GPU-DAEMON: GPU Algorithm Design, Data Management & Optimization template for array based big omics data

Muaaz Gul Awan[a], Taban Eslami[a], Fahad Saeed[b,*]

[a]*Department of Computer Science, Western Michigan University, Kalamazoo, MI, USA*
[b]*School of Computing and Information Sciences, Florida International University, Miami, FL, USA*

Abstract

In the age of ever increasing data, faster and more efficient data processing algorithms are needed. GPUs are emerging as a cost-effective alternative architecture for high-end computing. However, optimal design of GPU algorithms is a challenging task that requires significant amount of effort and a thorough understanding of the architectural and the algorithmic design. The steep learning curve needed for effective GPU-centric algorithm design is a hinderance to widespread adoption. This situation calls for a GPU algorithm design template which outlines the critical bottlenecks and provides generic methods to tackle them and can be followed to implement high performance, scalable GPU algorithms for given big data problem. In this paper, we present GPU-DAEMON, a GPU Data Management, Algorithm Design and Optimization technique suitable for array based omics data. We study the capability of GPU-DAEMON by reviewing the implementation of GPU-DAEMON based algorithms for three different big data problems. Speed up of as large as 386x (over the sequential version) and 50x (over naive GPU design methods) have been observed using GPU-DAEMON. GPU-DAEMON template is available at https://github.com/pcdslab/GPU-DAEMON and the source codes for GPU-ArraySort, G-MSR and GPU-PCC are available at https://github.com/pcdslab.

Keywords:
GPU, CUDA, High-Performance-Computing, Big-Data, Omics-Data

1. Introduction

Computational techniques have rapidly increased the pace of scientific inquiry and progress. Big data is ubiquitous in our society today. We get large

*Corresponding Author
Email address: fsaeed@fiu.edu (Fahad Saeed)

volume and velocity of data from a variety of sources including biological experiments, social interactions, IoT sensors or other scientific investigations. Many of these sources can produce enormous amount of data in short periods of time. Faster and efficient computational techniques are essential to make sense out of the data from these various sources [1] [3]. For instance, mass spectrometry based proteomics is a problem of interest for precision medicine, cancer research and drug discovery. However, experiments in this domain produce big and complex data sets reaching peta-byte level [1] [3] [24]. Simple protein and metaproteomic library searches can take impractically long compute times [13][12]. Similarly, for proteogenomic experiments when proteomics is studied in tandem with genomics, the compute times based on existing sequential approaches become excruciatingly slow [1] [23].

Multicore and manycore devices such as GPUs, Intel-Phi and FPGAs have been shown to be useful for scaling big data problems for variety of applications [26] [9]. With the advent of these devices, there is a need to develop well-designed and scalable algorithms that can exploit the underlying HPC architecture [15] [18]. One of the most exciting devices of the modern times is Graphics Processing Unit (GPU). Because of its low cost and high performance, it is becoming the go-to device for computational labs [27] [6]. However, despite its advantages, it is a very tedious task to develop an optimized GPU algorithm. Because of the application specific designs of GPU algorithms, re-using existing designs with minor tweaks is not possible and naively designed algorithms may perform even poorer than their sequential versions [26]. To facilitate rapid designing of an optimized GPU algorithm, a set of fundamental guidelines and generic principles needs to be available. Which can be followed to develop an efficient GPU algorithm without worrying too much about the complexities of GPU architecture.

To make this process more efficient and scalable for large number of programmers and application developers and for a variety of disciplines, a generic GPU algorithm design template for big omics data must be made available. To this end, we present GPU-DAEMON (GPU Algorithm **D**esign, D**a**ta **M**anagement and **O**ptimizatio**n** template), a GPU algorithm design template for array based big omics data sets. This template can be followed by computer scientists and developers to design an efficient and scalable GPU based algorithm for big omics data, provided that the data can be transformed into array based structures. To accomplish this, we consider all the possible bottlenecks in a GPU design, methods of efficient memory management inside a GPU and the much-needed optimizations to achieve maximum occupancy and performance on GPUs.

As a proof of concept, we present three GPU based algorithms as case studies of GPU-DAEMON i.e. GPU-ArraySort [2], G-MSR [4] and GPU-PCC [9]. We based the design of these algorithms on the proposed GPU-DAEMON template which allowed us to implement extremely scalable strategies for big data from proteomics and connectomics. When using GPU-DAEMON, we report more than 386x speedup over sequential and 50x speedup over unified memory development technique. These case studies provide an insight into the thought process behind our assumptions and decisions in GPU algorithm design.

2. Discussion

To facilitate the implementation and porting of existing sequential algorithms to GPUs, techniques like Unified Memory in CUDA 6 have been introduced [22]. These aid in porting the existing codes to GPUs by introducing simple modifications in the code. These modifications include addition of structs and calls to initiate Unified Memory to make data structures accessible from CPU as well as GPU. Followed by some manipulations of code, programs can be run on GPUs. This removed the need for complete re-implementation of existing algorithms and in some cases reasonable speed-ups can be achieved. But such an implementation, is done while being oblivious to the underlying architecture and resulting algorithms are not scalable and highly under-utilize the GPU resources as shown in section 9.3. An alternate for this problem is to have GPU programming templates available, these can provide the developer with a program skeleton along with the steps of optimization to be used at each stage of development. Such templates can prove to be more robust and flexible for developing optimized GPU algorithms than techniques like Cuda Unified. The only limitation for the template based approach is that they are application and data specific and need to be designed for different applications separately.

One such template has been introduced to aid in the development of GPU algorithms in the field of remote sensing image processing [18]. For further referencing we will use the term Yan′s template for this. Yan′s template was designed to accelerate the development of optimized GPU strategies for the field of remote-sensing image processing. It provides the users with class templates and structs which can efficiently manage and store typical remote-sensing data sets that have been converted to a unified format. This conversion can be performed by using the tools that have been integrated as a part of the template. The remote-sensing data consists mostly of pixel information of images and geographical details such as longitudes, latitudes and map projections along with some other metadata. Along with data management templates, it also provides basic code skeletons as a code base to implement desired image processing operations on GPUs. Due to the data and application specific nature of this template, it would not be possible for scientists to use this template for developing GPU based algorithms for any other domain e.g. omics-sciences. Besides, Yan′s template is based on the features offered by CUDA 4.0, since then a lot of advances have been made in the field of GPU computing and with the introduction of CUDA 9.0, Yan′s template is outdated.

In comparison, the proposed GPU-DAEMON template has been specifically designed for big-omics data which can be transformed and stored in the form of arrays. Since majority of omics datasets can undergo such a transformation or naturally exist as such, GPU-DAEMON can prove to be useful for developers, working in the domain of bioinformatics. GPU-DAEMON offers a code skeleton, data management strategy and a set of optimization rules which can be followed to implement high-performance GPU based algorithms. Comparison with other nave strategies and CUDA unified method has shown superior performance for GPU-DAEMON

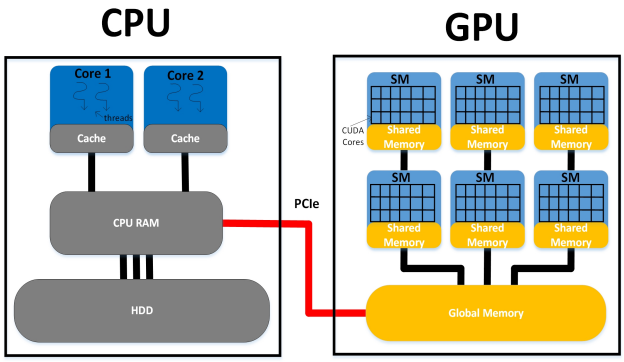

Figure 1: Figure showing CPU-GPU architecture overview. All the data transfers happen via PCIe.

3. GPU Architecture and CUDA

Graphics processing units (GPUs) were introduced as dedicated graphics accelerating devices. These can run millions of compute units in parallel; which allows them to process individual elements of an image matrix in parallel. A set of computations which can be reduced to simple matrix manipulations can take advantage of GPUs massively parallel processing power [10].

A GPU contains several Streaming Multiprocessors(SM) each of which contains multiple CUDA cores. Number of these units on each device varies with the GPU model. For instance, a K-40 Tesla GPU contains 15 SMs with 192 CUDA cores each, making a total of 2880 cores. While GTX 1080Ti GPU contains 28 SMs with 128 CUDA cores each.

Each SM has a fast on-chip memory which is shared among its cores and is called the Shared Memory. It is about 100x faster than the GPU global memory but is quite small and usually varies from 32 kByte to 64 kByte depending upon the GPU [2]. It is also available to be used as a user defined cache. An off-chip memory of much larger size, called Global memory is used for storing data and communicating with the host. Sizes of Global memory are of the order of GBs. Fig. 1 shows an overview of CPU-GPU architecture.

3.1. CUDA Environment

3.1.1. CUDA overview

With increasing interest in GPUs for data processing applications, NVIDIA introduced CUDA platform to aid in the use of their GPUs for general purpose processing. CUDA is a programing environment which can be used with multiple programming languages to implement programs on GPUs [21]. It forms a software overlay and provides programmer an easy access to programmable features of a GPU. CUDA uses SIMT (Single Instruction Multiple Thread) model, which combines the usual SIMD (Single Instruction Multiple Data) with multiple threading thus providing two levels of parallelism [20] [16]. CUDA compute

units are arranged in the form of a Grid of Blocks while each block contains several threads. Number of threads and blocks are determined by the compute capability of the device. Each thread within a block is assigned two IDs; a block ID and a thread ID, using these a unique ID can be calculated for each thread. In a CUDA model the SMs in Fig. 1 would be replaced by blocks and CUDA cores with threads.

3.1.2. CPU-GPU computing

In CPU-GPU computing, CPU acts as the host and offloads tasks to the GPU which behaves as a coprocessor. Data from CPU RAM is transferred to GPUs global memory via a PCIe cable and a set of instructions known as CUDA *Kernel* is launched on the GPU. Each CUDA compute unit executes these instructions independently. Once the kernel completes execution, the results are copied back to the host. First step in designing any GPU algorithm is to carefully profile the problem and offload only the most compute intense and data independent tasks to the GPU [4]. In a lot of cases, CPU can perform tasks faster than GPU when the data transfer overheads are taken into considerations [11].

4. Challenges in GPU algorithm design

Following is an overview of common challenges and bottlenecks faced in the design of efficient GPU algorithm.

4.1. Need for Data Parallel Design

Even though large in number, the GPU compute units are quite simple without deep pipelines or any optimizations for executing long lines of codes in an efficient manner. Best way of exploiting GPUs power is to design a data parallel algorithm such that each compute unit has to perform simple operations while being independent of results from other units.

4.2. Data transfer Bottlenecks

The part of algorithm which is offloaded to GPU, requires that the data be present in the GPU memory before a kernel can be launched. This data transfer happens via a slow PCIe cable. The potential speedups are subdued if the time needed to transfer the data from CPU to GPU is larger than the execution time of the program. Hence efficient techniques are required to reduce the amount of data transfers. Similarly, the data generated on GPU after execution of the kernel can be much larger than the input and may require novel result sifting techniques to avoid GPU-CPU transfer bottlenecks.

4.3. Non-Coalesced Memory Accesses

Active GPU threads are grouped into chunks of 32 threads called a *warp*. These warps are scheduled onto SMs as the resources become available. Global memory accesses from threads of a warp are coalesced together into same memory transaction if the locations being accessed have spatial locality. Otherwise threads access global memory in multiple transactions, which stalls the execution of warp until the data is available. This problem considerably slows down execution by reducing the number of concurrent compute units active at a given time.

4.4. Warp Divergence

In an SIMT execution, threads in a warp execute in a lock step which means that all the instructions are executed simultaneously by the threads of a warp. In case of a branch or if threads must diverge, a warp divergence occurs leading to a loss of efficiency and slowdown of a GPU algorithm. One of the challenges in efficient GPU algorithm design is to minimize the warp-divergence.

4.5. Exploiting Coarse Grained and Fine Grained Parallelism

GPU offers two levels of parallelism, to exploit each level to its fullest fine-grained data management techniques are required. This requires that the data be managed in such a way that it can be disintegrated to a fine level. In the absence of such technique large amount of GPU resources are left unutilized.

5. Basic Principles of GPU-DAEMON

The proposed GPU algorithm design approach provides a template for the design of GPU algorithm for big-omics data. GPU-DAEMON is divided into seven steps, each step proposes a generic solution for tackling a GPU bottleneck. These solutions need to be specialized depending upon each application. Fig. 2 shows flow of the steps in GPU-DAEMON.

The first step is to analyze and profile the algorithm under consideration to determine compute and data intensive parts. These data and compute intense tasks are reserved for GPU while other bookkeeping and simpler operations are left for CPU side. After this we begin by considering each bottleneck step by step.

5.1. Simplifying Complex Data Structures

Data is mostly stored in the form of larger data structures for ease of access and better organization purposes. First bottleneck occurs when large data structures are transferred over PCIe. Approach of transferring complete data structures is very easy and attractive for any programmer since no considerable redesigning of the algorithm/code is required. However, most of the computations performed on GPUs require only a part of large data structures. As a first step, instead of transferring complete data structures to GPU, data should be transformed such that only the portion required by GPUs part of processing is transferred. This considerably cuts the transfer bottleneck created by sluggish PCIe.

5.2. Simplifying Complex Computations

The architecture of GPU compute units is very simple and is not capable of processing more involved or large number of operations in an efficient way. Their shallow pipelines and naive designs make them specialized for simpler computations.

Keeping this in mind, in the second step of GPU-DAEMON, complex computations are simplified. For instance, converting floating point numbers into integers or representing them in binary will make GPU computations much simpler and faster. Depending on the algorithm, in some cases simpler logical operations will simplify the computations for a complex algorithm. At times precision is too important to simplify computations. In such a scenario GPU can be used to process a simpler representation of more complex data to approximate simpler solutions. These simpler solutions can then be processed to yield more precise results on CPU with the aid of actual data. In such a scenario GPU aids by performing the brunt of processing and CPU just does the bare minimum. These simplification techniques may vary depending upon the application, we discuss them in more detail in case studies.

5.3. Efficient Array Management in GPU

A scalable GPU algorithm design boils down to array management strategies inside CPU-GPU architecture. This step discusses management and mapping of data to CUDA compute units to achieve fine grained parallelism. As discussed before, compute units in CUDA platform are classified as *Blocks* and *Threads*. To fully exploit this two-level parallelism, we recommend an array fragmentation strategy. This can be done in two steps:

5.3.1. Coarse Grained Distribution

First we perform coarse grained distribution by mapping each array to a unique block. This mapping is feasible because the number of CUDA blocks is much larger than the number of arrays which a GPU can hold in its memory at a given time.

5.3.2. Fine Grained Distribution

In second step, we achieve fine grained mapping by segmenting each array into subarrays and then mapping each subarray to a cluster of threads or a single thread depending upon the nature of problem. The data mapping can be divided into two categories each requiring a different approach.

If the nature of data and target function is such that each element can be processed independently then considering an array of size m, following number of elements assigned per thread should suffice:

$$E_i = \frac{m}{nT} \tag{1}$$

$$E_{nT-1} = E_i + m \bmod (nT)$$

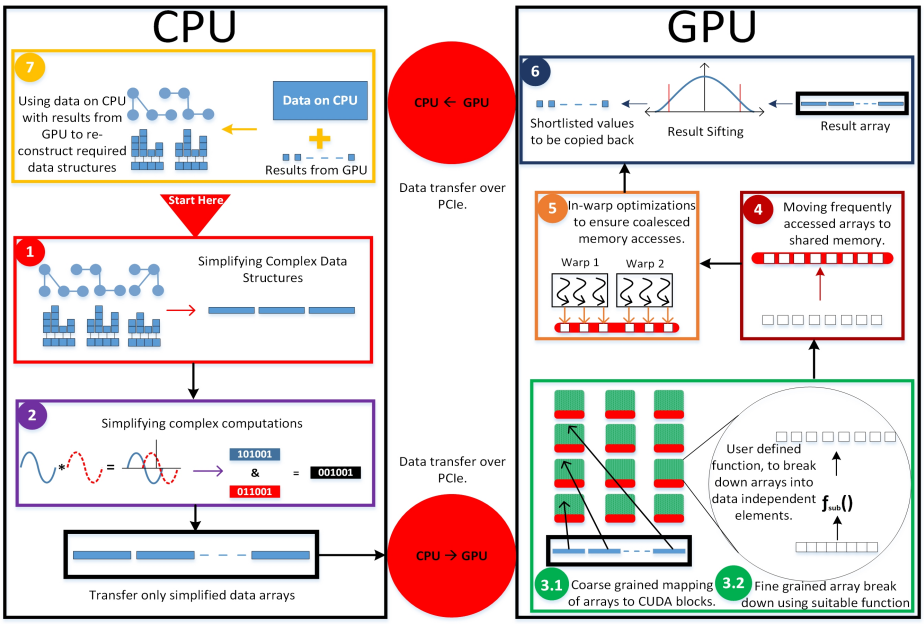

Figure 2: Figure shows the template for GPU-DAEMON

where E_i is the number of elements to be mapped to thread i where nT is the total number of threads available per block, and E_{nT-1} represents the number of elements mapped to the last thread in block. Here we assume that Thread IDs start at position 0. Start and end indices for subarray assigned to each thread can be calculated as:

$$SI_i = i * E_i$$

$$EI_i = (SI_i + (i+1) * E_i) - 1$$

$$EI_{nT-1} = (EI_{nT-2} + E_{nT-1}) - 1$$

Here SI_i and EI_i are the locations for first and last elements of the subarrays assigned to thread i, respectively.

When the nature of data and operations to be performed are such that elements cannot be processed independent of each other, then elements need to be divided into data independent subsets using a suitable user defined function as shown in GPU-DAEMON template Fig. 2. We denote this function by F_{sub}.

5.4. Exploiting shared memory

Shared memory is about 100x faster than the global memory. To utilize this speed most frequently accessed parts of data should be moved to the shared memory. But doing so may not always yield better results [4] [2]. If the following equation holds then it is reasonable to move the data to shared memory:

$$(T_{tf}) + (P_{SM}) < (P_{GM}) \quad (2)$$

Here T_{tf} is the time to move data from global to shared memory while P_{SM} and P_{GM} are the processing times in Shared and Global memory, respectively.

5.5. In-Warp Optimizations

This step considers optimization strategies which can be used to get the best out of a GPU. As discussed in section 4, thread divergence inside a warp and lack of memory coalescing in accessing global memory leads to loss of performance. Among these two the latter has more dominant affect. Thread divergence can be avoided by redesigning the algorithm so that threads of a warp do not diverge.

To achieve global memory coalescing a good thread to data mapping strategy is needed. The mappings discussed in 3rd step of GPU-DAEMON simplifies this mapping. By mapping consecutive threads to independent contiguous array segments of step 3 can help achieve memory coalescing.

5.6. Result Sifting

Output arrays generated on GPU as a result of big data processing can be very large in size, at times even larger than the size of total input [14]. Copying these back to CPU over PCIe cable results in a memory transfer bottleneck. This step of GPU-DAEMON deals with techniques for removing this bottleneck. These techniques can include either compressing the results or copying back only the most interesting results while filtering out the others. These methods are application specific and are not generalized in our proposed strategy. We discuss a memory efficient methods of result sifting in case study 2.

5.7. Post Processing Results

If in the first step, if a transformation is performed on the data to simplify the transfer and processing then there may be a need for a post processing phase. This phase is mostly performed on the host processor and is basically an inverse of the data transformation performed in the first step.

5.8. Out-Of-Core Design

Modern GPUs have a very limited in-core memory, when dealing with big data it is essential that the algorithm can work out-of-core. For each case study, we discuss an out-of-core processing approach which minimizes the CPU-GPU communication while maximizing the through-put.

5.9. Time Complexity Model for GPU-DAEMON

Any algorithm developed using GPU-DAEMON will have total time T_{tot} comprising of two terms

$$T_{tot} = T_{CPU} + T_{GPU}$$

where T_{CPU} is the total time complexity of CPU part of the design and T_{GPU} is the total time complexity of GPU part of the design. Here we will give a generic formulation for T_{GPU}, this formulation can be used to derive the actual time complexity of the GPU part of the algorithm.

T_{GPU} depends on the time taken to disintegerate a given array into data independent segments (T_{sub}), time for processing the data independent arrays (T_{proc}) and the time for result sifting step (T_{sift}) i.e. $T_{GPU} = T_{sub}+T_{proc}+T_{sift}$. If we consider N arrays with each of size n then the total time for applying disintegeration function f_{sub} to N arrays on GPU would be equal to $T_{sub} = \frac{N}{B} * (\frac{T(f_{sub})}{p})$ where B is the number of Cuda Blocks active at a given time, p is the number of threads active per block and $T(f_{sub})$ is the time for f_{sub}. Similarly, we can compute $T(f_{proc})$ to be $\frac{N}{B} * (\frac{T(f_{proc})}{p})$ for processing function f_{sub} and $T_{sift} = \frac{N*x*T(f_{sift})}{B*p}$ for result sifting function f_{sift}. Here x is the number of elements in each result array. This gives us:

$$T_{GPU} = \frac{N}{B * p} * (T(f_{sub}) + T(f_{proc}) + x * T(f_{sift})) \tag{3}$$

6. Case Study 1: GPU-ArraySort

Sorting a given list of numbers is one of the most studied problem in computer science. A lot of algorithms have sorting as an integral step [3]. There is a large number of sequential and parallel algorithms available for sorting one large array of numbers [25] but not much effort has been made to tackle the problem of sorting large number of moderately sized arrays. Sorting large number of moderately sized arrays results into a computational bottleneck in several algorithms [3] [2].

To this end, we present a GPU based array sorting algorithm capable of sorting large number of moderately sized arrays. This algorithm was first presented in [2]. GPU-ArraySort was developed following the GPU-DAEMON template, here we briefly review the implementation of GPU-ArraySort as a case study for GPU-DAEMON. Fig. 3 shows design of GPU-ArraySort overlaid on GPU-DAEMON template.

6.1. 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.

6.2. Efficient Array Management

In a sorting problem, correct placement of each element depends on the value of every other element, so this presented a tailor-made situation for dependent sub-array case of section 5.3. The coarse-grained mapping was achieved by mapping each array to a different CUDA block using the method discussed in section 5.3. To perform fine grained segmentation, we made use of sample based bucketing technique [17]. Using this strategy, the larger arrays were fragmented into smaller data independent sub-arrays and then mapped to fine grained compute units. In GPU-DAEMON template, F_{sub} was replaced with sample based bucketing function.

A pseudo code for the bucketing phase of the algorithm is given in supplementary materials. The splitters here are sampled from the array to be sorted and determine the upper and lower bound of a given bucket.

6.3. Exploiting shared memory

Independent subarrays from previous step were moved to the shared memory for in-place sorting operation. Since the subarrays are quite small in size, they can always fit inside the shared memory.

6.4. In-Warp Optimizations

The independent sub-arrays were then assigned to threads of a warp for sorting. The sub-arrays assigned to threads of a warp were placed in contiguous locations in the memory. This minimized the number of memory transactions required by each warp thus optimizing the memory accesses.

The next step of the GPU-DAEMON template was skipped for GPU-ArraySort because result sifting is required when all of the results are not of interest or the results are impractically large to be transferred back. In this case our output was sorted arrays which had to be transferred back completely, so we skipped the result sifting step.

6.5. Post Processing Results

The sorted arrays can then be used for remainder of the processing of the algorithm of which the array-sorting was part of.

6.6. Out-Of-Core Design

It happens very often when all the arrays to be sorted cannot fit inside the GPU memory and have to be sorted in batches. Our results showed that the use of CUDA streams to overlap the data transfer and data processing times created a pipeline like affect and gave smaller processing times than simple batch processing.

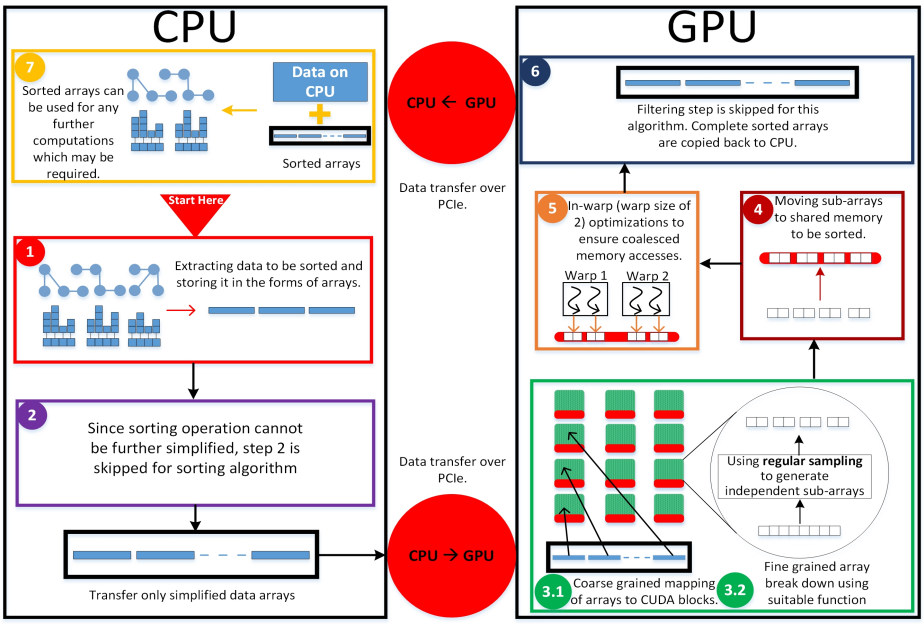

Figure 3: Design of GPU-ArraySort overlaid on GPU-DAEMON template

6.7. 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. 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. Since GPU-ArraySort does not have any pre- or post-processing steps, remaining values will be replaced by zero.

$$O(\frac{n}{p} + \frac{n}{p} * log(\frac{n}{p})) \tag{4}$$

7. Case Study 2: Proteomics Dimensionality Reduction algorithm (G-MSR)

Now a days, high-throughput study of proteins is being performed using a Mass Spectrometry (MS) based proteomics pipeline. This pipeline consists of analyzing a protein sample using Mass Spectrometers and processing that data using a software pipeline for rapid protein sequencing and assessment [8] [3][7]. The number of spectra generated by an MS can vary between thousands to up to a billion depending on the nature and the objective of a systems biology experiment. Each spectrum is a set of 2-tuples where each tuple consists of a mass to charge ratio and a corresponding intensity; we call each such tuple a peak [13] [4]. For accurate and timely peptide deduction, preprocessing

of this data is an essential part of the proteomics pipeline. However, conventional pre-processing algorithms are very slow and take days of computations [citewuDenoise [19]. As an attempt to resolve this problem, we introduced G-MSR, a GPU based dimensionality reduction algorithm for proteomics data [4]. G-MSR is a GPU-DAEMON based implementation of previously presented MS-REDUCE algorithm [3].

G-MSR algorithm basically consists of three major steps i.e. Spectral Classification, 2) Quantization and 3) Weighted Random Sampling. At input it accepts a spectrum s and a reduction factor R. If size of the input spectrum can be denoted by $|s|$ then at the output G-MSR will generate a reduced spectrum of size $R * |s|$.

In the classification stage, based on the estimate of noise content each spectrum is classified into one of four classes. Spectra belonging to different classes are then quantized i.e. peaks in each spectrum are then grouped based on their tendency to be a significant peak. Finally, using a weighted random sampling stage, peaks are randomly sampled from each quantum to form a reduced spectrum. The sampling weights ensure that only the most significant peaks make it to the final reduced spectrum. The weighted random sampling step is governed by the following total peak equation:

$$\sum_{i=0}^{n} \frac{x_i}{100} = p' \tag{5}$$

Here x_i is the sampling weight for the i-th quantum, q_i is the number of peaks in the quantum i, p' represents the total peaks in the reduced spectrum and n is the number of quanta for given spectrum.

Fig.4 shows the design of G-MSR overlapped on the GPU-DAEMON template.

7.1. Simplifying Complex Data Structures

As discussed before, the mass spectra obtained from MS consist of mass to charge ratios and their corresponding intensities. In a naive methodology complete spectra along with meta-data would be transferred over the PCIe cable to GPU for processing. But following the GPU-DAEMON template we separate the intensities from the larger data structure in the forms of multiple arrays (one array for each spectrum) and only transfer them over to GPU memory. This cuts down the amount of data being transferred by more than 50%. The actual spectra are kept on the host for book-keeping and post processing phase.

7.2. Simplifying Complex Computations

Since intensities are floating-point numbers, we round them off to nearest integer before transferring them to GPU. This converts all the floating-point computations to integer computations thus simplifying the computations. As shown in [4], this approximation does not affect the algorithm′s performance.

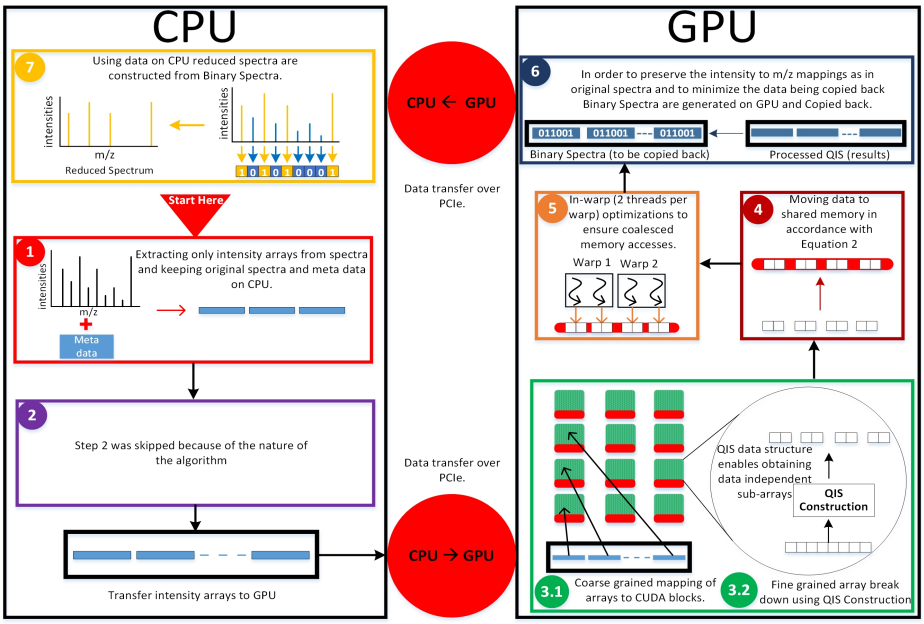

Figure 4: Design of G-MSR overlaid on GPU-DAEMON template

7.3. Efficient Array Management

The quantization stage of the dimensionality reduction algorithm discussed in [3] transforms the spectra into 3-Dimensional data structures. Managing this 3-D data structure is challenging for data processing on a GPU architecture [5], also in-order for GPU-DAEMON′s array management technique to work we need to map the data into a 1-Dimensional array.

To achieve this, we introduced a novel data structure called Quantized Index Spectrum (QIS) which maps a 3-D quantized spectrum onto a 1-D array which can then be easily managed using the techniques discussed in section 5.3. The QIS data structure serves a dual purpose of transforming 3-D quantized spectra to 1-D array while performing the step of quantization.

As discussed before, the quantization step basically groups together the peaks of a spectrum. In a QIS data structure, these groups of peaks are present in contiguous memory locations, with a separate array of pointers keeping track of starting and ending points. Each of this group can be considered as a sub-array, since these sub-arrays are independent of each other we can use the strategy of section 5.3 for exploiting fine-grained parallelism. For G-MSR algorithm, we replace F_{sub} by QIS construction in GPU-DAEMON template. Detailed design and implementation of QIS data structure is out of the scope of this paper and can be found in [4]

7.4. Exploiting Shared Memory

To better exploit the shared memory, sub-arrays are then moved to the shared memory for further processing if the Eq. 2 is satisfied.

7.5. In-Warp Optimizations

The sub-arrays created by the QIS are a part of a larger array, with their beginning and end pointers listed separately. So, all the sub-arrays created by QIS are in contiguous memory locations. This feature of QIS helps ensure that when consecutive sub-arrays are processed by consecutive threads of a warp, memory coalescing takes place.

7.6. Result Sifting

In the first step, rather than transferring complete spectra we transferred only the part which was needed for GPU-processing, this and because of the random sampling which takes place in the third phase of dimensionality reduction algorithm [3], it becomes difficult to maintain which intensities are eliminated on the GPU-side. To tackle this problem, we used an additional property of QIS data structure i.e. the indices of peaks which are eliminated on the GPU-side are retained with a place-holder. These place-holders help in constructing a binary spectrum indicating the indices of intensities to be retained in the reduced spectrum. We define Binary Spectra as:

Definition: Given a spectrum $s_i = \{p_1, p_2, p_3, \ldots, p_n\}$ a Binary Spectrum B_i for the corresponding reduced spectrum s'_i is defined as, $B_i = \{e_j = 1 | p_j \in s'_i\} \cup \{e_j = 0 | p_j \notin s'_i\}$.

In other words, if a peak at index j in s_i is included in the reduced spectrum then there will be a 1 at index j of B_i; otherwise it will be zero.

For each spectrum, a Binary Spectrum is generated and only these Binary Spectra are then copied back to CPU. Binary spectra are memory efficient and helpful in quick reconstruction of reduced spectra on the CPU side. Introduction of QIS and Binary Spectra thus enabled G-MSR to copy back just bare minimum and resolve the GPU-CPU bottleneck.

7.7. Post Processing Results

The Binary Spectra copied back in the previous phase are then used for constructing the reduced spectra on the CPU side as shown in Fig. 4.

7.8. Time Complexity Model

To compute the time complexity of G-MSR we replace $T(f_{sub}) = O(\frac{N}{B}) + O(\frac{N*n^2}{B*p}) + O(\frac{N*n}{B})$ and $T(f_{proc}) = O(\frac{s*N}{B})$ in Eq. 3. Here the f_{sub} time includes sorting, classification and construction of QIS data structure while the f_{proc} time consists of weighted random sampling phase. Replacing the values in Eq. 3 and simplifying leaves us with:

$$O(\frac{N * (n^2 + l)}{B * p}) \tag{6}$$

where $l = p * (2 + n + n * s)$ and s is the sampling rate.

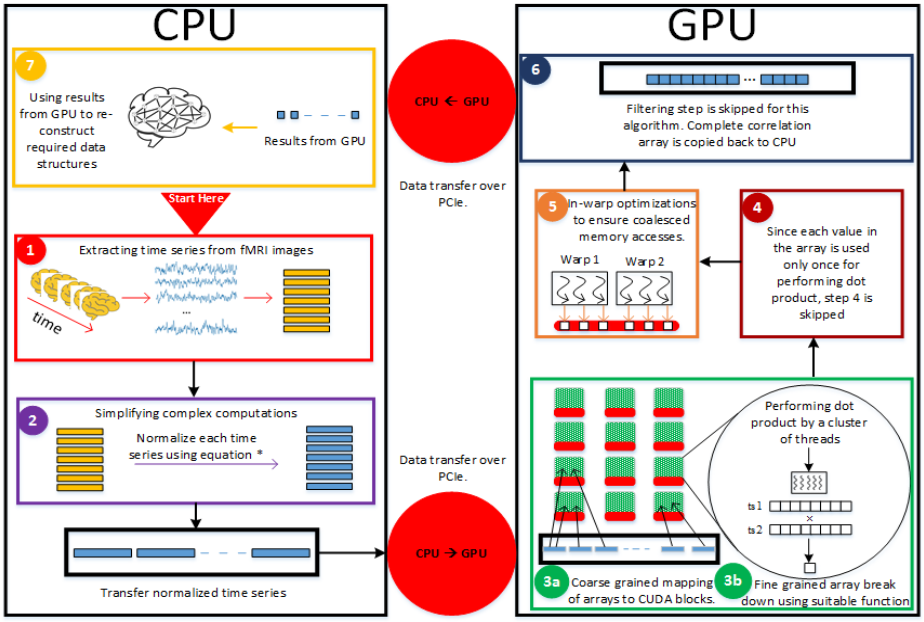

Figure 5: Design of GPU-PCC overlaid on GPU-DAEMON template

8. Case Study 3: GPU-PCC

Studying functional connectivity of the brain is one of the important aspects of connectomics. Functional magnetic resonance imaging (fMRI) is a widely used brain imaging technique for exploring activity of the brain. In fMRI technology, several images are taken from different levels of the brain over time. These images are divided into huge number of very small cubic elements called voxels each containing thousands of neurons inside them. A time series is extracted out of each voxel which shows how its activity is changing during the scanning time. Functional connections among voxels can be constructed by calculating pairwise Pearson′s correlation between them. Since there are huge number of voxels in an fMRI scan, this process is really time consuming and using GPU based techniques can help reduce the running time significantly. Previously, we presented a GPU based algorithm called GPU-PCC to accelerate this process [9]. In the following sections we explain how this algorithm has been implemented using the GPU-DAEMON template. Fig. 5 shows the design of this algorithm based on GPU-DAEMON template.

8.1. Simplifying Complex Data Structures

The data which is used for computing correlation coefficients is the time series of each voxel of the brain. This time series data cannot be further simplified since all values over time are needed for computing correlation so we skip this section in this algorithm.

8.2. Simplifying Complex Computations

By normalizing the time series of each voxel using Eq. 7, computing Pearson′s correlation between two voxels can be reduced to dot product of their normalized time series which needs fewer number of operations rather than applying Pearson′s correlation equation. In this equation v_i denotes the original time series and u_i denotes the normalized time series.

$$u_i = \frac{v_i - \bar{v}_i}{|v_i - \bar{v}_i|_2} \tag{7}$$

8.3. Efficient Array Management

In GPU-PCC each block is responsible for computing several correlation coefficients at the same time. Time series of voxels whose correlation is computed by block i are mapped to that block. Hence, the coarse-grained distribution is achieved. On the other hand, threads inside each block are divided into small clusters and each cluster is responsible for performing the dot product of two time series. The size of this cluster is smaller than the length of the time series and multiplication of corresponding elements can be done independently, so each cluster traverse the array and performs the multiplication until it reaches the end of the array. The number of times each cluster shifts over the array can be computed using Eq. 1 and in this case nT is the number of threads per cluster. The final value of correlation is computed by summing up the partial sums that each thread is computed using warp shuffling technique.

In GPU-PCC each block is responsible for computing several correlation coefficients at the same time. Since time series of voxels are basically arrays, so using the mapping approach in section 5.3 we can map each array to a unique block thus achieving coarse grained distribution. An observation of the time series data shows that this is the case of independent sub-arrays from section 5.3 so each array can be processed using a strategy similar to discussed in that section.

We compute the dot product of two series by multiplying smaller sub-arrays and then summing up the partial results into the final value. This strategy is same as discussed in Efficient Array Management part of GPU-DAEMON from section 5.3. To achieve this, we group the threads inside each block into small clusters and then each cluster is responsible for performing the dot product of two arrays. Since the size of this cluster is smaller than the length of the array, the arrays are fragmented to smaller subarrays and the threads inside these clusters multiply corresponding elements of sub arrays. The results of these products are stored inside thread registers, and are then used for computing the final value of dot product. After multiplying corresponding elements of the first subarray, the cluster shifts to the next subarray. The value computed by each thread is added to the result as computed for previous subarray. This process is repeated until this cluster traverses all subarrays. At the end, the partial values computed by threads inside each cluster are summed up to the final value of the dot product using warp shuffling technique.

8.4. Exploiting Shared Memory

The only step that needs to be performed for computing correlations is the dot product of two corresponding arrays. Since each thread multiplies two corresponding values right after reading them from global memory and doesn't use them any further, copying data to shared memory adds additional overhead to the computation so we skipped this step in GPU-PCC.

8.5. In-Warp Optimization

Since each thread is performing the same task which is multiplying two corresponding values, the branch divergence is avoided. Threads in each cluster belong to the same warp and access contiguous memory locations of global memory which ensures global memory coalescing.

8.6. Result Sifting

Since the whole correlation array may be needed for further processing, we skip this section in GPU-PCC and copy the correlation array to CPU.

8.7. Post Processing Result

Since there was no transformation applied on data for simplification and neither does this algorithm require any type of result sifting, no post-processing was needed for GPU-PCC.

8.8. Out-Of-Core Design

Size of correlation array is usually greater than the size of GPU global memory, so GPU-PCC computes the correlations in several rounds. In each round, pairwise correlations are computed based on free space in GPU and copy back to CPU when there is no more space to store them. This strategy minimizes the number transfers from GPU to CPU.

8.9. Time Complexity Model

In order to compute the total time complexity of GPU-PCC, we replace T_{CPU} in Eq. 3 with $O(Nn)$ which is the time complexity of normalizing time series of voxels using Eq. 7. Here N corresponds the to number of voxels and n is the length of time series. Correlations are computed by performing dot product between normalized time series which can be performed in parallel by threads and there is no need for disintegration function (F_{sub}). As stated earlier, no post processing is performed on final correlations hence $T(F_{sub})$ and $T(F_{sift})$ are replaced with zero. Dot product between two vectors has linear time complexity and is performed in parallel by a cluster of l threads which results in $O(\frac{n}{l})$. A total of $\frac{N(N-1)}{2}$ pairwise correlation computations are divided among blocks and each block computes $\frac{p}{l}$ correlations concurrently which gives total time complexity for GPU-PCC as $O(\frac{N^2}{B \times \frac{p}{l}} \times \frac{n}{l})$. Here, B and p are the number of CUDA blocks and threads per block respectively. Simplifying, we get

$$= O(\frac{N^2 n}{Bp})$$

9. Experiments and Results

Here we discuss the experiments and their results for each of the algorithms developed in the three case-studies above.

9.1. Experimental Conditions

For all the following experiments we used a server with 24 CPU cores each with operating frequency of 1200 MHz. The server was equipped with a Tesla K-40c GPU which has 15 Multiprocessors with each Multiprocessor having 192 CUDA cores, making the total number of CUDA cores equal to 2880. Total Global memory available on the device was equal to 11520 MBytes and the shared memory was of 48 KBytes.

9.2. For GPU-ArraySort

To assess the efficiency of GPU-ArraySort we compared it against the NVIDIA′s Thrust Library′s stable sort algorithm. The Thrust library though widely used for sorting one large array can be used to sort a large number of small arrays using what we call the tagged array sorting or STA. This approach has been discussed in [2] in detail.

We tested both the algorithms for sorting the arrays of sizes 1000, 2000 and 3000 floating point elements each and then taking the average of runtimes. Fig. 6 shows a comparison between the execution times of GPU-ArraySort and the STA approach using Thrust Library. It can be observed that GPU-ArraySort provides a speed up of over 3-4x over the more popular GPU array sorting method.

9.3. For G-MSR Algorithm

To assess the performance of G-MSR (a GPU-DAEMON based version of MS-REDUCE algorithm), we compared it against a unified memory based GPU implementation of MS-REDUCE. The unified memory technique enables quick and easy development of GPU based algorithms. For our purpose, we simply took the sequential version of G-MSR [3] and modified the code following rules of GPU algorithm development using CUDA unified memory [22].

For scalability study, we appended the UPS2 dataset (details of datasets can be found in Supplementary Materials Section 1) multiple times to get progressively larger datasets.

Fig. 7 and Fig. 9 shows that GPU-DAEMON based implementation consistently out-performs the naive implementation. It can be observed in Fig. 9 that CUDA unified memory based implementation reaches its in-core memory limit at only 14,000 spectra, while G-MSR as shown in [4] reaches its in-core memory limit at 400,000 spectra. Along with better speed, GPU-DAEMON helps conserve limited in-core memory so that more throughput can be achieved. Fig. 8 shows that GPU-DAEMON version uses a very small amount of in-core memory in comparison to the unified memory based implementation.

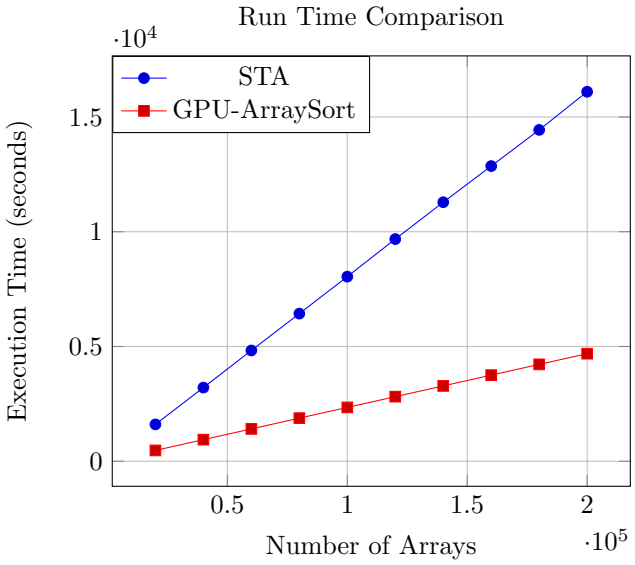

Figure 6: Comparison of average run times of STA sorting technique and GPU-ArraySort. Run times here are an average of times it takes to sort arrays of 1000, 2000 and 3000 elements

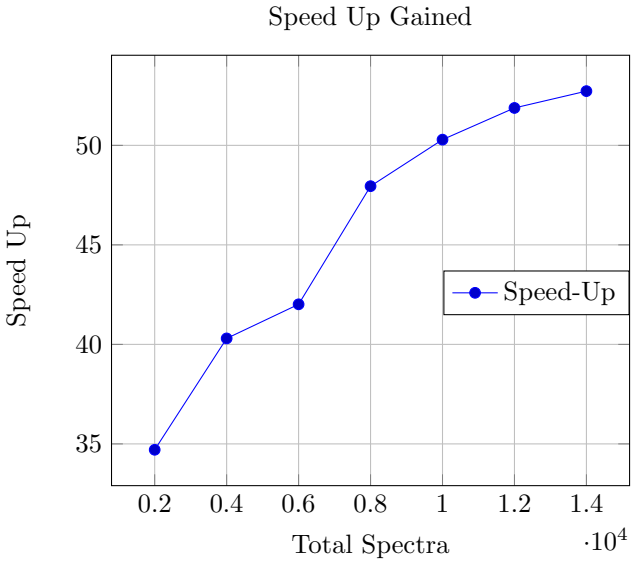

Figure 7: Total speed up achieved by GPU-DAEMON implementation over CUDA unified memory based implementation.

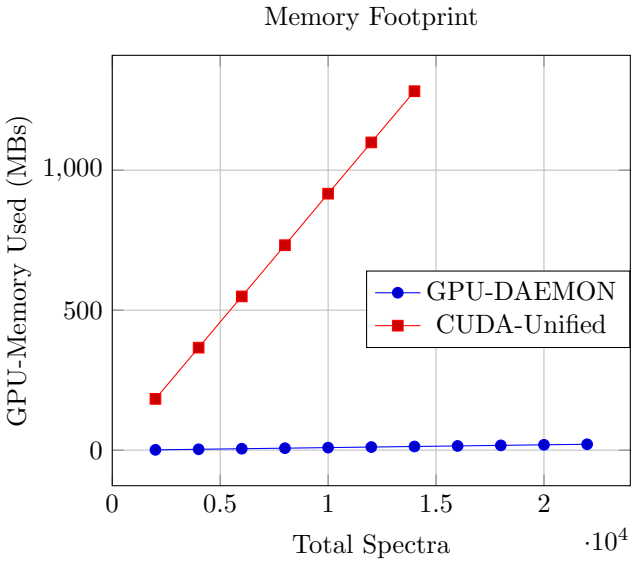

Figure 8: Figure shows that GPU-DAEMON based implementation of MS-REDUCE uses only a fraction of memory as used by the CUDA unified memory implementation.

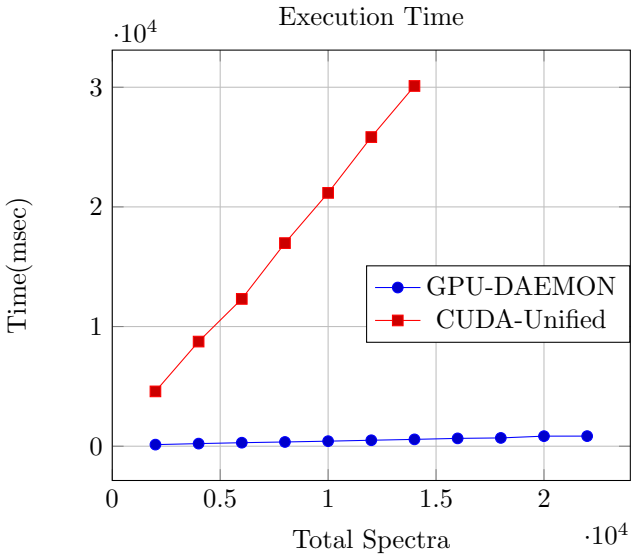

Figure 9: Figure shows that GPU-DAEMON based implementation of MS-REDUCE scales better with increasing spectra. It should be noticed that CUDA unified memory based version reaches in-core limit earlier and cannot process more than 14,000 spectra in a single pass while GPU-DAEMON implementation can process about 400,000 spectra before that limit is reached [4].

9.4. For GPU-PCC Algorithm

We compared the running time of GPU-PCC (GPU-DAEMON based implementation) with sequential and a naive GPU-based implementation for computing Pearsons correlation. In the first experiment, we used synthetic time series and in the second experiment, we used real fMRI data. Details of both the datasets can be found in supplementary materials Section 1.

Fig. 10 shows the running time comparison of the three mentioned approaches based on increasing the number of voxels, using synthetic datasets. Table 1 shows the running time comparison of different techniques on real data. Same server and GPU was used for these experiments as for the previous two case studies.

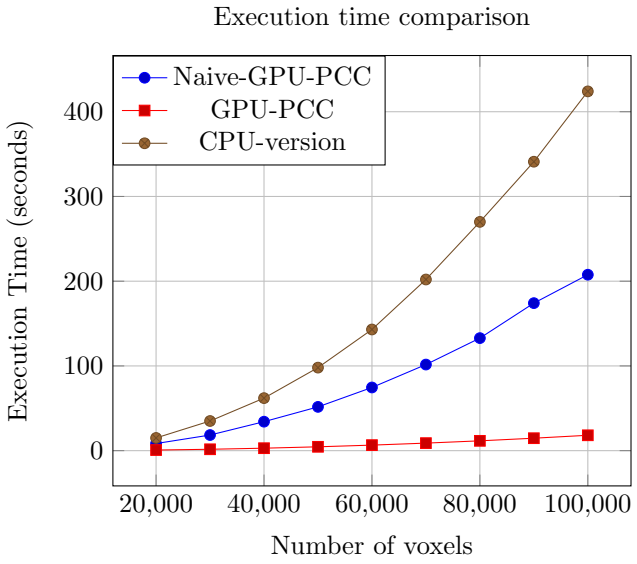

Figure 10: Running time comparison of GPU-PCC with naive GPU-based and CPU-version on synthetic datasets

Table 1: Running time comparison of different techniques on real fMRI data

GPU-PCC	**Naive-GPU-PCC**	**CPU-version**
20.83	279.99	577

Conclusion

In this paper we have presented GPU-DAEMON, a design template for implementing high performance, memory efficient GPU based algorithms for big

omics data. We have demonstrated the capability of GPU-DAEMON by reviewing three algorithms which were implemented separately using this design template. Our experiments have shown that GPU-DAEMON based implementation provides more than 50x speed up over naive GPU based implementation along with conserving already scarce GPU in-core memory.

GPU-DAEMON is a leap forward towards fast development and deployment of high performance GPU based algorithms in the fields of bioinformatics and connectomics.

Acknowledgement

This research was supported by the National Institute Of General Medical Sciences (NIGMS) of the National Institutes of Health (NIH) under Award Number R15GM120820. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health. Fahad Saeed was additionally supported by National Science Foundation grants NSF CAREER ACI-1651724. We would also like to acknowledge the donation of a K-40c Tesla GPU from NVIDIA which was used for all GPU based experiments performed in this paper.

References

[1] Abuín, J.M., Pichel, J.C., Pena, T.F., Amigo, J., 2016. Sparkbwa: speeding up the alignment of high-throughput dna sequencing data. PloS one 11, e0155461.

[2] Awan, M.G., Saeed, F., 2016a. Gpu-arraysort: A parallel, in-place algorithm for sorting large number of arrays, in: Parallel Processing Workshops (ICPPW), 2016 45th International Conference on Parallel Processing, IEEE. pp. 78–87.

[3] Awan, M.G., Saeed, F., 2016b. Ms-reduce: an ultrafast technique for reduction of big mass spectrometry data for high-throughput processing. Bioinformatics 32, 1518–1526.

[4] Awan, M.G., 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.

[5] Baskaran, M.M., Bordawekar, R., 2008. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. IBM Reserach Report, RC24704 (W0812-047) .

[6] Baumgardner, L.A., Shanmugam, A.K., Lam, H., Eng, J.K., Martin, D.B., 2011. Fast parallel tandem mass spectral library searching using gpu hardware acceleration. Journal of proteome research 10, 2882–2888.

[7] Dancik, V., Addona, T.A., Clauser, K.R., Vath, J.E., Pevzner, P.A., 1999. De novo peptide sequencing via tandem mass spectrometry. Journal of Computational Biology 6, 327–342.

[8] Eng, J.K., McCormack, A.L., Yates, J.R., 1994. 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 5.

[9] Eslami, T., Awan, M.G., Saeed, F., 2017. Gpu-pcc: A gpu based technique to compute pairwise pearson's correlation coefficients for big fmri data, in: Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, ACM. pp. 723–728.

[10] Fatahalian, K., Sugerman, J., Hanrahan, P., 2004a. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM. pp. 133–137.

[11] Fatahalian, K., Sugerman, J., Hanrahan, P., 2004b. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM. pp. 133–137.

[12] Jagtap, P., Goslinga, J., Kooren, J.A., McGowan, T., Wroblewski, M.S., Seymour, S.L., Griffin, T.J., 2013. A two-step database search method improves sensitivity in peptide sequence matches for metaproteomics and proteogenomics studies. Proteomics 13, 1352–1357.

[13] Kong, A.T., Leprevost, F.V., Avtonomov, D.M., Mellacheruvu, D., Nesvizhskii, A.I., 2017. Msfragger: ultrafast and comprehensive peptide identification in mass spectrometry-based proteomics. Nature Methods 14, 513–520.

[14] Lee, J.Y., Fujimoto, G.M., Wilson, R., Wiley, H.S., Payne, S.H., 2017. Blazing signature filter: a library for fast pairwise similarity comparisons. bioRxiv , 162750.

[15] Lin, C.H., Li, J.C., Liu, C.H., Chang, S.C., 2017. Perfect hashing based parallel algorithms for multiple string matching on graphic processing units. IEEE Transactions on Parallel and Distributed Systems .

[16] Lindholm, E., Nickolls, J., Oberman, S., Montrym, J., 2008. Nvidia tesla: A unified graphics and computing architecture. IEEE micro 28.

[17] 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.

[18] Ma, Y., Chen, L., Liu, P., Lu, K., 2016. Parallel programing templates for remote sensing image processing on gpu architectures: design and implementation. Computing 98, 7–33.

[19] Mujezinovic, N., Schneider, G., Wildpaner, M., Mechtler, K., Eisenhaber, F., 2010. Reducing the haystack to find the needle: improved protein identification after fast elimination of non-interpretable peptide ms/ms spectra and noise reduction. BMC genomics 11, S13.

[20] Nickolls, J., Buck, I., Garland, M., Skadron, K., 2008. Scalable parallel programming with cuda. Queue 6, 40–53.

[21] Nvidia, 2016. CUDA Toolkit Documentation v7.5. URL: `http://docs.nvidia.com/cuda/index.html#axzz42Wi4k0Qc`.

[22] Nvidia, 2018. CUDA Toolkit Documentation. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[23] Saeed, F., 2015. Big data proteogenomics and high performance computing: Challenges and opportunities, in: Signal and Information Processing (GlobalSIP), 2015 IEEE Global Conference on, IEEE. pp. 141–145.

[24] Saeed, F., Hoffert, J.D., Knepper, M.A., 2013. Cams-rs: Clustering algorithm for large-scale mass spectrometry data using restricted search space and intelligent random sampling. IEEE/ACM transactions on computational biology and bioinformatics 11, 128–141.

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

[26] Tariq, U., Cheema, U., Saeed, F., 2017. Power-efficient and highly scalable parallel graph sampling using fpgas .

[27] Warris, S., Yalcin, F., Jackson, K.J., Nap, J.P., 2015. Flexible, fast and accurate sequence alignment profiling on gpgpu with paswas. PloS one 10, e0122524.