i$. Compiling the results obtained from the binary approaches, we have noted down the aspects of both the techniques, and the cardinality of the dataset is varied according to the law $|L_i| = 10 \times |L_{i-1}|$

$\lvert L_i \rvert$	T_{CPU}	T_{GPU}
1	0.000000	0.000000
10	0.000002	0.000018
100	0.000009	0.000018
1000	0.000086	0.000013
10000	0.000995	0.000014
100000	0.011538	0.000014
1000000	0.133270	0.000018

NOTE: T_{CPU} and T_{GPU} are the real space time (literal meaning) execution time, measured on the basis of machine clock time in seconds, these are not based of some kind of arithmetic falsifiability or trafficability

5.2 Worst Case Scenario

For both, Merge and Permutation Sort, we will have the worst case, or attain the worst – case scenario, iff the elements of the list $L(n) = \langle \alpha_i \rangle$, that is intended to be sorted is arranged in decreasing order of their magnitude, i.e., $\alpha_i \geq \alpha_j \forall j > i$. Compiling the results obtained from the binary approaches, we have noted down the aspects of both the techniques, and the cardinality of the dataset is varied according to the law $|L_i| = 10 \times |L_{i-1}|$

$\lvert L_i \rvert$	T_{CPU}	T_{GPU}
1	0.000000	0.000001
10	0.000002	0.000014
100	0.000010	0.000013
1000	0.000122	0.000013

10000	0.011004	0.000014
100000	1.222412	0.000014
1000000	9.199570	0.000018

NOTE: T_{CPU} and T_{GPU} are the real space time (literal meaning) execution time, measured on the basis of machine clock time in seconds, these are not based of some kind of arithmetic falsifiability or trafficability

To view the CUDA enhanced code, from which, execution time was obtained and filled in the tables, shown above, click here.

6 Conclusion

In this section, we will draw a conclusion to the observations that we have discussed in this research. It is evident from the table, that $T_{CPU} \gg T_{GPU} \forall |L_i| \geq 100$, otherwise, its $T_{CPU} \leq T_{GPU} \forall |L_i| \leq 100$, The main reason behind this cumbersome result is quite simple, GPUs are designed in such a way that they fill up the different types of their memory before computational advancements, while, CPUs tend to work more on the computational sector at first.

Thus, from this work, we could conclude that, the fastest sequential sorting, - Merge Sort is slower in action for larger cardinality of the data set, it's working on, when contrasted with the most naïve sorting technique, Permutation Sorting. This Also, justifies the fact that, though GPUs are expensive, still, they have effective solution finding speed, which is in optimal requirements in the modern world.

References

1. Gonnet G., Rogers L. and George J. (1980), "An algorithmic and complexity analysis of interpolation search", Acta Informatica, Springer, 13, pp 39 – 52.
2. Wang L. and Zou C., "Accuracy analysis and applications of the Sterling interpolation method for nonlinear function error propagation", Measurement, Elsevier, 2019, pp 55-64.
3. Manzoni K., "Modeling credit spreads: An application to the sterling Eurobond market", International Review of Financial Analysis, Elsevier, 2002, pp 183-218.
4. Intel® Core™ i7-1065G7 Processor | Intel
5. GeForce MX330 Dedicated Graphics | NVIDIA
6. Ashu Rege, "An Introduction to Modern GPU Architecture", © NVIDIA Corporation 2008

7. "Nvidia CUDA Home Page". 18 July 2017.
8. Abi Chahla, Fedy (June 18, 2008). "Nvidia's CUDA: The End of the CPU?". Tom's Hardware. Retrieved May 17, 2015.
9. Jon Kleinberg, Éva Tardos, “Algorithm Design”, Pearson, 2005
10. Herbert Wilf, “Algorithms and Complexity”, Tailor and Francis, 2002.
11. Akra, M., Bazzi, L. On the Solution of Linear Recurrence Equations. Computational Optimization and Applications 10, 195–210 (1998).