Efficient 2-Body Statistics Computation on GPUs: Parallelization & Beyond

Napath Pitaksirianan, Zhila Nouri, and Yi-Cheng Tu
Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Ave. ENB118, Tampa, FL, 33620, U.S.A.
Email: {napath, zhila, tuy}@mail.usf.edu

***Abstract*—Various types of two-body statistics (2-BS) are regarded as essential components of data analysis in many scientific and computing domains. Due to the quadratic time complexity, use of modern parallel hardware has become an obvious direction for research and practice in 2-BS computation. This paper presents our recent work in designing and optimizing parallel algorithms for 2-BS computation on Graphics Processing Units (GPUs). First, we classify 2-body applications into three groups based on their data output pattern. Then, we introduce a straightforward parallel algorithm under the CUDA framework. The unique architecture of modern GPUs, however, provides abundant opportunities for optimizing the algorithm. To that end, we split the algorithm into two stages: pairwise distance function computation and writing output. Then, we present modifications to the basic algorithm by integrating various techniques at each stage. Since the architecture of modern GPUs is much more complex than that of multi-core CPUs, traditional wisdom on decomposing problems in a parallel platform is often insufficient in developing GPU-based algorithms. Therefore, our algorithms design focuses on effective use of hardware/software features that are unique in GPU platforms. In addition to the various programming cache and atomic operations, we also introduce novel load balancing and register content sharing techniques. We develop models to analyze such techniques and identify the best ones for each type of 2-BS. Experiments run on modern GPU hardware show that our GPU algorithms outperform the best known CPU program by at least an order of magnitude in various applications. Furthermore, our implementation achieves very high level of GPU resource utilization, indicating near-optimal performance. This work builds a solid foundation towards realizing our vision of a framework that can automatically generate optimized code for any new 2-BS problems.**

I. Introduction

Various types of 2-body statistics are essential components of data analysis in many scientific domains. Bearing many forms and definitions, 2-body statistics as we refer to in this paper, is a group of statistical measurements that evaluate all pairs of datum among an N-point data set. There are numerous examples of 2-body statistics [1]: 2-tuple problem [1], all-point nearest-neighbor classification [1], nonparametric outlier detection and denoising [1], kernel density regression [1], relational joins [2], two-point angular correlation function [3], 2-point correlation function [1], Radial distribution function (RDF) [4], and spatial distance histogram (SDH) [5] to name a few. n this paper, we simply use the word "2-body statistics" (2-BS) to cover all of them.

In general, a 2-BS can be computed by solving a function between all pairs of datum. Such a function often demand constant time to compute, and for convenience of presentation, let us call them *distance* functions. The worse-case quadratic time complexity can be a big obstacle to the successful deployment of applications that depend on the computation of such statistics. The first line of defense against this is obviously better algorithms with lower complexity. For example, our previous work used quad-tree and batching technique to reduce complexity of SDH computing to $O\big(N^{1.5}\big)$ [5]. Similar techniques are proposed in [6]. On the other hand, parallel computing techniques can be utilized to speed up the computation in practice, and is the focus of our study reported in this paper. In the context of 2-BS, parallel computing techniques are extremely useful for two reasons: (1) particular types of 2-BS lack efficient algorithms. For example, kernel functions for Support Vector Machine (SVM) [7], Pairwise comparison in various applications [8], [9] can only be solved in quadratic time; (2) performance of more advanced algorithms can be further improved via parallelization. Related to the latter, it is interesting to see that such algorithms share common computational primitives with the quadratic algorithms therefore they can be put into the same parallel computing framework. In fact, the vision of our work is to develop such a framework for computing a large group of problems that show similar data access and computational features as those found in typical 2-BSs. In this framework, we will implement core computational units that are found in 2-BSs and optimize them towards running on modern many-core hardware systems. That said, the scope of this paper is set to the various techniques to implement and optimize the computational primitives needed for computing 2-BSs on modern Graphics Processing Units (GPUs).

With massive computing power and high-speed memory, GPUs have become a part of many high-performance computing (HPC) systems. Originally designed for graphics processing, the popularity of general-purpose computing on GPUs (GPGPU) has boosted in recent years with the development of software frameworks such as Compute unified device architecture (CUDA) [10] and Open Computing Language (OpenCL) [11]. Due to the compute-intensive nature of 2-BS problems and the fact that the main body of computations can be done in parallel for most 2-BSs, GPUs stand out as desirable platform

for implementing 2-BS algorithms.

GPU algorithms for several 2-BS problems have been studied in the past few years. For example, Levine *et al*. parallelized algorithm for solving a special form of RDF, which utilizes constant memory for tiling input data [4]. Another paper briefly sketched techniques for speeding up two-point angular correlation function computation [3]. In addition to the surprisingly little attention paid to this topic, existing work lacks a comprehensive study of the possible techniques to achieve maximum performance on GPUs. This is a non-trivial task because the architecture of modern GPUs is much more complex than that of multi-core CPUs. As a result, traditional wisdom on decomposing problems in a parallel platform is often insufficient. Although the 2-BS problems we consider share the same core computations, each 2-BS problem however carries its own characteristics that calls for different strategies in code optimization.

In this paper, we present a series of techniques to decompose 2-BS problems and methods for effective use of computing resources on GPUs. We identify two phases in computing typical 2-BS problems: a pairwise data processing phase and result outputting phase. In the first phase, we focus on tiling methods and use of different types of GPU cache to reduce data access latency on global memory. For the output phase, we focus on privatization and summation of output to reduce race condition and use of atomic operations. A major contribution of this work is that, our algorithms, while following general parallel computing strategies, effectively use hardware/software features that are unique in GPU platforms. In addition to the various programming cache and atomic operations, we also introduce novel load balancing and register content sharing techniques. Extensive experiments show that our GPU algorithms outperform the best known CPU program by at least an order of magnitude in various applications. Furthermore, our program achieves very high level of GPU resource utilization, indicating near-optimal performance. This work is a first step towards our long-term goal: to combine these techniques into a framework that can automatically generate optimized code for any new 2-BS problems.

This paper is organized as follows: in Section II, we review work related to 2-BC problems; in Section III, we brief background of our study; in Section IV, we demonstrate techniques to speed up pairwise computation and writing output on GPUs; we conclude this paper in Section V.

II. Related Work

There are many applications of 2-BS problems. In addition to those mentioned in Section I, there are a lot of applications that use various distance measures (e.g., *Euclidean*, *Jaccard*, and *cosine distance*) to find the closeness of all pairs of input datum. One important example is the recommendation systems for online advertising that predicts the interest of customers and suggests correct items. Jensen *et al.* reports a music predictive model [12] based on pairwise comparisons of Gaussion process priors between musics. There are two types of recommendation systems: content-based filtering (CB) and collaborative filtering (CF) [8], [9]. Both require 2-BS computation: CB depends on pairwise comparisons between items and CF on those between users.

There are a number of reports on lowering complexity of 2-BS problems. For example, the state-of-art SDH algorithm works on particle counts in nodes of a tree-based structure [5], [13], and reduces complexity to $\theta(N^{3/2})$ for 2D data and $\theta(N^{5/3})$ for 3D data. The basic idea is to conduct pairwise comparisons of tree nodes (instead of individual particles). Therefore, the core procedure of pairwise comparison as well as the strategy to parallelize the algorithm remains the same.

The past few years witnessed a strong movement of using GPGPU for solving scientific computing problems and numerous reports on such are generated each year. However, to the best of our knowledge, there are few reports on computing 2-BSs on GPUs. As a part of efforts to parallelize relational joins, He *et al.* implemented a nested-loop join algorithm on GPUs and reported a 7X speedup over CPUs [2]. Similar results are presented in [14]. Levine *et al.* [4] studied GPU-based processing of RDF, of which the main task is to compute a histogram of all point-to-point distance. They used data privatization techniques to speed up the algorithm. More recently, Stratton *et al.* sketched tiling and privatization techniques in computing two-point angular correlation function [3], yet no technical details are reported. Unlike the focus of individual problems and techniques seen in above work, this paper is about a comprehensive study of the multitude of techniques that can be used for the development and optimization of GPU-based 2-BS algorithms.

III. Background

A. GPU Architecture and CUDA

In this section, we briefly introduce the architecture of modern GPUs. We use the latest generation of NVidia GPU product as an example. We believe such information is essential in our discussions of (parallel) algorithm design in the remainder of this paper. Readers already familiar with GPU architecture can skip this section.

A GPU contains many processing units (cores) for handling complex graphics computing. A group of cores is organized into a *multiprocessor* and a GPU can have up to 16 multiprocessors. A GPU contains a few GBs of *global memory* that uses DDR5 technology. Host can transfer data to the global memory via direct memory access (DMA) over PCI-E link. Transferring data from global memory to multiprocessors is much faster. Global memory can be accessed by different multiprocessors simultaneously at a bandwidth up to 224 GB/sec [15]. Each multiprocessor also provides high-speed programmable *shared memory* of size 96KB. The use of shared memory is under full control of the programmer. There are also the programmable *read-only data cache* in each multiprocessor for holding data that cannot be overwritten during the lifespan of the program, as well as the nonprogrammable L1 cache (within each multiprocessor) and L2 cache (shared by all multiprocessors).

On the software side, the CUDA programming model allows a large number of threads to be launched to compute a function (called *kernel*). The entire collection of threads (named *grid*) are organized into groups (called *blocks*), therefore each thread can be identified by a block ID and thread ID within the block. In the CUDA runtime environment, all threads in a block will be executed in the same multiprocessor. On the other hand, one multiprocessor can execute multiple blocks. However, only a small number of threads (called a *warp*) can be executed at the same time. Each warp contains 32 threads with consecutive thread IDs in recent NVidia products. In a warp, each thread has its own registers, and the threads are executed in a single-instruction-multiple-data (SIMD) manner. Divergence is the situation when threads in the same warp have different execution paths thus should be avoided if possible.

Over the years, GPU architectures from NVidia has evolved through several generations: Fermi [16], Kepler [17] and Maxwell [15]. Newer architectures provide more computing resources. Moreover, along with the newer architectures, there are also new functions and features in the CUDA framework. For example, starting from Kepler, *shuffle instructions* can be used to exchange data in registers among threads in the same warp. Such new features encompass additional opportunities for improving program efficiency.

B. Computing 2-body statistics in GPUs

A straightforward GPU algorithm for computing 2-BS is shown as Algorithm 1. Note the pseudocode is written from the perspective of a single thread, reflecting the Single-Program-Multiple-Data (SPMD) programming style of CUDA. Each thread loads one datum to a local variable, and uses that to loop through the entire input dataset for the distance function computation. The output will be updated with the results of each distance function computation.

To optimize the above 2-BS algorithm, the main challenges are in dealing with the input and output data, respectively. First, each input datum will be read many times into register for the distance function computation. Therefore, the strategy is to push the input data into the cache as much as we can. The many types of cache in GPUs, however, impose challenges in program development. Second, every thread needs to read and update the output data at the same time. Updating the output data simultaneously might cause incorrect results. Some work have proposed strategies to avoid race condition in multiprocessor [18]. However, their method is limited by the number of threads and number of bins in the histogram, because each thread needs to have a private output in shared memory. Recent versions of CUDA provide *atomic instructions* to ensure correctness under race condition. However, an atomic instruction also means sequential access to the protected data thus lowers performance. As a result, the strategy to update output data is still important. We need to avoid update collision as much as possible.

First, there is a need to characterize the multitude of 2-BS cases based on the computational paths. This helps us determine the proper combination of techniques we use for optimizing individual 2-BS problems. We found that the 2-BS we studied are very similar at the point-to-point distance function computation stage. However, members of the 2-BS family tend to have very different patterns in the data output stage. We have identified three groups of 2-BSs based on the output pattern, and will introduce different techniques in dealing with these types in the following sections.

Algorithm 1: Generic GPU-based 2-BS algorithm

```
Local Var: t (Thread id)
1: currentPt ← input[t]
2: for i = t + 1 to N do
3:     d ← DisFunction(currentPt, input[i])
4:     update output with d
5: end for
```

Type-I: members of this group generate a very small amount of output data from each thread. These output must be small enough to be placed in registers for each thread. For example, 2-point correlation function [1], which is fundamental in astrophysics and biology, outputs a number of pairs of points that determine correlation in dataset. Other examples are all-point k-nearest neighbors (when k is small) and Kernel density/regression [1], which output classification results or approximation numbers from regression.

Type-II: the output in this group cannot be placed in registers but are still small enough to be put into GPUs' shared memory. Examples include: (1) Spatial distance histogram (SDH) [5], which outputs a histogram of distances between all pairs of points; (2) Radial distribution function (RDF) [4], which outputs a normalized form of SDH.

Type-III: in this group, the size of the output can be large so they can only be put into global memory. In extreme cases, the size of the output is quadratic to the size of input. Some examples are: (1) relational join [2], which outputs concatenated tuples from two tables - total number of output tuples can be quadratic (especially in non-equality joins) ; (2) Pairwise Statistical Significance [19], which computes pairwise alignment between two datasets and generates quadratic output; and (3) Kernel methods which compute kernel functions for all pairs of data in the feature space [7].

IV. GPU Algorithm Design and Implementation

A. Algorithms for Pairwise Computation Stage

We first present design strategies in the pairwise distance function computation stage. First of all, the input data is stored in the form of multiple arrays of single-dimension values instead of using an array of structures that each holds a multi-dimensional data point. This will ensure coalesced memory access when loading the input data. Note that the naive algorithm loads one input (i.e., tmpPt in Algorithm 1) to the distance function from the global memory, and there are $O\left(N^2\right)$ total distance function calls. Due to the high latency (i.e., about 350 cycles [20], [21]) of data transferring between the global memory and cores, our goal is to reduce the number

TABLE I
SYMBOLS AND NOTATIONS

Symbol	Meaning
$C_{GAtomic}$	latency of using Atomic operation in Global memory
C_{GW}	latency of writing to Global memory
C_{GR}	latency of Reading to Global memory
$C_{ShmAtomic}$	latency of using Atomic operation in Shared memory
C_{ShmW}	latency of Writing to Shared memory
C_{ShmR}	latency of Reading to Shared memory
H_S	Histogram Size or Output Size
N	Number of input datum
B	Block size
M	Number of blocks

of data reads from global memory. In particular, we use the well-known *tiling* method [22] to load data from the global memory to on-chip cache. Whenever two data points are used as inputs to the distance function, they are retrieved from cache instead of the global memory. Symbols and notations used throughout this paper are listed in Table I.

Fig. 1 illustrates the tiling method. We divide input data into small blocks, and the size of a block ensures it can be put into cache (we will discuss scenarios of loading to different types of cache later). Normally, the data block size is the same as the number of threads in each CUDA block. Each thread loads one datum into the cache to ensure coalesced access to the global memory. With blocks of data loaded to cache, the main operation of the algorithm is now to compute distance function between two different blocks of data (*inter-block* computation). Algorithm 2 shows the pseudo code of the tiling-based algorithm. Basically, each thread block first loads an anchor block L, and then loads a series of other blocks R. Specifically, all data blocks with an index higher than that of the current thread block will be loaded one by one as R (line 1). For each R loaded, distance function will be computed between the thread's datum (L[t]) and every datum in R. Note that both inputs to the distance function is read from data blocks that are cached. Obviously, the number of thread blocks we launch should be the same as the data blocks in the entire input data array. In other words, we have

$$\mathrm{M} = \frac{\mathrm{N}}{\mathrm{B}} \tag{1}$$

One other thing is that, each block of threads needs to compute distance functions between all pairs of datum inside the anchor block L (lines 9 to 12 in Algorithm 2). Only through this we can achieve the computation of all pairwise distance functions in the entire dataset. In particular, each thread compares its own datum with every other datum with higher index than its own thread ID (line 9).

To implement the above algorithm, an important decision to make is: *which cache do we use to hold block L and block R?* There is no straightforward answer due to the multiple cache systems in the NVidia GPUs. By ignoring the non-programmable L2 cache, we still have the programmable shared memory and read-only data cache, both have TBps-level bandwidth and response time of just a few clock cycles

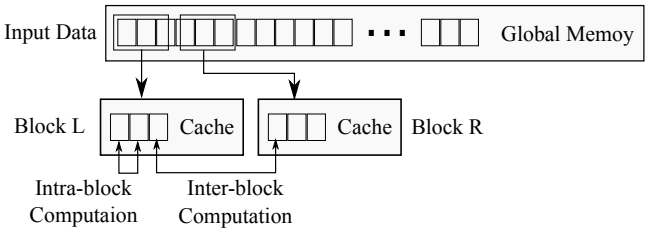

Fig. 1. Tiling method requires loading data in blocks

Algorithm 2: Block-based 2-BS computation

```
Local Var: t (Thread id), b (Block id)
Global Var: B (Block size), M (total number of blocks)
1:  L ← the b-th input data block loaded to cache
2:  for i = b + 1 to M do
3:      R ← the i-th input data block loaded to cache
4:      for j = 0 to B do
5:          d ← DisFunction(L[t], R[j])
6:          update output with d
7:      end for
8:  end for
9:  for i = t + 1 to B do
10:     d ← DisFunction(L[t], L[i])
11:     update output with d
12: end for
```

[20], [21]. According to [20], shared memory has the lowest latency in GPUs (i.e., 28 clock cycles). It is natural for us to use shared memory to hold both blocks L and R, and this can be viewed as a starting point for our discussions.

By taking a closer look at Algorithm 2, we found that each datum will have to be placed into a register before it can be accessed by the distance function anyway. And each thread only accesses a particular datum throughout its lifetime. Therefore, it makes little sense to store L in shared memory first – we are better off by defining a local variable for each data member of block L. By using the *register* modifier in CUDA, such a variable will be stored and accessed in registers. This will reduce the consumption of shared memory in each thread – shared memory is a bottlenecking resource when we consider large data output (Section IV-C). Plus, latency of accessing registers is just one clock cycle [22]. Note that the same argument does not hold true for block R: all data in block R is meant to be accessed by all threads in the block but a register is private to each thread. Therefore, we have to load R into cache. Given that, we introduce the second technique which optimizes the first technique by using registers to hold one datum from block L, and allocating shared memory to hold block R. The program also needs another change to handle the intra-block distance computation (lines 9 to 12 in Algorithm 2): such computation requires threads to have access all data in block L. For that, we now have to load block L to shared memory before we run the last **for** loop. But the trick we play here is: instead of asking for a new chunk of shared memory

Algorithm 3: Block-based 2-BS with optimized input tiling and output

Local Var: t (Thread id), b (Block id)
Global Var: B (Block size), M (total number of blocks)

```
1: SHMOut ← Initialize shared memory to zero
2: reg ← the t-th datum of b-th input data block
3: for i = b + 1 to M do
4:    R ← the i-th input data block loaded to cache
5:    for j = 0 to B do
6:       d ← DisFunction(reg, R[j])
7:       SHMOut[d] ← SHMOut[d] + 1
8:    end for
9: end for
10: L ← the b-th input data block to overwrite R's cache
    location
11: for i = t + 1 to B do
12:    d ← DisFunction(reg, L[i])
13:    SHMOut[d] ← SHMOut[d] + 1
14: end for
15: Output[b][t] ← SHMOut[t]
```

for L, we overwrite the space we just used for block R. By that, the total shared memory used is still one block. Details of such optimizations can be found in Algorithm 3.

We also explore another solution that further relieves the bottleneck of shared memory. Although this solution may not yield higher performance in the distance computation stage, it is meaningful if we have to use shared memory for other demanding operations in the output stage (Section IV-C). This solution basically does not change the code structure of the second solution (with use of registers). However, we use the read-only data cache (also named *texture* memory) instead of shared memory to store blocks R (for inter-block computation) and L (for intra-block computation). Read-only data cache has higher latency than the shared memory [20] (i.e., about 64 clock cycles higher) but it is still an order of magnitude faster than global memory. As a side note about implementation, read-only cache is not fully programmable as the shared memory, but we can use the "*const __restrict__*" keyword combination before a variable to ask CUDA runtime framework to store the variable into the read-only data cache.

B. Evaluation of Pairwise Algorithms

In this section, we present results of analytical and empirical evaluation of the performance of algorithms mentioned above. In particular, we evaluate: (1) SHM-SHM: caching both blocks L and R in shared memory; (2) Register-SHM: caching one datum in register and block R in shared memory; (3) Register-ROC: placing one datum in register and block R in read-only cache; and we also compare with (4) Naive: generic GPU-Based 2-BS algorithm as shown in Algorithm 1.

Analytical Evaluation: In order to robustly compare the performance of proposed algorithms, we present an analytical model of the number of accesses to different types of GPU memories during the execution of these algorithms. Rigorous studies [20], [21] have shown the access latency of global memory, read-only data cache (ROC) and shared memory are 350, 92, and 28 clock cycles, respectively. Moreover, the bandwidth of shared memory is much larger than others (3TB/s vs. 1TB/s for the ROC). The Naive algorithm does not benefit from shared memory and ROC, and just uses global memory. The number of accesses to global memory for this algorithm is:

$$N + \sum_{i=1}^{N-1}(N-i) \tag{2}$$

Since each access costs 350 clock cycles, this algorithm is very costly. However, other improved algorithms take advantage of faster cache and yields better performance. SHM-SHM and Register-SHM both use shared memory plus global memory, and Register-ROC uses data cache and global memory. These three mentioned algorithms have the same number of accesses to global memory, which equals to:

$$N + \sum_{i=1}^{M-1}(M-i)B \tag{3}$$

Having the same number of accesses to global memory, we should consider other memory accesses for the sake of comparison. In SHM-SHM algorithm, number of accesses to shared memory is:

$$2\sum_{i=1}^{M-1}(M-i)B^2 + 2\sum_{i=1}^{B-1}(B-i)M \tag{4}$$

In Register-SHM, number of accesses to shared memory is:

$$\sum_{i=1}^{M-1}(M-i)B^2 + \sum_{i=1}^{B-1}(B-i)M \tag{5}$$

Considering the number of accesses to shared memory by SHM-SHM and Register-SHM, it is obvious that Register-SHM cuts the number of accesses quite considerably, dropping by half, that is why it is faster than SHM-SHM. On the other hand, Register-ROC exploits data cache instead of shared memory and the number of accesses to this memory is the same as the number of accesses of Register-SHM to share memory. Considering the much higher latency and smaller bandwidth of ROC compared to shared memory, it's clear that Register-SHM outperforms the other.

Empirical Evaluation: We implemented all four algorithms in CUDA and experimented using synthetic data with different sizes. In particular, we implemented CUDA kernels to compute a Type-I 2-BS: the 2-point correlation function (2-PCF). The 2-PCF requires computation of all pairwise Euclidean distances and the output is of very small size: one scalar describing the number of points within a radius. We see 2-PCF as a good example here because the work is almost exclusively on the distance computation. We run our experiments in a workstation running Linux (Ubuntu 14.04 LTS) with an Intel Xeon E5-2640 v2 CPU, 64GB of DDR3 1333-MHz memory and an Nvidia Titan X GPU with 12GB of global memory. This workstation is also the platform we use for other experiments reported throughout this paper.

Our input data has size ranging from 512 to 2 million particles. Particle coordinates are generated following a uniform

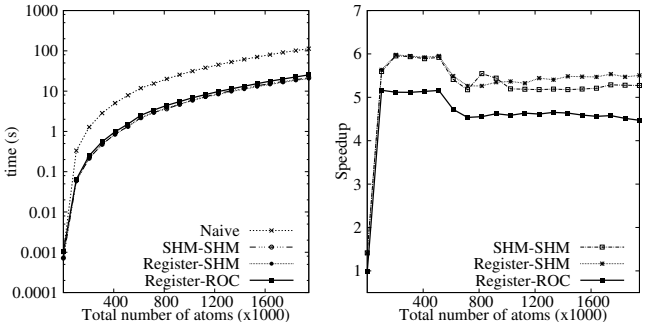

Fig. 2. Performance of different GPU-based algorithms for computing 2-PCF: total running time and speedup over naive algorithm

TABLE II
UTILIZATION OF DIFFERENT GPU RESOURCES IN RUNNING DIFFERENT 2-PCF KERNELS. MEMORY TYPES NOT SHOWN HERE ARE WITH VERY LOW UTILIZATION

Kernel	Arithmetic Operation	Control-flow Operation	Memory
Naive	15%	3%	76% (L2 Cache)
SHM-SHM	50%	7%	35% (Shared Memory)
Reg-SHM	52%	11%	35% (Shared Memory)
Reg-ROC	24%	10%	65% (Data cache)

distribution in a region. For kernel parameters, we set the total number of threads as the data size and the value of threads per block to 1024, which is derived from an optimization model developed in our previous work [23] – that model guarantees best kernel performance among all possible parameters.

Figure 2 shows the total running time of each experimental run. We first observe that the running time grows with data size in a quadratic manner – this is consistent with the $O(N^2)$ complexity of such algorithms. Among all tested parallel algorithms, the Register-SHM kernel has the best performance under all data sizes – it achieves an average speedup of 5.5X (max. speedup of 6X) over the naive algorithm. The SHM-SHM kernel shows similar results, with an average speedup of 5.3X (max. speedup is also 6X). For data size 800K and beyond, Register-SHM consistently shows better performance than SHM-SHM, although with a narrow margin. The Register-ROC kernel shows the least improvement over naive algorithm, with an average speedup of 4.7X and maximum speedup of 5X. The above results are clearly in conformity with our understanding of the proposed caching solutions.

To evaluate the level of optimization we achieved in our solutions, we looked into the resource utilizations in the GPU while running our kernels. Normally, the bottleneck is on the memory bandwidth in processing 2-BSs such as the 2-PCF, due to the simple calculations in the distance function. If we feed the cores with sufficient data, the cores show a high utilization, and that is a good indication that the code is highly optimized. Another way to look at this is: since the total number of distance function calls is the same for all solutions, the less idling time the cores experience, the better performance the algorithm has. Information related to resource utilization can be obtained by running the program through the NVidia visual profiler, a tool for analyzing runtime characteristics of CUDA kernels. Table II shows utilization of different hardware units as recorded by the profiler. Clearly, the three cache-based techniques significantly increases utilization of compute core resources as compared to naive algorithm. The Register-SHM and SHM-SHM kernels both achieve over 50% utilization of arithmetic units. This is achieved at around 35% of shared memory bandwidth utilization. While the shared memory has a utilization of 35%, it is by no means saturated. Thus, as the profiler suggests, both kernels are compute bound. The Register-ROC kernel achieves a 24% arithmetic unit utilization, verifying the result that its performance is not as good as the other two. Without a surprise, it reaches a high utilization (65%) of read-only cache bandwidth.

C. Data Output Stage

In this section, we present techniques to efficiently output the results from GPUs in 2-BS computing. Depending on the features of data output, the design strategy on this stage can be different for various 2-BSs. The simplest type is that each thread omits a very small amount of output (e.g., Type-I 2-BS) – we simply use local variable(s) to store an active copy of the output data in registers, and transmit such data back to host when kernel exits. For problems with very large output size (e.g., Type-III 2-BSs), we have to output results directly to global memory. The main problem for using global memory for output is the race condition caused by different threads' writing into the same memory location – a *gather* operation in parallel computing terminology. To avoid incorrect results caused by race condition, atomic instructions are used in GPUs to have protected access to (global) memory locations. In CUDA, such protected memory location is not cached and obviously cannot be accessed in a parallel way. Therefore, it renders very high performance penalty to use atomic instructions.

We adopt a *privatization* technique to reduce race condition. In particular, we store private copies of the output data to be used by a subset of the threads in the shared memory. This solution is suitable for Type-II 2-BS problems as the latter do not require large space for output. The read-only data cache cannot be used here since it cannot be overwritten during the lifespan of the kernel. That leaves the shared memory the only choice. By this design, the data output is done in two stages: (1) whenever the distance function generates a new distance value, it is used to update the corresponding location of the private output data structure via an *atomic write*. Although this still involves an atomic operation, the high bandwidth of shared memory ensures minimum overhead; (2) when all distance functions are computed, the multiple private copies of the output array are combined to generate the final output (Figure 3). Here we assume the final output can be generated via a parallel *reduction* algorithm such as the one presented in [24]. Algorithm 3 shows details of the privatization technique.

As an implementation detail, we use one private copy of the output for each thread block. Thus, the threads in the

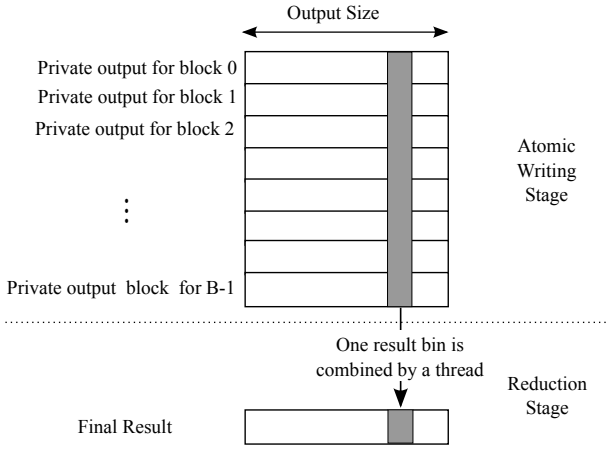

Fig. 3. Combine private output to final result

same block can "simultaneously" update the private output data via atomic operations. By this, the race condition only happens within a thread block, and as we mentioned before, the bandwidth of the shared memory can effectively hide the overhead. We tested more private copies per block and found that it does not bring overall performance advantage (data not shown). In the output reduction phase, private outputs on shared memory are first copied (in parallel) to global memory, which is in global scope and can be accessed by other kernels. Then a reduction kernel is launched to combine the results into a final version of output array. This kernel is configured to have one thread handle one element in the output array.

D. Evaluation of Complete Algorithms

To evaluate the algorithm with optimized pairwise distance computation and outputting stage, we first present an analytical model of the number accesses to different types of GPU memories during the writing output stage. The cost of accessing global memory for writing output in naive algorithm is $N^2 C_{GAtomic}$. As we know that the latency of accessing global memory $C_{GAtomic}$ is high (i.e., over 350 cycle in Maxwell [20]), the naive algorithm is bottlenecked by writing output. On the other hand, our writing output can reduce access time to global memory. There are two stages in output writing. In update stage, the cost of (shared) memory access is:

$$\sum_{i=1}^{M}(N + B - i)C_{ShmAtomic} \tag{6}$$

In the reduction stage, the cost of memory access is:

$$H_s[M(C_{GW} + C_{ShmR} + C_{GR}) + C_{GW}] \tag{7}$$

Considering the number of accesses to global memory by our outputting technique, it is obvious that number of accesses to global memory drops significantly from naive algorithm: it decreases from N^2 to $H_s[2M + 1]$. Moreover, global memory accesses in the reduction stage are done without atomic instruction. In addition, the major activities of writing output is in shared memory with atomic operation that is of low access cost. Putting together, our method of writing output will significantly outperform the naive algorithm.

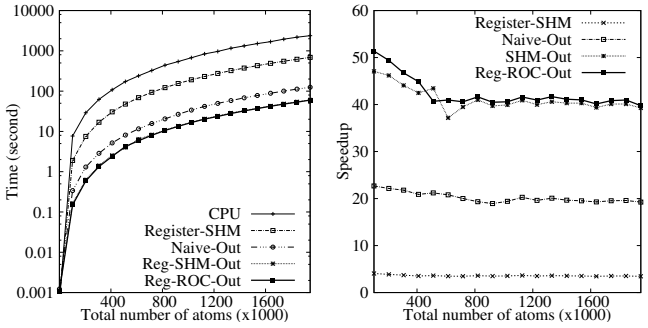

Fig. 4. Performance of different GPU-based algorithms for computing SDH: total running time and speedup over CPU algorithm

For empirical evaluation, we use the Spatial Distance Histogram (SDH) as an example for implementing our algorithms. Classified as a Type-II 2-BS, the SDH is a problem similar to 2-PCF except in the output stage. In particular, SDH also requires computing all pairwise Euclidean distances, but it outputs a histogram that shows the distribution of all distances computed. The output size (i.e., number of buckets) of SDH is not related to the data size N, but it normally comes at the level of tens of kilobytes therefore can be placed in shared memory.

In this set of experiments, we compare six kernel functions: the first three are algorithms we studied in Section IV: Naive, Register-SHM, and Register-ROC. The output stage of those three algorithms is handled in the straightforward way: we directly output to a shared output data structure in global memory via atomic operations. The next set of three algorithms, named Naive-Out, Reg-SHM-Out, and Reg-ROC-Out, are based on the above three but we optimize the output stage with the privatization technique. In addition, we also compare the above algorithms with a CPU-based parallel algorithm to study the overall advantage of running 2-BS on GPUs vs. multi-core CPUs. We again generate uniformly distributed datasets with a size ranging from 512 to 2 million. Other aspects of the experimental setup are the same as those described in Section IV-B.

Design and Implementation of CPU-based Algorithm: We implement a highly-optimized algorithm for computing SDH in multi-core CPUs using OpenMP in C. Optimizations applied to the CPU version are summarized as follows. First, we optimize output stage to reduce the effects of atomic operations. In particular, every thread is given an independent copy of the output histogram and parallel reduction is conducted after all distance function calls are returned. Second, we compare the effects of OpenMP thread affinity schedulers and choose the one most beneficial to overall performance. OpenMP supports three methods of assigning threads to cores: *scatter*, *compact*, and *balanced*. Among these methods, *balanced* shows the best performance in our code and is selected as our thread affinity scheduler. Third, parallel loops can be executed in different scheduling modes. Available mode in OpenMP are: *static*, *dynamic*, and *guided*. Selecting a

TABLE III
ACHIEVED BANDWIDTH OF DIFFERENT MEMORY UNITS IN RUNNING DIFFERENT SDH KERNELS

Kernel	Shared Memory	L2 Cache	Data cache	Global Load
Naive	0 B/s	270 GB/s	32 GB/s	104 GB/s
Naive-Out	1.66 TB/s	437 GB/s	138 GB/s	563 GB/s
Reg-SHM-Out	2.86 TB/s	10 GB/s	3 GB/s	10 GB/s
Reg-ROC-Out	2.59 TB/s	55 GB/s	267 GB/s	68 GB/s

TABLE IV
UTILIZATION OF DIFFERENT GPU RESOURCES IN RUNNING DIFFERENT SDH KERNELS

Kernel	Arithmetic Operation	Control-flow Operation	Memory
Naive	5%	N/A	Max (L2)
Naive-Out	23%	5%	Max (L2)
Reg-SHM-Out	25%	5%	95.33% (SHM)
Reg-ROC-Out	20%	5%	86.33% (SHM) 26.71% (ROC)

scheduling mode is usually a trade-off between overhead and load imbalance. We studied the effects of different schedulers and chose guided as the best one for our algorithm. Other optimizations such as algebraic elimination of costly instructions and enabling aggressive compiler optimizations are also applied to the CPU code. In summary, we believe our CPU code is of very high (if not optimal) performance.

Experimental Results: Figure 4 shows the running time of the aforementioned kernels. First of all, we found that the three kernels without the output privatization technique run at almost at the same speed. Therefore, we just plot one of the three (i.e, Register-SHM) in Figure 4. The total running time of such kernels is about one order of magnitude longer than the ones with output privatization technique. This clearly shows how much performance penalty can be generated by atomic operations against a global memory location. The Visual Profiler shows that performance of this kernel is limited by memory bandwidth instead of compute unit. On the other hand, applying output privatization can significantly improve the speed of kernels, as shown by the short running time of the three output-optimized kernels. The Reg-ROC-Out kernel, by using the ROC for pariwise computation and shared memory for output caching, combines the power of both cache systems and therefore shows the best performance. Specifically, Reg-ROC-Out is about 11 times as fast as Register-SHM.

Further profiling of the involved kernels support discussions made above. Table III shows the achieved bandwidth in different GPU cache systems by the tested kernels. It basically says that shared memory is the limiting factor of the three output-optimized kernels. Among them, Reg-ROC-Out achieved very high bandwidth utilization in both shared memory (2.59TB/s, which is 86.33% utilization) and read-only cache (267GB/s, 26.71% utilization), leading to the best kernel performance. The other two kernels, Reg-SHM-Out and Naive-Out, have

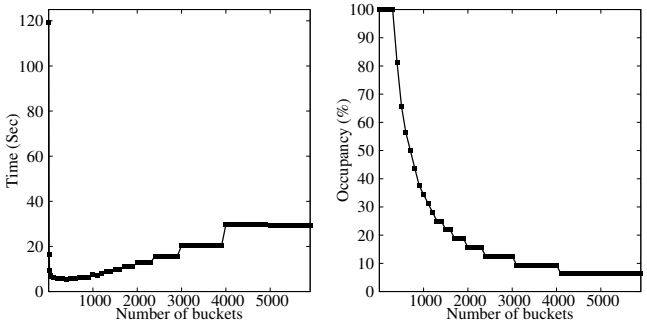

Fig. 5. Performance of the Reg-ROC-Out kernel under different bin sizes: Running time and Occupancy

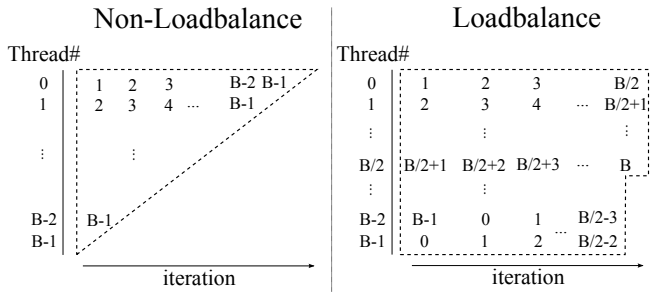

Fig. 6. Comparing regular kernel and load-balanced kernel

lower utilization in either shared memory or ROC. All GPU kernels beat the CPU program running on a 8-core Intel Xeon CPU, showing GPUs being a superior platform for computing 2-BSs. The best GPU program (i.e., Reg-ROC-Out) is about 50 times as fast as the CPU program. Even the least optimized Reg-SHM kernel is about 3.5 times as fast as the CPU code.

We also study the effects of output size on the performance of the output-optimized kernels. Figure 5 shows such results of the Reg-ROC-Out kernel in computing the SDH of a dataset with 512,000 data points. The general trend is: when output size (i.e., total number of buckets in the output histogram) increases, the running time also increases. Note that the running time increases as a step function of output size. This is because the output size affects the performance via changing the occupancy (i.e., number of threads running concurrently in hardware) of the kernel. Figure 5 shows that occupancy decreases when the output size increases. Interestingly, the kernel also shows degraded performance when the output size is too small. This shows the other side of the story: when an output has too few elements, it will suffer from high contention: the many threads in the block always compete for accessing an output element via the atomic operations.

E. Additional Techniques

In this section, we introduce two additional techniques that could help boost the performance of 2-BS programs.

1) Load balancing technique: Code divergence is the situation when different threads taking different execution paths in an SIMD architecture. As a result, such threads will be executed in a sequential manner and that leads to performance

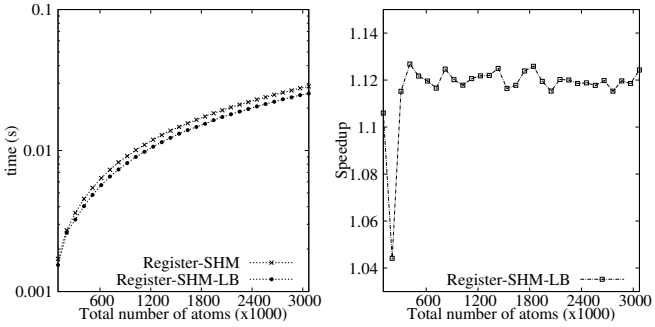

Fig. 7. Performance of different GPU-based algorithm for computing SDH: total running time and Speedup over Register-SHM kernel

penalties. In CUDA, since the basic scheduling unit is a warp (of 32 threads), only divergence within a warp will be an issue. By looking at Algorithm 2, it is not hard to see that the kernel will only suffer from divergence in the intra-block distance function computation (line 9 to 12 in Algorithm 2). This is because each thread goes through a different number of iterations (Figure 6). Here we introduce a *load balancing* method to eliminate divergence from the intra block computation. As we mentioned before, divergence occurs because the workload on each thread is different. Our technique thus enforces each thread to compute the same amount of work, i.e., half of block size. Previously, for a thread with index i in a block (thus $i \in [0, B-1]$), the total number of datum it pairs with is $B-1-i$, meaning every thread has a different number of datum to process. This leads to divergence everywhere. With the load balancing technique, we let each thread pair with $B/2$ datum. In particular, at iteration j, the thread with index i pairs with datum with index $(i+j)\%B$. Figure 6 illustrates the main idea. Note that in the last iteration, only the lower half of all threads in a block need to compute output. This does not cause a divergence as the block size is a multiple of warp size.

We also conduct some experiments to evaluate the load balancing technique. In such experiments, we only record the time for processing intra-block distance function computations in processing the SDH. We implement the load balancing technique on top of the tiling-based kernel Register-SHM, which is shown to be the most efficient solution in Section IV-B. We compare the running time of kernel before and after applying the technique, and Figure 7 shows such results – a 12%-13% improvement can be seen.

2) Tiling with Shuffle instruction: As seen in Section IV-A, tiling via shared memory or read-only cache is the key technique to improve kernel performance. However, under some circumstances, both the shared memory and read-only cache may not be available for the use of 2-BS kernels. For example, they could be used for other concurrent kernels as a part of a complex application. In this section, we present another technique that relieves the dependency on cache. Note that register content is generally regarded as private information to individual threads. However, the *shuffle instruction* introduced

Algorithm 4: Block-based 2-BS with shuffle instruction

```
Local Var: t (Thread id), b (Block id)
Global Var: B (Block size), M (# of blocks), w (warp size)
1:  reg0 ← the t-th datum of b-th input data block loaded
2:  for i = b + 1 to M do
3:      for j = t%w to B; j+=w do
4:          reg1 ← the j-th datum of i-th input data block
            loaded to register
5:          for k = 0 to wsize do
6:              regtmp ← shuffle broadcast of reg1 from the
                k-th thread
7:              d ← DisFunction(reg0, RegTmp)
8:              update output with d
9:          end for
10:     end for
11: end for
```

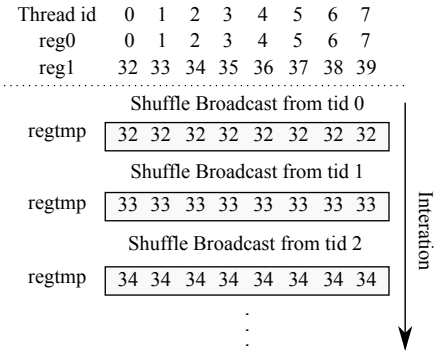

Fig. 8. Tiling with shuffle instruction technique

in recent versions of CUDA allows sharing of register content among all threads in the same warp (not in the same block). By that, we optimize Algorithm 3 by using shuffle instructions. Algorithm 4 shows the pseudo code. We allocate three registers to store input data. Reg0 (line 1) is used to store datum from L which is the same as algorithm 3; Reg1 (line 4) is used to store datum from R and changes after every 32 iterations; Regtmp (line 6) is a temporary variable, which updates every iteration with shuffle instruction. Figure 8 shows tiling with shuffle instruction method. We let each thread load a datum to a register (line 4). Then, in each iteration, shuffle broadcast instruction is used to load data from other thread's registers (line 6) to regtmp. After regtmp value is loaded, Then reg0 and regtmp are used to calculate distance (line 7). As can be seen, this tilling method is required only two more register and dosen't require shared memory or read-only cache.

We also evaluate our technique by implementing it in the algorithm for computing SDH. We conduct experiments similar to those mentioned in Section IV-C. We compare the shuffle instruction with tiling via shared memory and tiling via read-only data cache. Figure 9 shows some results of the experiments. Clearly, tiling with shuffle instruction has almost the same performance as tiling with read-only cache and tiling

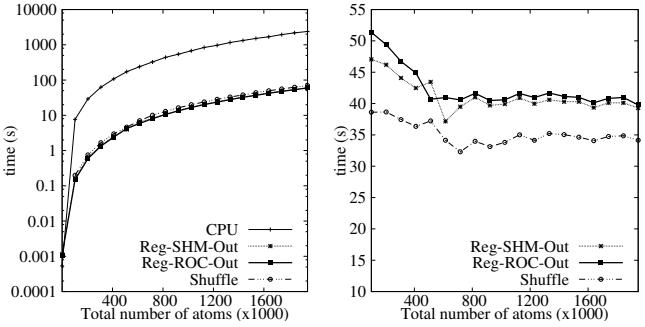

Fig. 9. Performance of different GPU-based algorithm for computing SDH: total running time and speedup over CPU algorithm

with shared memory kernel. This shows that the technique based on shuffle instruction can be an alternative method when shared memory and read only cache are not available, and we expect the algorithm to show the same level of performance.

V. Conclusions and Future work

In this paper, we study parallel algorithms for processing 2-BS by exploiting the high computing power of GPUs. First, we introduce a straightforward parallel algorithm under the CUDA framework. Then, we split it into two stages: pairwise computation and writing output. In order to increase the performance, we present modifications to the algorithm by integrating various techniques in each stage. In pairwise computation stage, we optimize the algorithm by applying blocking and tiling data into multiprocessor using different data paths, shared memory, read-only data cache, and register. We evaluate this stage by 2-PCF problem. The results show that tiling via shared memory and register outperforms other techniques for this type of problems (i.e., type-I of 2-BS). Considering writing output stage, we utilize on-chip shared memory to privatize output and use parallel reduction method to combine each private output. We evaluate this strategy by implementing it on the SDH algorithm. We also found that tiling via data cache can significantly improve speed of the applications in type-II 2-BSs. To further improve the efficiency of the algorithm, we also introduce load balancing and tilling techniques with shuffle instructions.

In the near future, we plan to work on techniques that can improve the efficiency of type-III 2-BSs on GPUs. Moreover, future study should focus on developing the envisioned framework that automatically applies different techniques presented in this paper to a larger group of 2-BSs. The analytical model can be refined to provide accurate prediction of kernel running time by considering more environmental and kernel features. Our work can also be extended to a multi-GPU environment or even cluster-level optimization to handle very large input/output data in 2-BS computation.

References

[1] A. G. Gray and A. W. Moore, "N-body problems in statistical learning," *Advances in Neural Information Processing Systems (NIPS)*, pp. 521–527, 1993.

[2] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Procs. ACM SIGMOD International Conference on Management of Data*, 2008, pp. 511–524.

[3] J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-M. Hwu, and N. Obeid, "Algorithm and data optimization techniques for scaling to massively threaded systems," *Computer , vol.45, no.8*, pp. 26–32, 2012.

[4] B. G. Levine, J. E. Stone, and A. Kohlmeyer, "Fast analysis of molecular dynamics trajectories with graphics processing units-radial distribution function histogramming," *Journal of Computational Physics. J. Comp. Phys.*, pp. 3556–3569, 2011.

[5] Y.-C. Tu, S. Chen, and S. Pandit, "Computing distance histograms efficiently in scientific databases," *ICDE*, pp. 796–807, 2009.

[6] S. Chen, Y. Tu, and Y. Xia, "Performance analysis of a dual-tree algorithm for computing spatial distance histograms," *VLDB J.*, vol. 20, no. 4, pp. 471–494, 2011.

[7] B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999.

[8] L. Rokach and S. Kisilevich, "Initial profile generation in recommender systems using pairwise comparison," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 6, pp. 1854–1859, Nov 2012.

[9] S. Jiang, X. Wang, and H. Zhu, "Learning pairwise comparisons of items with bigram content features for recommending," in *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on*, Oct 2013, pp. 446–449.

[10] *NVIDIA: CUDA C Programming Guide Version 7.0*. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[11] T. Group., "Opencl." [Online]. Available: https://www.khronos.org/opencl/

[12] B. Jensen, J. Saez Gallego, and J. Larsen, "A predictive model of music preference using pairwise comparisons," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, March 2012, pp. 1977–1980.

[13] A. Kumar, V. Grupcev, Y. Yuan, Y.-C. Tu, and G. Shen, "Distance histogram computation based on spatiotemporal uniformity in scientific data," *EDBT*, 2012.

[14] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Procs. ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04, New York, NY, USA, 2004, pp. 215–226.

[15] *NVIDIA. GTX 980 whitepaper*. [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[16] *NVIDIA's Next Generation CUDATM Compute Architecture:Fermi*. [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdfF

[17] *NVIDIA's Next Generation CUDATM Compute Architecture:Kepler GK110*. [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdfF

[18] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for nvidia cuda compatible devices," *Proceedings of the International Conference on Signal Processing and Communications Systems, IEEE, Gold Coast, Australia*, pp. 418–422, 2007.

[19] A. Agrawal and X. Huang, "Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 8, no. 1, pp. 194–205, Jan 2011.

[20] *Analyzing GPGPU Pipeline Latency*, 2014. [Online]. Available: http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf

[21] *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*. IEEE Computer Society, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5446240

[22] *NVIDIA. CUDA C Best Practices Guide, version 7.5*. [Online]. Available: https://developer.nvidia.com/category/zone/cuda-zone

[23] H. Li, D. Yu, A. Kumar, and Y. Tu, "Modeling in cuda strems - a means for high-throughput data processing," *Big Data (Big Data), IEEE International Conference*, pp. 301–310, 2014.

[24] *Optimizing Parallel Reduction in CUDA*. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf