

# Chapter 7

## Computational CPU-GPU Template for Pre-processing of Floating-Point MS Data



Fahad Saeed and Muhammad Haseeb

The data from MS spectra is usually stored as shortlist of numbers. To process these spectra, more often than not, one has to “see” inside the data to make data pre- and post-processing decisions [2]. Sorting, and searching of data for an array of numbers is one of the oldest problems in computer science. There has been significant effort in developing algorithms that can sort very large array [3]. However, for MS data, instead of having a single large array (of  $m/z$  values) there are a lot of moderately sized arrays of a very large number. We have demonstrated that sorting (or searching) of MS data with a lot of spectra is a bottleneck for many pre-processing routines [2, 4]. In order to make this pre-processing efficient, and allowing the users to be able to use our proposed techniques we have formulated a template-based GPU strategy known as GPU-DAEMON. GPU-DAEMON is a strategy that allows developers who might not be familiar with CPU-GPU architecture but would want to utilize the parallel strategy for efficient processing. We have developed a GPU-Array sort algorithm that first appeared in our paper [4] which allows us to utilize CPU-GPU architecture, and sort millions of short MS spectra. Figure 7.1 shows design of GPU-ArraySort overlaid on GPU-DAEMON template.

### 7.1 Simplifying Complex Data Structures

Since GPU-ArraySort is mostly used as integral part of a bigger algorithm, in this step the data to be sorted can be extracted from larger data structures and stored in the form of simple arrays. These arrays are then transferred over to GPU memory via the PCIe cable. Since the sorting operation cannot be further simplified, the step for simplification of computations was skipped for this algorithm.

---

Some parts of this chapter may have appeared in [1].

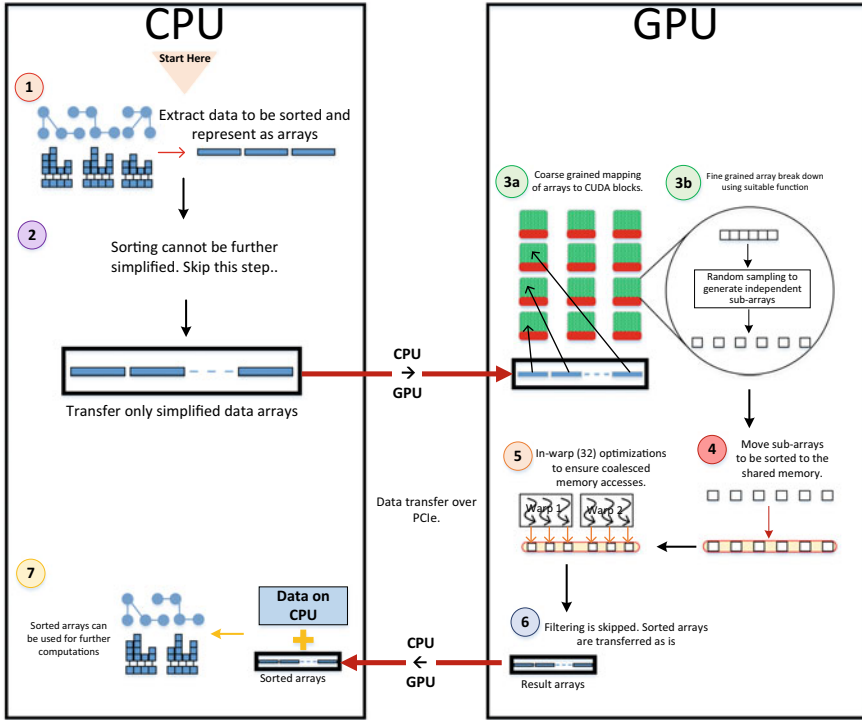


Fig. 7.1 Design of GPU-ArraySort overlaid on GPU-DAEMON template

## 7.2 Efficient Array Management

Sorting problem allows placement of the elements depending on the stored value which is presented as dependent sub-array case of Sect. 6.3.3. We get the coarse-grained mapping of each array on CUDA blocks as discussed in Sect. 6.3.3. We achieve coarse-grained mapping of each array on different CUDA blocks using the method discussed in Sect. 6.3.3. Sample-based bucketing technique was performed to exploit fine-grained parallelism [5]. We used this strategy to fragment data into sub-arrays which are mapped on the compute units. Sample-based bucketed functioning ( $F_{sub}$ ) for splitter and bucketing is discussed in the section below.

### 7.2.1 Splitter Selection

The arrays are assumed to be small enough that it fits within GPU’s shared memory, and the number of splitters required depends on the number of buckets needed for the array. The size of these buckets must be optimized to get maximum efficiency.

Our empirical study [1] has shown that the best performance is gained when the number of elements is no more than 20 [1]. This choice of size of bucket is totally independent of size of individual array as well as total number of arrays.

**Definition 7.1** If  $n$  is the size of an array, let  $B_i$  be the set of buckets for array  $i$ ,  $B_i = \{b_1, b_2, b_3, \dots, b_p\}$  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 [1, 6] have shown that if the data is uniformly distributed then 10% regular sampling resulted in the most load-balanced buckets. The samples are obtained first using the in-place insertion sort, and then the remaining  $p - 1$  splitters are chosen by picking splitter at *regular* intervals. Each block then returns to its splitter that is written to the global memory at indices calculated using the block ids that are written in consecutive memory locations with each block performing all operations using a single thread. Since the sampled array is small it can be placed inside the memory and the proposed technique is efficient to be used.

The array of splitters thus formed can be defined as

**Definition 7.2** 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  $A_i$ .  $S = \{s_1, s_2, s_3, \dots, s_N\}$  where  $s_i = \{sp_1, sp_2, sp_3 \dots, sp_q\}$  and  $q = p - 1$ .

Algorithm 5 describes a per-thread pseudo-code for first phase.

## 7.2.2 Bucketing

In this phase, the splitter values obtained from the previous phase are translated into a global array used to keep track of the bucket sizes.

**Definition 7.3** 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  $A_i$ .  $Z = \{z_1, z_2, z_3, \dots, z_N\}$  where  $z_i = \{zb_1, zb_2, zb_3 \dots, zb_p\}$ , here each  $zb_j \in z_i$  represents size of bucket  $j$  in array  $A_i$ .

Each array is assigned a unique block with threads equal to the number of buckets  $p$ . The sub-array  $sp_i$  is small in size and can be moved to the shared memory for effective and frequent usage. The pointer to these arrays can also be determined on the fly depending on the block and thread id such that each thread gets a unique pair of splitters.

**Definition 7.4** 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 splitter pair allows us to have thread that avoids branch divergence by removing other paths of the code as observed in Algorithm 4. To avoid any overlapping buckets, we use two additional splitters in sub-array  $sp_i$  by adding a splitter smaller than the smallest, and larger than the

largest value in  $A_i$ . Keeping track of a counter  $zb_j \in z_i$ , where  $j$  is the bucket and  $i$  is the array, array  $A_i$  can be traversed in parallel to complete the bucketing process which will result in each counter containing the size of the bucket. Each bucket is written back to the actual memory location of array  $A_i$ . Using this method we are able to parallelize this write back process with the advantage of saving more than 50% of device's global memory.

Algorithm 4 describes a per-thread pseudo code for second phase.

---

**Algorithm 4:** 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

```

1 splitterPair = obtainSplitters( $r_i$ )
2 initializeBucket(bucket)
3 index = 0
4 bucketIndex = 0
5 while not the end of array  $A_i$  do
6   if splitterPair[1] <  $A_i$ [index] < splitterPair[2] then
7     bucket[bucketIndex] =  $A_i$ [index] bucketIndex ++
8   index ++

```

---

### 7.3 In-Wrap Optimizations and Exploiting Shared Memory

The buckets and the sub-arrays, formed in the previous step, are small enough to fit in the shared memory. However, the sub-arrays are assigned to a warp which are placed in contiguous location in the memory to minimize memory transactions and accesses. If the size of the array is larger than the GPU memory; it must be sorted in batches. Our design and results have shown that CUDA streams data transfer, and data processing times overlap to create a pipeline like affect resulting in better processing times as compared to simple batch processing.

### 7.4 Time Complexity Model

Time complexity of GPU-ArraySort can be determined by replacing the values of  $T(f_{sub})$  and  $T(f_{proc})$  in Eq. 6.3 with  $O(\frac{n}{p})$  and  $O(\frac{n}{p} * \log(\frac{n}{p}))$ , respectively. Here  $n$  is the length of each array while  $p$  is the number of threads per CUDA block.

$$O\left(\frac{n}{p} + \frac{n}{p} * \log\left(\frac{n}{p}\right)\right) \quad (7.1)$$

---

**Algorithm 5:** 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  $A_i$ 

```

1 samples = obtainSamples( $A_i$ )
2 sortedSamples = insertionSort(samples)
3 index = 0
4 sampleIndex = 0
5 stride = calculateStride(sortedSamples)
6 while sizeOf( $s_i$ ) not equal to  $q$  do
7    $s_i$ [index] =  $A_i$ [sampleIndex]
8   sampleIndex+ = stride
9   index ++

```

---

## 7.5 Performance Evaluation

Performance evaluation of the proposed technique was performed by sorting large number of arrays, and then see which techniques can have a higher throughput (i.e. can sort more number of arrays on a given GPU). Since there is no dedicated GPU-based algorithm we used NVIDIA's Thrust Library to used stable sort by tagging them with keys. A brief explanation is given below:

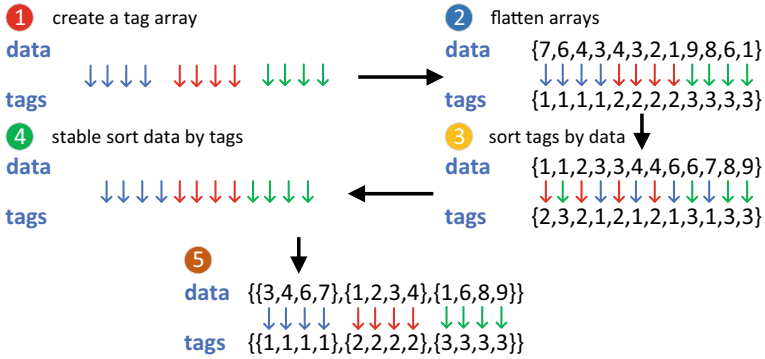
### 7.5.1 *Sorting Using Tagged Approach (STA)*

Let  $I = \{A_1, A_2, A_3, \dots, A_i\}$  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 7.5** Let  $T = \{T_1, T_2, T_3, \dots, T_i\}$  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 Fig. 7.2. It is clear that STA kind of strategy takes a lot more resources (both memory, and time) than would be needed for an optimal parallel computing strategy. Redundant work includes adding tags to the array, sorting them, and the need to sort the tag arrays in the GPU global memory. Further the STA technique uses Radix sort for sorting of the numbers which utilizes



**Fig. 7.2** 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

almost  $O(N)$  more space than the data under process [7]. The estimated memory that is used by STA is approx. 3 times more than the memory required for optimal processing [8] be required to sort all the arrays.

### 7.5.2 Runtime Analysis and Comparisons

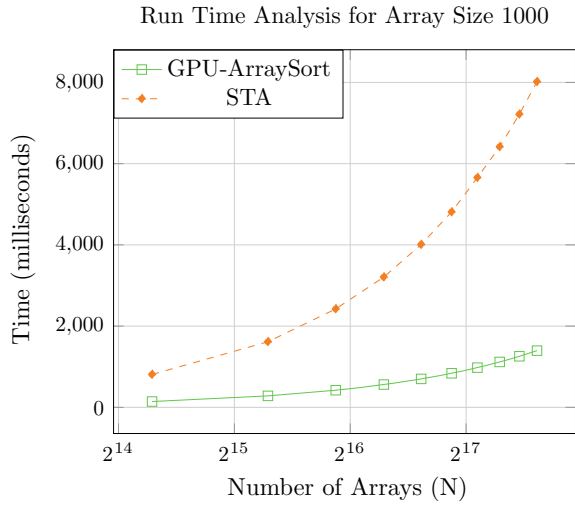
We performed the experiments to create 4 different data sets where each set consists of 200k arrays where each array was generated using uniform distribution between 0 and  $2^{31} - 1$ . The size of these arrays was 1000, 2000, 3000, and 4000 respectively for four data sets that were produced with floating point data type. All the experiments listed below were performed using 24 CPU cores each operating at 1200 MHz, Graphic Processing Unit used was NVIDIA’s Tesla K-40c consisting of 2880 CUDA cores. Total global memory available on the device was 11520 MBytes and the shared memory of 48 KBytes was available per block.

Figures 7.3, 7.4, 7.5 and 7.6 show runtime comparison between STA and GPU-ArraySort. GPU-Array sort with its superior memory efficient design out-performs STA technique for all the array sizes that we investigated.

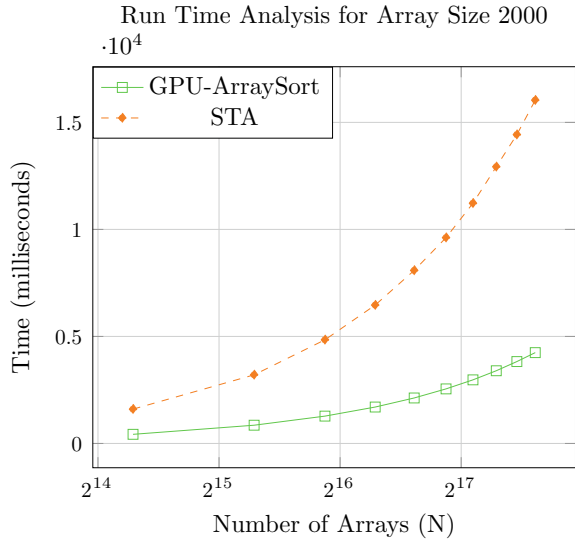
### 7.5.3 Data Handling Efficiency

The experiments that we discussed in the previous section were performed again without any bounds on the number of arrays. Table 7.1 shows that the maximum

**Fig. 7.3** 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

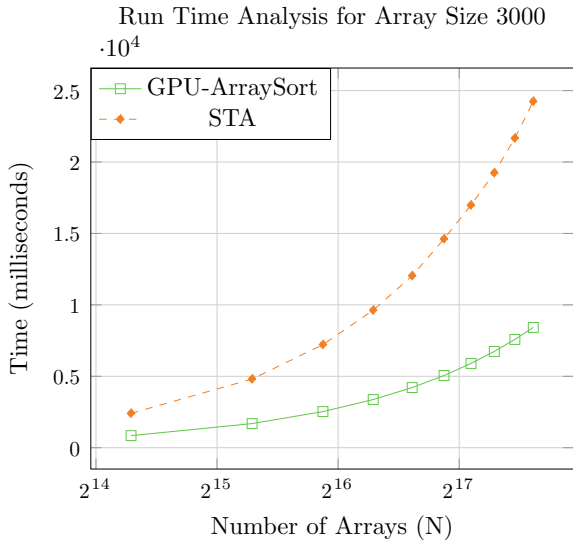


**Fig. 7.4** The figure shows time versus number of arrays plots for GPU-ArraySort and the tagged sorting approach using key-based stable sorting algorithm from Thrust library

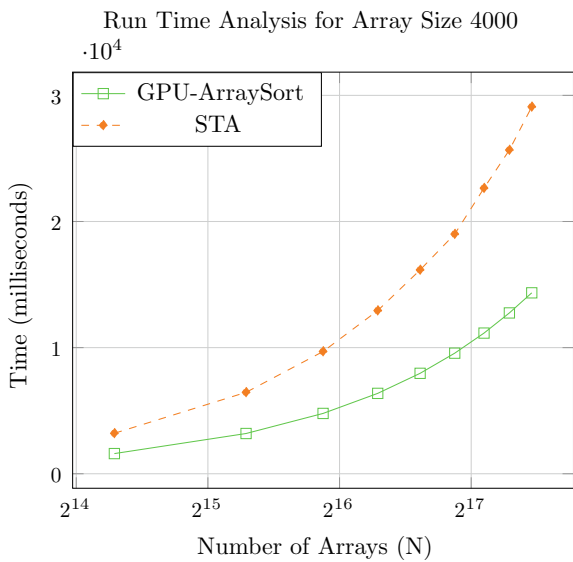


number of arrays processed by each competing method. These experiments demonstrate that GPU-ArraySort algorithm is able to process more than 3 times more arrays than the competing STA-based approach.

**Fig. 7.5** 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



**Fig. 7.6** 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





**Table 7.1** Table for memory efficiency of GPU-ArraySort

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

Note: 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

## References

1. Awan MG, Saeed F (2016) Gpu-arraysort: a parallel, in-place algorithm for sorting large number of arrays. In: 2016 45th international conference on parallel processing workshops (ICPPW). IEEE, pp 78–87
2. Awan MG, Saeed F (2016) Ms-reduce: an ultrafast technique for reduction of big mass spectrometry data for high-throughput processing. *Bioinformatics* 32(10):1518–1526
3. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore gpus. In: IEEE international symposium on parallel & distributed processing, 2009. IPDPS 2009. IEEE, pp 1–10
4. Awan MG, Saeed F (2016) Gpu-arraysort: A parallel, in-place algorithm for sorting large number of arrays. In: 2016 45th international conference on parallel processing, parallel processing workshops (ICPPW). IEEE, pp 78–87
5. Liu F, Huang M-C, Liu X-H, Wu E-H (2009) Efficient depth peeling via bucket sort. In: Proceedings of the conference on high performance graphics 2009. ACM, pp 51–57
6. Awan MG, Saeed F (2017) An out-of-core gpu based dimensionality reduction algorithm for big mass spectrometry data and its application in bottom-up proteomics. In: Proceedings of the 8th ACM international conference on bioinformatics, computational biology, and health informatics. ACM, pp 550–555
7. Horsmalahiti P (2012) Comparison of bucket sort and radix sort. [arXiv:1206.3511](https://arxiv.org/abs/1206.3511)
8. Guo Z, Huang T-W, Lin Y (2020) Gpu-accelerated static timing analysis. In: Proceedings of the 39th international conference on computer-aided design, pp 1–9