Chapter 6
A Easy to Use Generalized Template to Support Development of GPU Algorithms

Fahad Saeed and Muhammad Haseeb

Computational techniques have taken a new meaning for scientific inquiry in biology especially after the introduction of high-throughput experimental techniques. These instruments can produce massive amounts of data that needs to be processed in a scalable fashion to ensure that we can make sense of these data sets from various sources [2, 3]. As expected, Mass Spectrometry (MS) based omics is essential for precision medicine, cancer research, and drug discovery but the scale at which these data sets needs to be processed is massive (tera- to peta-byte levels) [2–4]. We have also shown that proteomics, and meta-proteomics search can taken impractically long times [5, 6]. which can become a major technical hurdle in investigating these systems biology studies. The existing serial algorithms scale very poorly with increasing size of the data sets, and HPC methods are also shown to be much less than optimal [2, 7].

The post-Moore era of computer architectures has given us ubiquitous access to multicore, manycore, CPU-FPGA, and CPU-GPU architectures which can be used for acceleration of applications [8, 9]. However, up until recently there has not been a serious effort toward developing high-performance computing algorithms for MS-based omics. Likewise, any underlying high-performance computing building blocks [8, 9] that are domain specific had not been built to date. However, it is clear that high-performance computing architectures can, and must, be used for accelerating the processing of big MS-based omics data; something that has been successful in so many other domains [10, 11].

One exciting development in computer architecture is the development of Graphics Processing Unit (GPU) which is a low-cost device but is capable of housing thousands of small computational cores which can be used for exploitation of parallelism in many scientific workflows [12, 13]. However, the limitation of GPU-based computing is that the parallel algorithm has to be designed specifically to exploit GPU

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

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

F. Saeed and M. Haseeb, *High-Performance Algorithms for Mass Spectrometry-Based Omics*, Computational Biology 34,
https://doi.org/10.1007/978-3-031-01960-9_6

architecture for accelerating the code. Since the design of the algorithms is application specific, many of the GPU-based algorithms do not have re-usable designs that can be used for MS-based omics. Other limitations include the development of parallel code without taking into account all of the factors that can affect the speedups and scalability (i.e. inserting parallel programming pragmas in code in the hope to get some speedups) leading to plethora of poorly designed parallel algorithm that can exploit the GPU for MS-based omics [8].

When we started this research, we did not just wanted to have another GPU-based algorithms for MS-based omics. We wanted to develop the fundamental building blocks for GPU-based MS omics data analysis that can be used as fundamental guidelines, and generic principles which would give speedups for many workflows. Such generic goalposts would allow MS domain scientists to develop GPU-based algorithm, which would scale, without worrying too much about complexities of GPU architectures (something that may not be possible for domain scientists to pick just because it is a whole other field of study).

To facilitate the development of GPU-based MS omics workflows; we designed, and implemented a generic GPU-based algorithmic design templates called GPU-DAEMON (GPU Algorithm **D**esign, D**a**ta **M**anagement and **O**ptimizatio**n**). GPU-DAEMON is a GPU-based template that allows exploitation of GPU architecture for any data that looks like large number of very small arrays. Of course, it is designed and implemented with MS data analysis in mind, but we showed in our paper that it is very much applicable to other domains (such a fMRI-based neuroscience) analysis. To design GPU-DAEMON we considered all possible bottlenecks, template design for efficient data management of array structures, and various optimization that allows maximal occupancy, and performance for the GPU-based cores. Once we have introduced the design of GPU-DAEMON, we will implement a GPU-based MS-REDUCE algorithm in the next chapter.

6.1 GPU Architecture and CUDA

Graphical processing units (GPU) were developed to improve the graphics (and related) quality for gaming. However, computational scientists have worked toward using this architecture for general purpose computing to improve the scalability of the scientific workflows. GPU consists of large number of cores that can process in parallel, and can be potentially used for processing individual elements of an image matrix, or matrix calculations to exploit the massively parallel cores available in a typical GPU [14]. A typical GPU consists of several *Streaming Multiprocessors (SM)* each of which contains several CUDA cores which can vary from one GPU model to another. For example, GTX 1080Ti GPU contains 28 SMs with 128 CUDA cores each where as K-40 Tesla GPU contains 15 streaming multiprocessors with 192 core each making a total of 2880 cores.

Irrespective of how many SMs or CUDA cores a GPU has; each SM in a GPU has a fast on-chip memory associated with it and shared among its cores. This fast

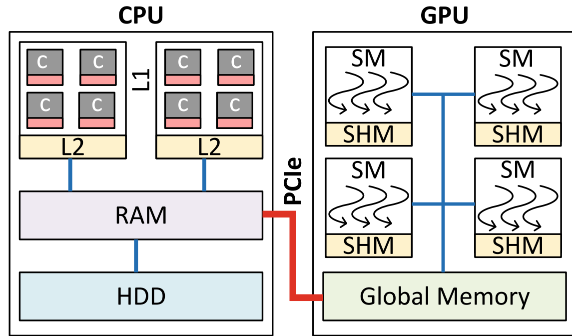

Fig. 6.1 Figure showing CPU-GPU architecture overview. All the data transfers happen via PCIe

on-chip memory is at least 100x times faster than the GPU global memory but is small in size (32kbyte or 64kbyte) [15]. There is also an off-chip memory, known as Global Memory, which is much larger in size (several GB is not uncommon in new GPUs), and is mostly used for storing data, and communicating it with the host CPU. A generic overview of the CPU-GPU architecture is shown in Fig. 6.1.

6.1.1 CUDA Overview

Interest in designing and implementing general purpose computing on GPUs made NVIDIA introduce CUDA standard which can be used with multiple languages to program GPUs [16]. CUDA uses SIMT (Single Instruction Multiple Thread) model which combined the SIMD (Single Instruction Multiple Data) with the assumption of multiple threads and allows exploitation of two levels of parallelism [17, 18]. CUDA standard forms a software overlay that could allow the programmer easy access to parallel architectural features of a GPU. In CUDA programming model, each compute unit is arranged in the form of a Grid of Blocks each containing several threads, where the number of threads, and blocks are GPU dependent. Each thread within a block is assigned 2 IDs, i.e. Threads ID, and Block ID which can be used to track and use each thread in each block. The SMs that are shown in Fig. 6.1 would be replaced with blocks and CUDA cores with threads. As discussed earlier, the number of threads that are active at any given time is dependent on the GPU card that is used [16].

6.1.2 CPU-GPU Computing

For any CPU-GPU computing, one needs a CPU that can act as a host which can offload tasks and data to the GPU which usually behaves like a co-processor. Data from RAM associated with CPU is transferred to the GPU's global memory via

PCIe bus with a set of CUDA instructions. CUDA *kernel* is executed on the GPU where each CUDA core complete the instructions independently. Once the kernel has completed the instructions, and calculation on a given data fragment, the results are transferred back to the host again via PCIe bus. Since PCIe bus is a bottleneck when processing big data, one has to design GPU-based algorithm by having a good understanding (and profiling) of the algorithm [19]. To ensure that scalable processing can happen only most compute intense parts are transferred to the GPU for processing. If enough effort is not invested in the design of the algorithm; the CPU-GPU code will perform poorly when compared to a single one threaded CPU [14]. In the next section we are going to discuss different challenges, and their solutions that one has to consider when design GPU-based algorithms for MS-based omics.

6.2 Challenges in GPU Algorithm Design

This section is dedicated to discuss different challenges, bottleneck, and their solutions when designed GPU-based algorithms for MS-based omics.

6.2.1 Need for Data Parallel Design

GPU compute nodes are generally large in number, and are simple cores without deep pipelines or complex architecture-specific optimizations. Therefore, the best way to exploit GPU cores is to be able to design a parallel strategy that can exploit the data parallelism, and can execute tasks in a parallel fashion without any communication between different cores.

6.2.2 Data Transfer Bottlenecks

Part of the algorithm which is offloaded to the GPU for processing requires that the data is present in the memory of the CPU before the kernel is launched. Since this transfer of data needs to take place via PCIe bus; it is one of the biggest bottlenecks that are faced for big data applications. Needless to say, like any parallel algorithms, if the communication time (to transfer the data) is larger than the time it takes to compute the data; then the scalability of the implementation is limited. In the same way, if the data that is transferred from the GPU to the CPU is larger than the available memory; again efficient data transfer techniques are needed to eliminate or reduce CPU-GPU bottlenecks.

6.2.3 Non-coalesced Memory Accesses

GPU threads that are active are grouped into 32 thread-chunks and are known as *warp* which are scheduled on to the SMs when they are available. These wraps can then be mapped to each of the SMs as the resources become available. Global memory accesses from the threads of a wrap can be combined together (colloquially known as coalesced together) to the same memory transaction if the locations have spatial locality . If spatial locality is not present, then accessing the global memory in multiple transactions leads to stalling the wrap execution. This specific issue can become a major bottleneck if the parallel method that can exploit a GPU is not carefully designed.

6.2.4 Warp Divergence

In SIMT execution, the thread in a wrap execute in a lock-step which allows all the independent instruction to be executed simultaneously. Branches on the other hand can lead the threads to diverge which can result in efficiency loss, and minimization of this warp-divergence is one of the challenges, and major design decision for parallel algorithms that can exploit GPUs.

6.2.5 Exploiting Coarse Grained and Fine Grained Parallelism

Since GPUs require two-levels of parallelism, each level would need fine-grained data management methods to exploit the parallelism that might be available in the architecture. Most often this will require that the data is managed in a way that decomposes the data in fine-grained sets so that they can be processed in parallel. If the decomposition is not fine-grained enough, the amount of parallelism that can be exploited using GPU would be under utilized leading to less scalable solutions.

6.3 Basic Principles of GPU-DAEMON

The proposed GPU base template provides a design that can be used for CPU-GPU-based algorithms for big data omics especially for MS-based omics, Next Generation Sequencing (NGS) based genomics, and fMRI based connectomics. For our purposes, we will focus on the design principles that are relevant to MS-based omics data analysis. The design of our proposed GPU-DAEMON is divided into seven steps where each step gives a generic solution that tackles one or more GPU bottleneck. Of

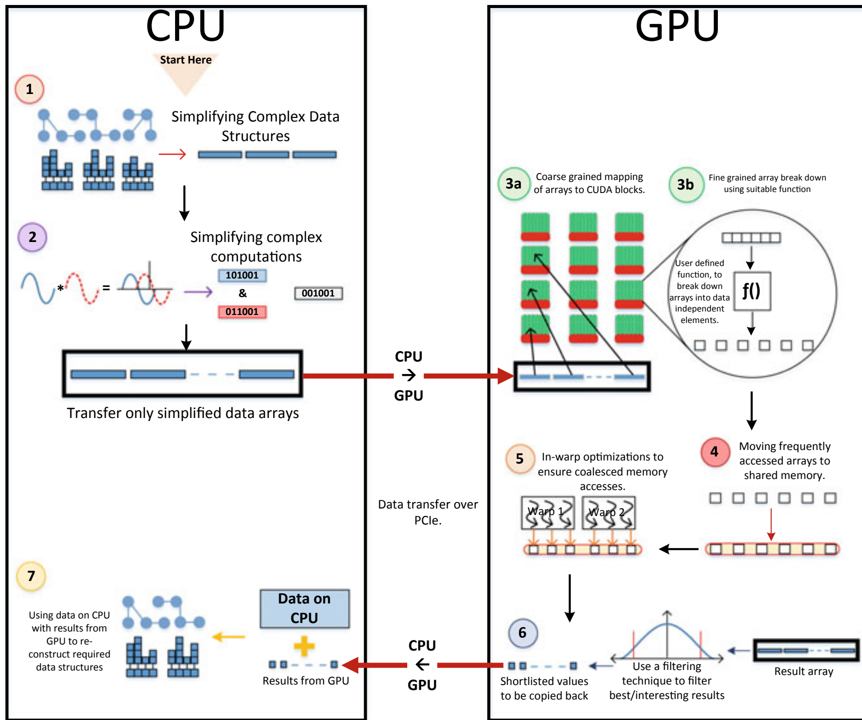

Fig. 6.2 Figure shows the template for GPU-DAEMON

course, these solutions need to be adapted to the application that is being considered but give a good starting point to modify depending on the problem being solved. Figure 6.2 shows the steps in GPU-DAEMON. The first step is to analyze and profile the algorithm for which the GPU-based method is being developed. This step will determine if the proposed method is compute- or data-intensive. Any compute- or data-intensive parts are kept for the GPU-side while other simpler operations may be completed on a CPU.

6.3.1 Simplifying Complex Data Structures

PCIe bus that is mostly used to transfer the large data structures is one of the first bottlenecks that are encountered. Our proposed approach of transferring the complete data structures is easy and intuitive for the programmer, and generally does not require the complete redesign of the parallel code. With limited memory compute area in the GPU dictates that only a small portion of the data structure is needed for computation at any given point in time. Therefore, a design methodology that requires one to only

transfer part of the data structure will considerably reduce the bottleneck that is associated with PCIe bus.

6.3.2 Simplifying Complex Computations

In general, existing GPU architectures are simple computing platforms and do not allow complex computations which makes them specialized for large number of computations. To this end, the second step of the GPU-DAEMON is to simplify the complex computations, e.g. by converting floating point numbers into integers, or representing those data sets as binary making data computations simpler for these cores. Of course, these simplifications are application specific, and not all computations or data can be made simpler especially in the scenario where precision is required.

6.3.3 Efficient Array Management in GPU

From the data management prospective, CPU-GPU strategy is dependent on how the arrays of the data are managed, and how different CUDA compute nodes compete for access to data; and how this data is accessed can determine the performance of the GPU-based parallel algorithm. To exploit the multiple levels of parallelism that are available, we introduce and evaluate fragmentation strategy that can be accomplished using two steps: (1) First step consists of ensuring that each array can be mapped to a unique block in the GPU in a coarse-grained fashion. This can almost always be accomplished since the number of CUDA blacks are much larger than the number of arrays.

(2) The second step is used to exploit fine-grained parallelism by decomposing each array into sub-arrays and then mapping them to a cluster of threads. Since this is more specific to the application, the data mapping can be categorized into two parts each with a different approach. If the data calculations are independent and are not dependent on the calculation from other arrays cells then an array of size m, following number of elements assigned per-thread should suffice:

$$E_i = \frac{m}{nT} \tag{6.1}$$

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

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 sub-array 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 sub-arrays assigned to thread i, respectively.

When the data is dependent on each other for calculations, then elements need to be divided into data independent subsets using a suitable user defined function as shown in GPU-DAEMON template Fig. 6.2. We denote this function by F_{sub}.

6.3.4 Exploiting Shared Memory

Shared memory that is available on a GPU is 100x times faster than any other memory, which makes exploitation of this memory module most consequential for the performance of the code. The programmer would want to make sure that the frequently accessed part of the calculations is moved to the shared memory. One condition [15, 19] that would ensure that such move will give reasonable speed advantage the following equation must hold:

$$(T_{tf}) + (P_{SM}) < (P_{GM}) \tag{6.2}$$

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.

6.3.5 In-Warp Optimizations

Optimization strategies which will ensure that we get the best performance from a GPU; we want to ensure that thread divergence inside a warp, and memory coalescing to access global memory are reduced to minimize loss in performance. For thread divergence, usually the parallel algorithm needs to be redesigned (or at least reconfigured) so that the threads needed for computations do not diverge in a warp. Global memory coalescing can be achieved by good thread to data mapping strategy. The mappings discussed in third step of GPU-DAEMON simplify this mapping. By mapping consecutive threads to independent contiguous array segments of Step 3 can help achieve memory coalescing.

6.3.6 Result Sifting

Once computations have been completed, the programmer has to ensure that the output is also managed correctly. Sometimes these output arrays can be larger than the input data [20], and attention has to be paid to ensure that memory transfer bottlenecks are not formed. Usual techniques include either compressing the output results or copying back just the relevant part of the calculations. Since these are very specific to the application under consideration, we will not generalize this for GPU-DAEMON.

6.3.7 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.

6.3.8 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 disintegrate 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 disintegration 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

$$T_{GPU} = \frac{N}{B * p} * (T(f_{sub}) + T(f_{proc}) + x * T(f_{sift})) \tag{6.3}$$

References

1. Awan MG, Eslami T, Saeed F (2018) Gpu-daemon: Gpu algorithm design, data management and optimization template for array based big omics data. Comput Biol Med 101:163–173
2. Abuín JM, Pichel JC, Pena TF, Amigo J (2016) Sparkbwa: speeding up the alignment of high-throughput dna sequencing data. PloS one 11(5):e0155461
3. 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
4. Saeed F, Hoffert JD, Knepper MA (2013) Cams-rs: clustering algorithm for large-scale mass spectrometry data using restricted search space and intelligent random sampling. IEEE/ACM Trans Comput Biol Bioinform 11(1):128–141
5. Kong AT, Leprevost FV, Avtonomov DM, Mellacheruvu D, Nesvizhskii AI (2017) Msfragger: ultrafast and comprehensive peptide identification in mass spectrometry-based proteomics. Nat Methods 14(5):513–520
6. Jagtap P, Goslinga J, Kooren JA, McGowan T, Wroblewski MS, Seymour SL, Griffin TJ (2013) A two-step database search method improves sensitivity in peptide sequence matches for metaproteomics and proteogenomics studies. Proteomics 13(8):1352–1357
7. Saeed F (2015) Big data proteogenomics and high performance computing: challenges and opportunities. In: 2015 IEEE global conference on signal and information processing (GlobalSIP). IEEE, pp 141–145
8. Tariq U, Cheema UI, Saeed F (2017) Power-efficient and highly scalable parallel graph sampling using fpgas. In: 2017 international conference on ReConFigurable computing and FPGAs (ReConFig). IEEE, pp 1–6
9. Eslami T, Awan MG, 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. 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. In: IEEE transactions on parallel and distributed systems
11. 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(1–2):7–33
12. Warris S, Yalcin F, Jackson KJ, Nap JP (2015) Flexible, fast and accurate sequence alignment profiling on gpgpu with paswas. PloS one 10(4):e0122524
13. Baumgardner LA, Shanmugam AK, Lam H, Eng JK, Martin DB (2011) Fast parallel tandem mass spectral library searching using gpu hardware acceleration. J Proteome Res 10(6):2882–2888
14. Fatahalian K, Sugerman J, Hanrahan P (2004) 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
15. 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
16. Nvidia (2016). http://docs.nvidia.com/cuda/index.htmlCUDA Toolkit Documentation v7.5. http://docs.nvidia.com/cuda/index.html
17. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with cuda. Queue 6(2):40–53
18. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) Nvidia tesla: a unified graphics and computing architecture. IEEE Micro 28(2)

19. 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
20. Lee J-Y, Fujimoto GM, Wilson R, Wiley HS, Payne SH (2017) Blazing signature filter: a library for fast pairwise similarity comparisons. bioRxiv 162750