Fall 8-15-2016

GPU-ArraySort: A parallel, in-place algorithm for sorting large number of arrays

Muaaz Awan
WMU, muaazgul.awan@wmich.edu

Fahad Saeed
Western Michigan University, fahad.saeed@wmich.edu

GPU-ArraySort: A parallel, in-place algorithm for sorting large number of arrays

Muaaz Gul Awan[1] and Fahad Saeed [1,2]
[1]*Department of Electrical and Computer Engineering,*
[2]*Department of Computer Science, Western Michigan University, MI, USA.*

Abstract

Modern day analytics deals with big datasets from diverse fields. For many application the data is in the form of an array which consists of large number of smaller arrays. Existing techniques focus on sorting a single large array and cannot be used for sorting large number of smaller arrays in an efficient manner. Currently no such algorithm is available which can sort such large number of arrays utilizing the massively parallel architecture of GPU devices. In this paper we present a highly scalable parallel algorithm, called GPU-ArraySort, for sorting large number of arrays using a GPU. Our algorithm performs in-place operations and makes minimum use of any temporary run-time memory. Our results indicate that we can sort up to 2 million arrays having 1000 elements each, within few seconds. We compare our results with the unorthodox tagged array sorting technique based on NVIDIA′s Thrust library. GPU-ArraySort out-performs the tagged array sorting technique by sorting three times more data in a much smaller time. The developed tool and strategy will be made available at https://github.com/pcdslab/

GPU; Big Data; Sorting; Mass Spectrometry; Proteomics

1 Introduction

Sorting is one the most researched topics in computational science since it is integral part of many algorithmic solutions [1]. Sorted data also brings an order to otherwise perplexing nature of data which makes it useful for many big data applications [2]. Sorting has been widely used in big data analytics problems in Geography [3], Geology [4], Combinatorics [1], Computational Biology [5] [6], Astrophysics [7] and Particle Physics [8]. Recently with the advent of massively parallel devices such as Graphic Processing Units (GPU) many sorting algorithms have come out utilizing their versatile architecture and CUDA programming model. With each new algorithm a better design for exploiting the two level parallelism is introduced. This has led to some very fast and effective sorting techniques for very large number of elements. However, most of these sorting techniques have been limited to 1-dimensional array of very large size. In

practice, however, many domain sciences face the problem of sorting large number of short arrays such as in Proteomics [5], Genomics [9], Geology [4], Environmental Engineering [10] and Characterization of other Biomolecules [11]. For example mass spectrometry datasets may consist of very large number of spectra where each spectra is a set of intensities corresponding to mass-to-charge ratios. Majority of the algorithms dealing with such dataset require these spectra to be sorted either with respect to intensities or mass to charge ratios [12][13][5]. In literature no such technique can be found which can sort such huge number of moderately sized arrays using modern high performance computing devices such as GPUs. Surveying around one can encounter many unconventional/unpublished techniques of sorting such huge number of arrays. One such technique is by using multiple runs of a stable key based sorting algorithm from CUDAs Thrust library [14]. We have discussed this technique in section VII.

In this paper we present a parallel algorithm for sorting large number of arrays on a GPU. Our technique exploits the inherent coarse-grained [15] nature of the data and further divides each array into finely grained chunks of data to be sorted independently. This brings in a near uniform distribution of data chunks for better load balancing across threads thus making our algorithm highly scalable. GPU-ArraySort apart from being manifolds faster uses about three times less memory because of its in-place nature.

2 Related Work

In literature there are a large number of GPU sorting algorithms available which can efficiently sort *one* large array [6]. However, none of them can be directly applied for sorting large number of arrays. Here we cover major GPU based sorting algorithms which have appeared in recent times. Different algorithms use different approaches for breaking down large number of elements in smaller bins such that they can be sorted by individual threads. The process of breaking down can be tedious as best results can be achieved when load across the threads is uniform. The two common ways of breaking down data into smaller bins include the creation of independent m-bins and m-way merge approach[6]. In the latter case sorted buckets have to be merged together once individual processing elements have sorted them while in case of independent bins there is no need of any merging since it is a simple case of concatenating all the bins.

In [16] a Hybrid sorting approach for GPUs has been introduced, the authors subdivide a large array of elements into smaller lists using bucket sort approach. All buckets are then sorted in parallel using merge sort technique. The authors claim to have best utilized the highly parallel architecture of a GPU in an efficient way to outperform the GPU radix sort [17]. In [18] authors have presented a highly optimized versions of GPU based radix and merge sort algorithms. Authors claimed their radix sort algorithm to be the fastest known sorting algorithm in literature at that time while their merge sort was fastest comparison based sorting technique. They worked very closely in exploiting the fine grained parallelism of GPU along

with maximum exploitation of shared memory. Their design utilizes an efficient GPU based algorithm for calculating prefix-sum. A GPU based version of sample sort has been presented in [6], this is the first GPU based study of sample sort technique. Their design consist of more involved techniques such as use of predication to avoid branch divergence thus exploiting the fine-grained parallelism. This along with introduction of a binary search tree structure for traversal of splitter elements makes this algorithm a highly optimized GPU based sorting technique.

Also in 2009 a GPU accelerated version of quick sort was introduced [19], the design of GPU quick sort also follows the conventional breakdown of a large list into smaller lists such that each small list takes a size equivalent to that of shared memory of GPU. This technique enables the algorithm to take the most out of the fastest memory on GPU. The design also ensures that consecutive threads are always accessing adjacent memory locations to ensure coalesced global memory accesses. Authors claim that their design was the fastest sorting algorithm available at that time.

Recently an improvement in Odd-Even Sorting algorithm [20] has been introduced, this algorithm focuses on improving the efficiency of GPU based Odd-Even Sorting algorithm especially making it more feasible for CUDA programming model. Their results show considerable improvement over existing versions of the algorithm.

All of the algorithms discussed above are dedicated sorting algorithms for a 1-dimensional list containing large number of elements. In order to sort several thousand smaller arrays using existing algorithms each array would have to be sorted one after the other thus making the process sequential in nature. The 1-dimensional sorting algorithms which offer the option of performing a stable sort on the elements with respect to an array of keys, can be employed to sort multiple arrays, using a make-shift methodology. NVIDIAs Thrust library offers one such option of sorting a given array with respect to an array of keys in a stable manner. Sorting large number of arrays using this methodology has been discussed in section VII. We call this technique; Sorting using Tagged Approach (STA). However this technique is very inefficient, it performs a lot of redundant functions and uses about three times more memory than is actually required. We present an algorithm dedicated for sorting large number of arrays in parallel, utilizing full potential of a GPU. Our algorithm is capable of sorting much larger number of arrays in a much shorter time as compared to the STA approach.

3 Graphic Processing Units and CUDA programing model

Graphic processing units were first introduced as dedicated graphics computing unit [21]. Capable of performing transforms and lighting with hardware accelerated support. This revolutionized the gaming and graphics industry. Highly parallel architecture of a GPU provided a lot more resources for compute intense problems, especially those related to graphics generation.

A Graphic Processing Unit consists of several Streaming Multiprocessors (SMs). Initially

each SM contained about 8 CUDA cores but the recent devices with compute capability 3.0 and later house staggering 192 CUDA cores on each SM. NVIDIAs K-40 tesla device contains a total of 2880 CUDA cores [22].

A GPU comes with a *global memory* which can be of the order of GBs and is commonly used for storing the data to be processed, mostly this is the memory to which all the data is transferred to and from the host before any processing begins. A much faster on-chip memory also known as *shared memory* is available per multiprocessor. In early devices shared memory varied from 16K to 32K and in devices having compute capability 2.0 and later it has been pushed up to 48K bytes. Shared memory has a latency about 100x less than the global memory which makes it a default choice for caching small amount of information needed frequently for processing [22].

NVIDIAs CUDA platform provides very flexible programing model to write codes that can be easily run on the GPUs. This model utilizes SIMT (Single Instruction Multiple Thread) Architecture which comprises of two levels of parallelism. The programming model constitutes of a grid of larger number of blocks while each block handles several hundred threads. For execution threads are scheduled onto a Multiprocessor in warps of 32 threads. Threads belonging to the same block are executed concurrently on a same multiprocessor. Threads in a warp are executed in a lock-step i.e. same instruction is executed across all the threads in a warp. In order to get the maximum out of this programming model, one needs to take care of several features [6] [19] of this architecture as discussed below:

3.1 Optimized Memory Accesses

Once a warp has been scheduled on a multiprocessor, the device coalesces the global memory accesses for all the threads in a warp. So it becomes essential that the data required by all the threads in a warp resides in a same locality otherwise device has to perform several transactions which makes the process inefficient owing to large bandwidth of global memory. It needs to be considered here that threads with consecutive thread id are scheduled together.

3.2 Minimizing Branch Divergence

As discussed above, all the threads in a warp execute in lock-step thus any divergence of paths among threads of a warp results in a performance penalty. Our algorithm has been designed in a way such that it avoids *all* the branch divergences.

3.3 Exploiting the shared memory

The shared memory is about 100x faster than the global memory. GPU-ArraySort considers this advantage and brings the chunks of data in shared memory for processing. Also all the frequently used data structures are also kept in the block shared memory.

4 Problem Statement and Proposed Algorithm

Our approach has been designed keeping in view the Mass Spectrometric data. At most in a proteomics dataset each spectrum can have up to 4000 peaks including the back ground noise and peaks due to impurities [23]. This number of peaks can easily fit in the shared memory of any device of compute capability 2.0 or later.

Most of the existing sorting algorithms recursively divide data into smaller chunks such that it fits into the size of shared memory and then apply some common sorting algorithm. In the case of an array of arrays each array is inherently small enough to fit in shared memory. We further divide each array into still smaller chunks making each chunk of data independent of others. This process is done using sample sort mechanism [6]. Each chunk is then independently sorted using insertion sort technique running on individual thread. Later all the sorted chunks are concatenated to obtain a sorted array. Now we formulate the problem statement:

Definition 1: let there be a set I of N unsorted arrays $I = \{A_1, A_2, A_3, \ldots, A_N\}$ where each array is of size n and $Ai = \{a_1, a_2, a_3, \ldots, a_n\}$. I' is a set of sorted arrays $I' = \{A'_1, A'_2, A'_3, \ldots, A'_N\}$ such that each A'_i is of the form; $A'i = \{a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n\}$.

4.1 Algorithm

In literature sample sort has been used as a default choice of sorting algorithm when using a parallel computing system [24]. Sample sort helps dividing large data set into smaller parts which removes the data dependency and thus a lot of room for parallelism is created. Advantage of sample sort over m-way merge sort is that there is no need of putting in extra effort for a merge stage. The data we are dealing with already contains a lot of room for parallelism. We utilize this property by distributing arrays across multiple thread blocks such that each block is assigned one array. Then we further subdivide each array into smaller data independent buckets, to utilize the thread level parallelism.

The GPU-ArraySort has been designed to minimize the on device memory usage so that large number of arrays can be dealt in one go. In order to keep the usage of memory to a minimum we apply in-place processing at each step.

The algorithm has been divided into three different phases, each phase runs in a separate kernel launch. In first phase each block operates on the assigned array and gathers the required number of splitters using regular sampling approach. In second phase each array is subdivided into independent buckets based upon splitters from first phase. The buckets and bucket sizes are sent forward to the third and final phase where in-place insertion sort takes place on each bucket thus giving out sorted arrays. Fig. 1 shows an overview of the algorithm. Following section contains detailed working of each phase.

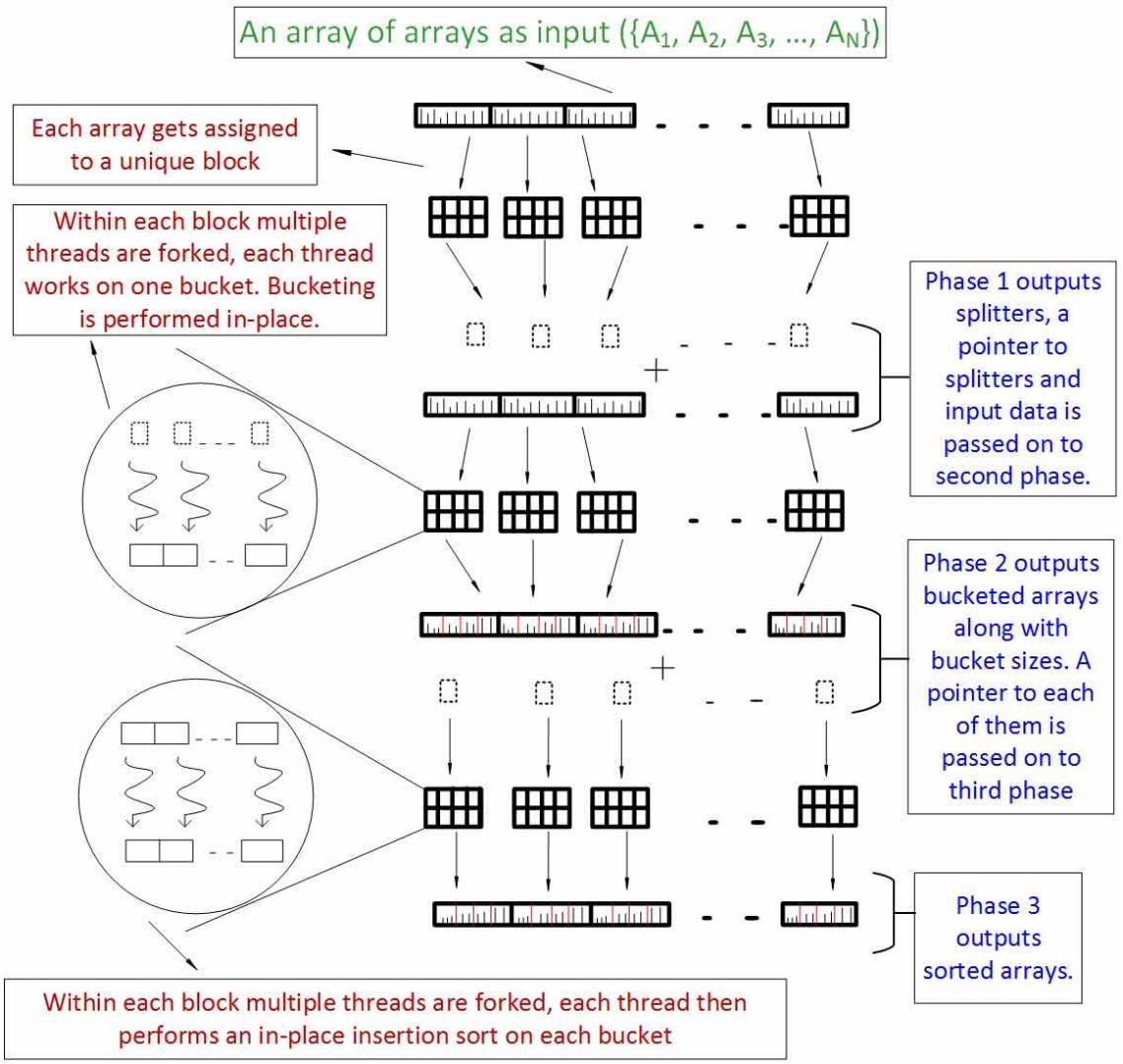

Figure 1: An overview of the GPU-ArraySort algorithm. The first phase accepts N arrays and distributes them across blocks. In the second and third phase multiple threads operate on each array. The buckets boundaries can be observed in red, in second and third phase.

5 Implementation

5.1 Phase 1 (Splitter Selection)

The arrays are usually small enough to fit within the shared memory of the GPU, so first each block moves its assigned array into its shared memory. The number of splitters required depends upon the number of buckets each array is divided into. As this step decides the sizes of smaller buckets which will be sorted by individual threads in the last phase, hence it becomes very critical that we optimize the size of these buckets, for maximum efficiency. Our empirical study showed that the best performance is obtained when there are at least 20 elements per bucket. This choice of size of bucket is totally independent of size of individual array as well as total number of arrays.

Definition 2: If n is the size of an array, let B_i be the set of buckets for array i, $B_i = \{b_1, b_2, b_3, \ldots b_p\}$ where $p = \lfloor \frac{n}{20} \rfloor$.

For p buckets we need to have $p-1$ splitters, these splitters are obtained from a sample set obtained from the unsorted array A_i using regular sampling method. Our studies showed that for uniformly distributed data 10% regular sampling gave most evenly balanced buckets and hence the best running time. The samples obtained are first sorted using in-place insertion sort. Then the $p-1$ splitters are chosen by traversing the sorted sample-array while gathering splitters at regular intervals. As each block returns its set of splitters, they are written to global memory at indices calculated using each blocks id such that consecutive blocks write in consecutive memory locations. Per block, single thread is used for performing all these operations, we tried using more complex strategies but owing to the small size of sampled array, over heads were too large. Also as sampled array is very small in size and can be conveniently placed inside the shared memory.

The array of splitters thus formed can be defined as:

Definition 3: let S be the array of size N, each element $s_i \in S$ is an array of size q which consists of splitters for array A_i. $S = \{s_1, s_2, s_3, \ldots, s_N\}$ where $s_i = \{sp_1, sp_2, sp_3 \ldots, sp_q\}$ and $q = p - 1$. Algorithm 1 describes a per thread pseudo code for first phase.

Algorithm 1 Per thread pseudo code for splitter selection

Data: An array A_i and required number of splitters q

Result: An array of splitters s_i for array A_i

samples = obtainSamples(A_i)

sortedSamples = insertionSort(samples)

index = 0

sampleIndex = 0

stride = calculateStride(sortedSamples)

while *sizeOf* (s_i) *not equal to* q **do**

    s_i[index] = A_i[sampleIndex]

    sampleIndex+ = stride

    index + +

end

5.2 Phase 2 (Bucketing)

This phase constructs the buckets based upon the splitter values from previous phase and builds a global array which keeps record of sizes of all the buckets of all the arrays.

*Definition 4:*Let Z be the array of size N, each element $z_i \in Z$ is an array of size q which consists of bucket sizes for array A_i. $Z = \{z_1, z_2, z_3, \ldots, z_N\}$ where $z_i = \{zb_1, zb_2, zb_3 \ldots, zb_p\}$, here each $zb_j \in z_i$ represents size of bucket j in array A_i.

Now again each array gets assigned to a unique block having threads equal to the number of buckets p. The sub-array sp_i is moved to shared memory because of its very small size yet high frequency of use. The pointers to sp_i are determined on the fly using each block′s and thread′s id.

Each thread in the block is assigned a pair of splitters from sub-array sp_i depending upon its thread id, such that each thread gets a unique pair of splitters.

*Definition 5:*Let r_i denote a splitter pair for a thread i then $r_i = \{sp_i[tid], sp_i[tid+1]\}$ here tid denotes each thread′s id.

The advantage of assigning a splitter pair to each thread is that it helps avoiding branch divergence by completely removing any other paths from the code, this can be observed in Algorithm 2. Assigning a pair of splitters can result in overlapping buckets which can upset the normal sample sort mechanism. To avoid overlapping we introduce two additional splitters in sub-array sp_i, by adding a splitter smaller than the smallest value in array A_i at the starting index while a value larger than the largest value of A_i at the last index.

Now each thread traverses the array A_i in parallel and buckets the element lying within the range of its splitter pair while keeping track of a counter $zb_j \in z_i$, where j is the bucket under consideration and i is the array being treated. At the end of bucketing process, each

such counter contains the size of corresponding bucket. We also explored the option of using multiple threads on single bucket but that slows down the process considerably, most possibly because of the additional overhead.

Once the buckets have been created, each bucket is written back to the actual memory location of array A_i. In conventional approach this write back process had to be sequential but using the calculated bucket sizes we were able to parallelize this write back process as well. The tedious process of writing back to the same memory location comes with an advantage of saving about 50% of device′s global memory.

Algorithm 2 describes a per thread pseudo code for second phase.

Algorithm 2 Per thread pseudo code for bucketing phase

Data: An array A_i and a pair of splitters r_i
Result: A bucket of elements within splitter pair range
splitterPair = obtainSplitters(r_i)
initializeBucket(bucket)
index = 0
bucketIndex = 0
while *not the end of array* A_i **do**
    if *splitterPair[1]* $< A_i$*[index]* $<$ *splitterPair[2]* **then**
        bucket[bucketIndex] = A_i[index]
        bucketIndex++
    end
    index++
end

5.3 Phase 3 (Sorting)

In the final phase the main focus is sorting of buckets. Each bucket can be sorted using any conventional sorting algorithm but insertion sort has proven to the fastest known sorting algorithms for very small number of elements [25]. As each bucket in our case is of size at least 20 so we chose insertion sort for this purpose, besides an added advantage of insertion sort is in-place sorting. Hence this phase does not use any additional memory other than that occupied by the bucketed arrays.

The kernel launches with a copy of pointer to the buckets array and a pointer to the array Z containing sizes of all the buckets. Each bucketed array A_i is assigned to a unique block having threads equal to the number of buckets p. Within each block each thread is assigned a unique bucket. The pointers(starting and ending) to each bucket are calculated based on the thread ids and the size of each bucket.

The in-place insertion sort algorithm takes the pointers to each bucket as input and output

is a sorted bucket. As all the independent buckets belonging to a same array are placed in contiguous memory locations, hence the action of in-place insertion sort leaves behind a completely sorted array. Using the sample sort mechanism we were able to save the additional time which might have been required for a merge phase.

Algorithm 3 describes a per thread pseudo code for third phase.

Algorithm 3 Per thread pseudo code for sorting phase.

Data: An array A_i and pointers to a bucket

Result: A sorted bucket of elements

bucket = retrieveBucket(A_i)

sortedBucket = insertionSort(bucket)

6 Time Complexity

Each array gets assigned to an individual block and in theory each block is processed in parallel [21], with this assumption, time complexity analysis becomes independent of number of Arrays. Following is a phase by phase analysis of the algorithm:

- For the first phase time complexity is simply a sum of time taken to sort the samples and time for picking out splitters i.e. $O(q + (r * n * log(r * n)))$. Here r is the sampling rate. Over here $O(q)$ is seemingly a dominant factor as the value of q depends upon the value of n hence with increasing n $O(q)$ will become dominant.
- In the second phase dominating factor is the time taken for bucketing of elements, which takes one traversal of array by each thread giving us time complexity of $O(\frac{n}{p})$. This factor acts as the most dominant factor among the three phases.
- The third phase is sorting of p buckets using insertion sort, so total time complexity can be written as $O(\frac{n}{p} * log(\frac{n}{p}))$.

After simplifying and omitting the non-dominating factors the resultant equation can be written as:

$$O(((n+q) + \frac{p*r+1}{p} * n * log(n)) * \frac{N}{N}) \tag{1}$$

$$O((n+q) + \frac{p*r+1}{p} * n * log(n)) \tag{2}$$

In Eq. 1 the whole value has been multiplied and divided by $\frac{N}{N}$ to show how assignment of each array to a different block nullifies the largest factor. N can be huge in size but here it is cancelled out, leaving n as the largest value.

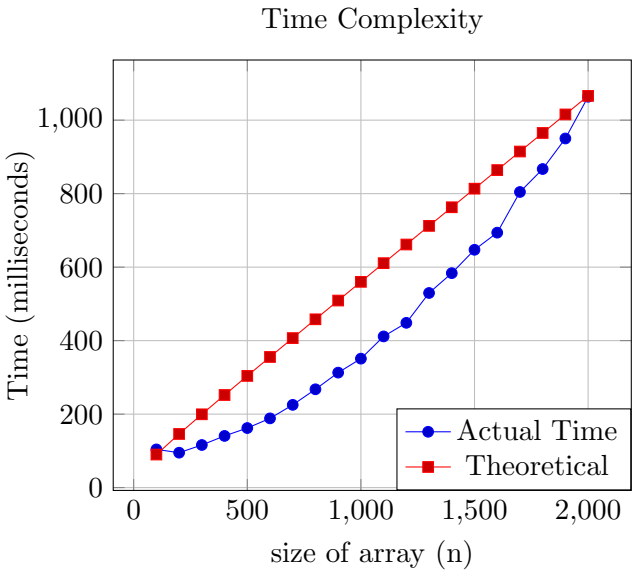

Figure 2: The figure shows plots between theoretically calculated values for each value of n and the practically obtained value. The plot for actual values follows the same trend as that of theoretically calculated values.

From Eq. 2 we can conclude that two dominating factors will be $O(n)$ from phase two and the sorting time $O(\frac{n}{p} * log(n))$. Eq. 2 can be further simplified as:

$$O(\frac{n}{p} + \frac{n}{p} * log(n)) \tag{3}$$

The time complexity of algorithm effectively depends upon the number of elements in each array. In order to verify the theoretical time complexity of the GPU-ArraySort we plotted the processing time with varying value of n i.e. size of each array and compared this plot with the theoretically calculated values. Fig. 2 a plot of actual values versus the theoretically calculated values. Here while varying the value of n we keep N constant at 50000.

7 Performance Evaluation

We evaluated the performance of the proposed algorithm in two different phases; first we did a runtime comparison with a known technique of sorting large number of arrays. The we perform an analysis to see the maximum number of arrays each technique can sort on the given GPU.

7.1 Existing Array sorting methodology

As discussed earlier in literature there is no dedicated GPU based sorting algorithm which can be used to sort large number of arrays. However NVIDIA′s Thrust library offers stable sort with key approach which can be used for sorting large number of arrays after tagging them with keys. This unorthodox technique of sorting arrays using Thrust library is explained below:

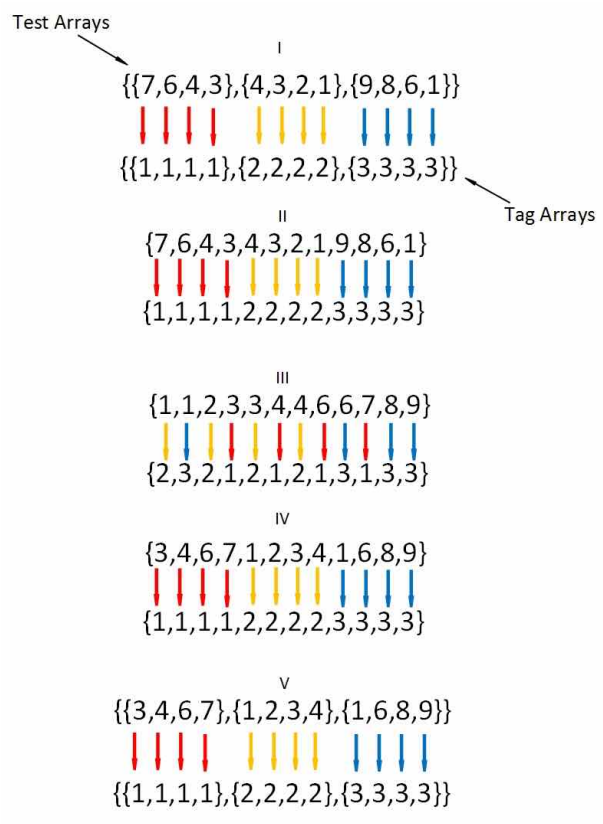

Figure 3: A step by step process explaining the STA technique, here the arrays to be sorted are referred as test arrays: I) A tag array is created for each array to be sorted. II) Arrays are merged into one big array. III) Arrays are sorted using the array of tags as keys. IV) Again arrays are sorted using the test arrays as key. V) Arrays are restored based upon their tags.

7.1.1 Sorting using Tagged Approach (STA)

Let $I = \{A_1, A_2, A_3, \ldots, A_i\}$ be a list of arrays to be sorted where $i = N$, then in order to use the STA approach we create another list of arrays and call it the array of tags.

Definition 6: Let $T = \{T_1, T_2, T_3, \ldots, T_i\}$ be list of arrays of tags such that $i = N$ and $|T_i| = |A_i|$. Here each element $t \in T_i$ represents a tag for array T_i and carries the same value i.e. $t = i$.

Once the tags have been created all the arrays of I are merged into one single array and all the tags are merged into another array. Then the sorting proceeds in two steps :

- Perform a stable sort on the array, containing the arrays to be sorted, using the array of tags, as keys.
- Perform a stable sort on the array of tags, using the array of arrays to be sorted, as keys.

The process has been explained in the Fig. 3.

It can be observed that sorting arrays like this is not only tedious but can be very time consuming. The whole process requires a lot of redundant work. Process of adding tags and

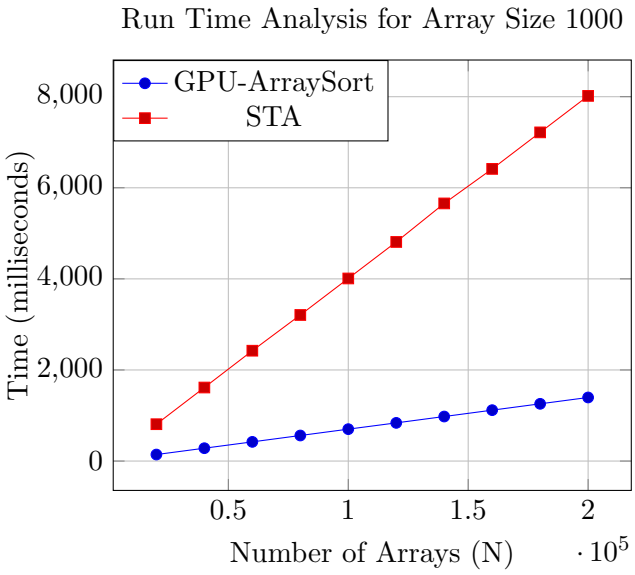

Figure 4: The figure shows time versus number of arrays plots for GPU-ArraySort and the tagged sorting approach using key based stable sorting algorithm from Thrust library.

then sorting them forms the brunt of time required for sorting multiple arrays using STA. In order to sort the tags, the tag array has to be stored in GPUs global memory thus utilizing twice the memory than actual data. Moreover Thrust librarys function which performs the required stable sort with respect to keys utilizes radix sort as its core sorting algorithm. Radix sort uses almost $O(N)$ more space than the data under process [26]. Thus we can conclude that theoretically STA uses about 3 times more memory than may actually be required to sort all the arrays.

7.2 Runtime Analysis

First we perform an analysis to test the running time of GPU-ArraySort and the STA technique discussed above. To perform these experiments we created four different datasets, each dataset consisted of about 200000 arrays while the sizes of arrays were 1000, 2000, 3000 and 4000 respectively for four datasets. Each array was randomly generated using a uniform distribution between 0 and $2^{31} - 1$. All the experiments discussed from here onwards were performed in identical environment. We made use of a server with 24 CPU cores each operating at 1200 MHz. The Graphic Processing Unit used was NVIDIAs Tesla K-40c, it consists of 15 Multiprocessors while each Multiprocessor consisted of 192 CUDA cores making a total number of CUDA cores equal to 2880. Total global memory available on the device was 11520 MBytes and the shared memory of 48 KBytes was available per block. Furthermore all the experiments were performed using float as the data type used.

Figs. 4 through 7 show runtime comparison between STA and GPU-ArraySort. Its clear from the figures that GPU-ArraySort out performs the STA technique for all the array sizes.

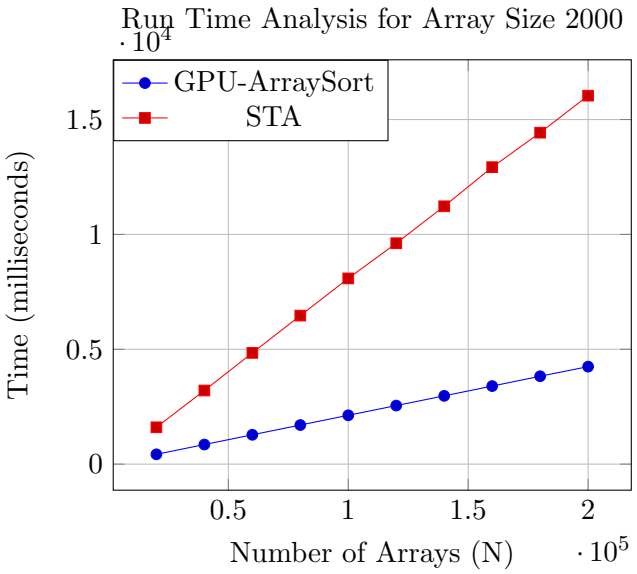

Figure 5: The figure shows time versus number of arrays plots for GPU-ArraySort and the tagged sorting approach using Thrust stable sort.

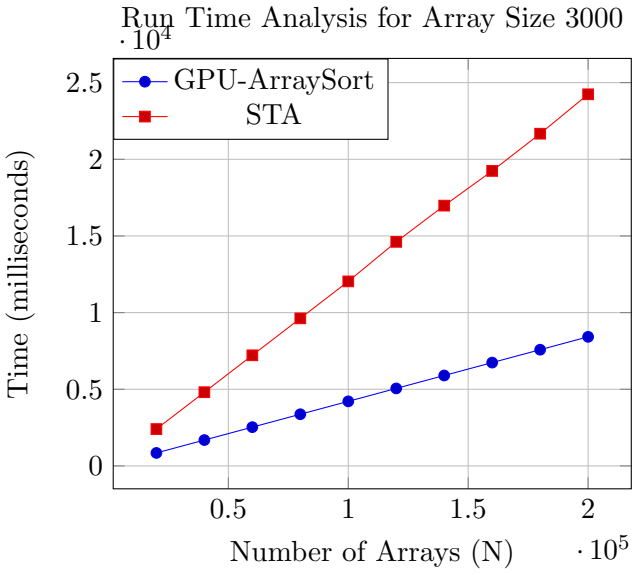

Figure 6: The figure shows time versus number of arrays plots for GPU-ArraySort and the tagged sorting approach using Thrust stable sort.

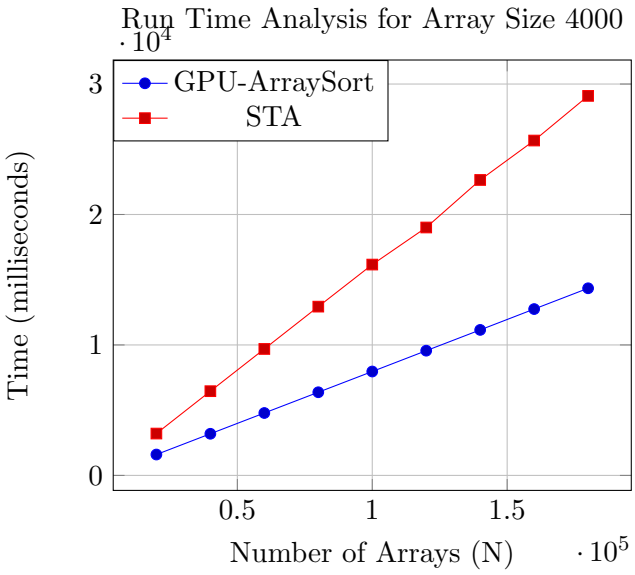

Figure 7: The figure shows time versus number of arrays plots for GPU-ArraySort and the tagged sorting approach using Thrust stable sort.

In the next section we demonstrate memory efficiency of GPU-ArraySort in comparison with the STA Technique.

7.3 Data handling efficiency

In order to test the data handling capacity of each technique we performed the same experiments as those of previous section but without any bound on the number of arrays.

Table 1 shows the total number of Arrays processed by each technique corresponding to size of array in the dataset. It can be observed that the GPU-ArraySort algorithm can sort about three times more arrays than STA technique in a much more efficient manner.

8 Conclusion

The modern scientific equipmenti is capable of generating GBs of data per second. Most of the data requires to be in a certain order for it to make any sense. Most of the Big Data analytics algorithm expect the input data to be in a sorted order or perform this as their initial step. Dealing with huge datasets involving very large number of lists can be very cumbersome even for modern machines. This calls for high performance solutions using modern GPU devices as coprocessors along with the host CPU.

In this paper we have presented a high performance, in-place, GPU based array sorting algorithm as a solution for a popular problem. This algorithm can be included as an integral part of many existing software to act as a vital GPU boost. Our algorithm is capable of sorting huge number of arrays without any redundancy and minimized storage of temporary data. Our

Array Size	GPU-ArraySort	STA
1000	2000000	700000
2000	1050000	350000
3000	700000	200000
4000	500000	150000

Table 1: The table shows number of arrays sorted by STA technique and GPU-ArraySort. The center column shows that GPU-ArraySort can sort upto 2 million arrays of size 1000 while in comparison STA technique was able to sort only 0.7 million arrays. This comparison is for Tesla K-40c GPU.

design out performs the existing solutions by quite a margin both in speed and data handling efficiency. Our experiments have shown considerable speed up over the STA technique which makes use of NVIDIAs Thrust library. Also our algorithm has demonstrated efficient use of GPU global memory by sorting about three times more data.

9 Future Work

This algorithm is a part of an on-going research which involves development of high performance GPU based algorithms for proteomics. We intend to extend this algorithm into an out-of-core GPU based array sort algorithm which will be able to sort huge datasets involving moderate sized arrays without any concern of GPU global memory. The global memory of GPU highly limits the performance of all the GPU based algorithms, because once the memory limit is reached, the host has to wait for GPU to finish processing and then transfer more data to be processed. This results in very large communication delays. A very effective solution for this problem would involve a carefully designed algorithm which hides data transfer latencies in runtime and gives a high throughput out-of-core array sort algorithm. Our design will involve the use of multiple sampling techniques in accordance with the distribution of the dataset under consideration.

Many proteomics algorithms make this assumption that incoming data will be in a sorted form however some pre-processing algorithms tend to perform functions which renders this data out of sequence thus making sorting a vital part of such algorithms. Also in literature there are algorithms which sort the input data in a certain sequence to make it better suitable for their processing, providing a GPU accelerated solution for this would improve the performance of these algorithms by considerable amount. Furthermore the out-of-core array sort algorithm will be integrated in several proteomics related softwares to boost their performance and make

them more scalable for ever increasing size of biological data.

10 Acknowledgments

This work was performed with support from the National Science Foundation (NSF) under the CRII award CCF-1464268. The authors would like to acknowledge hardware donation grant from NVIDIA, of a K-40 Tesla GPU which was used in the work described in this paper.

References

[1] R. Sedgewick and K. Wayne, *Algorithms*, 2011.

[2] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[3] A. Saalfeld, "Sorting spatial data for sampling and other geographic applications," *GeoInformatica*, vol. 2, no. 1, pp. 37–57, 1998.

[4] H. Longerich, G. Jenner, B. Fryer, and S. Jackson, "Inductively coupled plasma-mass spectrometric analysis of geological samples: a critical evaluation based on case studies," *Chemical Geology*, vol. 83, no. 1, pp. 105–118, 1990.

[5] M. G. Awan and F. Saeed, "MS-REDUCE: An ultrafast technique for reduction of big mass spectrometry data for high-throughput processing," *Bioinformatics*, p. btw023, 2016.

[6] N. Leischner, V. Osipov, and P. Sanders, "Gpu sample sort," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[7] B. Fu, "Algorithms for large-scale astronomical problems," DTIC Document, Tech. Rep., 2013.

[8] P. Mertmann, D. Eremin, T. Mussenbrock, R. P. Brinkmann, and P. Awakowicz, "Fine-sorting one-dimensional particle-in-cell algorithm with monte-carlo collisions on a graphics processing unit," *Computer Physics Communications*, vol. 182, no. 10, pp. 2161–2167, 2011.

[9] R. Bakhtiar and F. Tse, "Biological mass spectrometry: a primer," *Mutagenesis*, vol. 15, no. 5, pp. 415–430, 2000.

[10] A. Penalver, E. Pocurull, F. Borrull, and R. Marce, "Determination of phthalate esters in water samples by solid-phase microextraction and gas chromatography with mass spectrometric detection," *Journal of Chromatography A*, vol. 872, no. 1, pp. 191–201, 2000.

[11] E. J. Finehout and K. H. Lee, "An introduction to mass spectrometry applications in biological research," *Biochemistry and molecular biology education*, vol. 32, no. 2, pp. 93–100, 2004.

[12] J. K. Eng, A. L. McCormack, and J. R. Yates, "An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database," *Journal of the American Society for Mass Spectrometry*, vol. 5, no. 11, 1994.

[13] D. N. Perkins, D. J. C. Pappin, D. M. Creasy, and J. S. Cottrell, "Probabioity-based protein idenitification by searching sequence database using mass spectrometry data," *Electrophoresis*, vol. 20, pp. 3551–3567, 1999.

[14] Nvidia. (2016, Feb.) Thrust. [Online]. Available: http://docs.nvidia.com/cuda/thrust/#axzz42Cbku41y

[15] S. Spacey, W. Luk, P. H. Kelly, and D. Kuhn, "Improving communication latency with the write-only architecture," *Journal of Parallel and Distributed Computing*, vol. 72, no. 12, pp. 1617–1627, 2012.

[16] E. Sintorn and U. Assarsson, "Fast parallel gpu-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381–1388, 2008.

[17] M. Harris, S. Sengupta, and D. J. D. Owens. (2016, Feb.) Parallel prefix sum (scan) with cuda. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

[18] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[19] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2009.

[20] J. Ajdari, B. Raufi, X. Zenuni, and F. Ismaili, "A version of parallel odd-even sorting algorithm implemented in cuda paradigm," *International Journal of Computer Science Issues (IJCSI)*, vol. 12, no. 3, p. 68, 2015.

[21] J. Sander and E. Kandrot, *CUDA by Example*, 2010.

[22] Nvidia. (2016) CUDA Toolkit Documentation v7.5. [Online]. Available: http://docs.nvidia.com/cuda/index.html#axzz42Wi4k0Qc

[23] M. G. Awan and F. Saeed, "On the sampling of big mass spectrometry data," in *Proceedings of the 7th International Conference on Bioinformatics and Computational Biology*. BICOB, 2015, pp. 143–148.

[24] M. A. Brinkman and W. G. Duffy, "Evaluation of four wetland aquatic invertebrate samplers and four sample sorting methods," *Journal of Freshwater Ecology*, vol. 11, no. 2, pp. 193–200, 1996.

[25] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice.* ACM, 2009, vol. 44, no. 6.

[26] P. Horsmalahti, "Comparison of bucket sort and radix sort," *arXiv preprint arXiv:1206.3511*, 2012.