Algorithms and Framework for Computing 2-body Statistics on GPUs

Journal:	*Transactions on Computers*
Manuscript ID	TC-2017-04-0208.R1
Manuscript Type:	Regular
Keywords:	2-Body Statistics, Parallel Computing, GPGPU, GPU, CUDA, I.3.1.d Parallel processing < I.3.1 Hardware Architecture < I.3 Computer Graphics < I Computing Methodologies, I.3.1.a Graphics processors < I.3.1 Hardware Architecture < I.3 Computer Graphics < I Computing Methodologies

Algorithms and Framework for Computing 2-body Statistics on GPUs

Napath Pitaksirianan, Zhila Nouri, and Yi-Cheng Tu, *Senior Member, IEEE*

Abstract—Various types of two-body statistics (2-BS) are regarded as essential components of data analysis in many scientific and computing domains. However, the quadratic complexity of these computations hinders timely processing of data. Therefore, use of modern parallel hardware has become an obvious direction on this topic. This paper presents our recent work in designing and optimizing parallel algorithms for 2-BS computation on Graphics Processing Units (GPUs). Although the typical 2-BS problems can be summarized into a straightforward parallel computing pattern, traditional wisdom from (general) parallel computing often falls short in delivering the best possible performance. Thus, we present a suite of techniques to decompose 2-BS problems and methods for effective use of computing resources on GPUs. We also develop analytical models that guide us towards the best parameters of our GPU program. As a result, we achieve the design of highly-optimized 2-BS algorithms that significantly outperform the best known GPU and CPU implementations. Although 2-BS problems share the same core computations, each 2-BS problem however carries its own characteristics that calls for different strategies in code optimization. For that, we develop a software framework that automatically generates high-performance GPU code based on a few parameters and short primer code input.

Index Terms—2-Body Statistics; Parallel Computing; GPGPU; GPU; CUDA

1 Introduction

Handling analytical workloads efficiently is a major challenge in today's data-intensive applications. Various types of 2-body problems are essential components of data analysis in many application domains. Bearing many forms and definitions, 2-body statistics as we refer to in this paper, is a type of computational pattern that evaluates all pairs of points among an N-point data set. There are numerous examples of 2-Body statistics: 2-tuple problem [1], all-point nearest-neighbor problem [1], nonparametric outlier detection and denoising [1], kernel density regression [1], two-point angular correlation function [2], 2-point correlation function [1], and spatial distance histogram (SDH) [3], to name a few. Another flavor of 2-Body statistic takes two datasets as input: Radial distribution function (RDF) [4] and relational joins [5]. Moreover, there are many computing applications that require pairwise comparison among all data points: item rating with pairwise comparison [6] [7], Cosine similarity, predictive of music preference [8], ranking with pairwise comparison [9] and N-body simulation [10]. In this paper, we use the word "2-body statistics" (2-BS) to cover all of them.

In general, a 2-BS can be computed by solving a *distance* function between all pairs of datum. Although the distance function only demands constant time to compute for a pair of data points, the total running time is quadratic to the data size. The first line of defense is obviously better algorithms with lower complexity. For example, it is well known that sort-merge, hash, or indexed-based algorithms are used to compute relational joins in database systems. Our previous work [3] also used quad-tree and batching technique to reduce the complexity of SDH computing to $O(N^{1.5})$. On the other hand, parallel computing techniques can be utilized to speed up the computation in practice, and is the main topic of 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) [11], Pairwise comparison in various applications [6], [7] can only be solved in quadratic time. Although sort-merge, hash, and index-based algorithms are preferred for processing equijoins, nested-loop join is the better choice for joins with complex non-equality conditions. (2) performance of more advanced algorithms can be further improved via parallelization. For example, efficient join algorithms such as hash join still require complete pairwise comparison of data (e.g., within the same bucket of the hash table), for which parallel programs [5] have shown great success. With that in mind, this paper focuses on novel techniques to implement and optimize parallel algorithms for computing 2-BSs on modern Graphics Processing Units (GPUs).

By providing massive computing power and high memory bandwidth, GPUs have become an integrated 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) [12] and Open Computing Language (OpenCL) [13]. Due to the compute-intensive nature of 2-BS problems and the fact that the main body of computations can be done in a parallel manner for most 2-BSs, GPUs stand out as desirable platform for implementing 2-BS algorithms.

However, GPU algorithms for only a few 2-BS problems have been studied (see Section 2 for details). In addition to the surprisingly little attention paid to this topic, existing work lack a comprehensive study of the necessary

• *N. Pitaksirianan, Z. Nouri, and Y. Tu are with the Department of Computer Science and Engineering , University of South Florida, Tampa, FL, 33613. E-mail: {napath, zhila, tuy}@mail.usf.edu*

techniques to achieve high performance on GPUs. This is a non-trivial task because the architecture of modern GPUs is much more complex than the multi-core CPUs. As a result, traditional wisdom from (general) parallel computing often falls short in delivering the best possible performance. In this paper, we present a suite of techniques to decompose 2-BS problems and methods for effective use of computing resources on GPUs. Many of such techniques such as warp-level privatization (Section 4.3.2), direct output buffer (Section 4.3.3), warp-level load balancing (Section 4.4.1), and shuffle-enhanced tiling (Section 4.4.2) are the innovations not presented in the previous work, to the best of our knowledge. This paper makes the following contributions.

First, we identify two phases in computing typical 2-BS problems: a pairwise data processing phase, and a result outputting phase. In the first phase, we develop various 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. As a result, we achieve the design of highly-optimized 2-BS algorithms that significantly outperform the best known GPU and CPU implementations.

Second, configuration of run-time parameters for the GPU programs has significant effects on performance. For that, we develop analytical models that guide us towards the best choices of key parameters of our program, achieving performance guarantees.

Finally, 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. For that, we develop a software 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 implement core computational kernels developed in our work and output optimized GPU code based on a few parameters and a distance function given by the user.

Paper Organization: The remainder of the paper is organized as follows: in Section 2, we review work related to 2-BC problems; in Section 3, we introduce the technical background of our study; in Section 4, we demonstrate techniques to speed up pairwise computation and writing output on GPUs; We evaluate our GPU algorithm in Section 5; we describe the 2-BS framework for automatic code generation and two case studies in Section 6, and conclude this paper in Section 7.

2 Related Work

There are many applications of 2-BS problems in addition to those mentioned in Section 1. A common practice in many applications is to use various distance measures (e.g., *Euclidean*, *Jaccard*, and *cosine distance*) to find the similarity 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 [8] based on pairwise comparisons of Gaussian process priors between music pieces. There are two types of recommendation systems: content-based filtering (CB) and collaborative filtering (CF) [6], [7]. Both require 2-BS computation: CB depends on pairwise comparisons between items and CF depends 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 [14], [15], 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 and thus parallelization of 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. Surprisingly, there are few reports on computing 2-BSs on GPUs. As a part of efforts to parallelize relational joins, He *et al.* implemented various join algorithms on GPUs and reported a 7X speedup over CPUs [5]. Similar results are presented in [16]. 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 distances. They used data privatization techniques to speed up the algorithm. Ponece R. *et al.* [17] used tiling and privatization via shared memory with two point correlation function in cosmology application. They reported speed up to 100x from single CPU system. More recently, Stratton *et al.* sketched tiling and privatization techniques in computing two-point angular correlation function [2], yet no technical details are reported. Unlike the focus of individual problems and techniques seen in aforementioned 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.

3 Background

3.1 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 (Fig 1). We believe such information is essential in our discussions of (parallel) algorithm design in this paper. Readers already familiar with GPU architecture can skip this subsection.

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 tens of multiprocessors. A GPU contains a few GBs of *global memory* that uses DDR5 technology. The CPU (i.e., host) can transfer data to the global memory over a PCI-E link. Global memory can be accessed by different multiprocessors simultaneously at a bandwidth up to 480 GB/sec [18]. Each multiprocessor also provides high-speed programmable *shared memory* of size 96KB. The use of shared memory is under full control of the programmers. There are also the programmable *read-only data cache* (also named *texture* memory), which was first introduced in the Kepler Architecture, for holding data that cannot be overwritten during the lifespan of the program, as well as the non-programmable 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*) in parallel. The entire collection

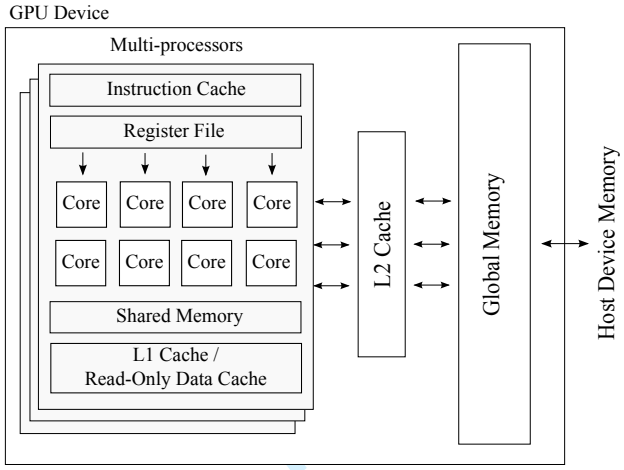


Fig. 1. Architecture of a recent NVidia (e.g., Maxwell, Pascal) GPU

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.

Over the years, the architecture of NVidia GPUs has evolved through several generations: Fermi [19], Kepler [20], Maxwell [21], Pascal [18], and Volta [22]. Newer architectures provide more computing resources. Moreover, along with the newer architectures, there are also new functionalities 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. Kepler also allows launching kernels within an existing kernel. This mechanism is called *dynamic parallelism*. Such features provide new opportunities for improving program efficiency.

3.2 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 input dataset to find other data points for the distance function computation. The output will be updated with the results of each distance function computation.[1]

To optimize the above 2-BS algorithm, the challenges can be roughly summarized as those in dealing with the input and output data, respectively. First, each input datum i will be read many times (by different threads into registers) 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, complicates the problem. Second, every thread needs to read and update the output data at the same time. Updating the output data simultaneously might cause incorrect results. Recent GPUs and CUDA framework 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, clever strategies are needed to avoid update collisions as much as possible.

Algorithm 1: Generic GPU-based 2-BS algorithm

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

Given that, there is a need to characterize the multitude of 2-BS cases based on the computational paths. This helps us to determine the proper combination of techniques we can use for optimizing individual 2-BS problems. We found that the 2-BS we have 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.

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 is too big for registers but are still small enough to be put into GPUs' shared memory. Examples include: (1) Spatial distance histogram (SDH) [3], 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 some extreme cases, the size of the output is quadratic to the size of input. Some examples are: (1) relational join [5], which outputs concatenated tuples from two tables - total number of output tuples can be quadratic (especially in non-equality joins); (2) Pairwise Statistical Significance [23], which is statistical significance of pairwise alignment between two datasets and generates large output; and (3) Kernel methods which compute kernel functions for all pairs of data in the feature space [11].

1. We focus our discussions on 2-BSs defined over a single dataset with commutative distance function (i.e., only one function call is needed for every pair of points). Therefore, the point with index i is only paired with all data points beyond position i. Note there are cases where 2-BS is defined between two different datasets (e.g., relational join) or with non-commutative distance function (e.g., SVM kernel functions). We will mention them in coming sections as needed.

TABLE 1
Symbols and notations

Symbol	Meaning
H_S	Histogram Size or Output Size
N	Number of input datum
B	Block size
M	Number of blocks
M_{BMP}	Maximum number of blocks of an MP
M_{SPM}	Maximum SM amount of an MP
ζ	Actual use of SM in a block
C_L	Latency of writing without conflict
C_{LP}	overhead latency of writing conflict

4 GPU Algorithms Design

In this section, we elaborate on the GPU algorithm design. Symbols/notations used in this paper are listed in Table 1.

4.1 Input Data Representation

Before we discuss algorithmic design, we first present data structures for loading input data. 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. Moreover, we vectorize each dimension array by loading multiple floating point coordinate values in one data transmission unit. In particular, we use the *float2, float3, and float4* data types supported in CUDA for such purposes. This reduces the number of memory transactions and thus the number of load instructions in code. Furthermore, vectorized memory access also indirectly increases instruction level parallelism (ILP) by unrolling a loop to calculate all pairwise distances between two vectors. Thus, in the remainder of this paper, a datum means a vector of multiple data points. Also a distance function call between two datum actually computes all pairwise distances.

As mentioned above, CUDA supports three vector floating point data types – float2, float3, and float4, which holds 2, 3, and 4 regular floating point numbers, respectively. In general, a wider vector yields higher memory bandwidth utilization, but also increases register use in the kernel, which in turn reduces warp occupancy. Therefore, we need to find a balance point between register usage and memory bandwidth. In this paper, the most suitable data type is determined by experiments (Section 5.1).

4.2 Algorithms for Pairwise Computation Stage

Now we present design strategies in the pairwise distance function computation stage. Due to the high latency of data transferring between the global memory and cores, our goal is to reduce the number of data reads from global memory. In particular, we use the well-known *tiling* method [24] 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.

Figure 2 illustrates the tiling idea. 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 vector of input data 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 loads a series of other blocks R. Then, compute distance functions between all pairs of datum of inter and intra blocks.

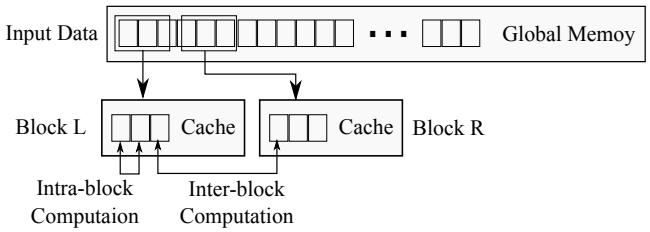


Fig. 2. Tiling method requires loading data in blocks

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: $L \leftarrow$ the b-th input data block loaded to cache
2: **for** $i = b + 1$ to M **do**
3: $R \leftarrow$ the i-th input data block loaded to cache
4: syncthreads()
5: **for** $j = 0$ to B **do**
6: $d \leftarrow$ DisFunction($L[t]$, $R[j]$)
7: update output with d
8: **end for**
9: **end for**
10: **for** $i = t + 1$ to B **do**
11: $d \leftarrow$ DisFunction($L[t]$, $L[i]$)
12: update output with d
13: **end for**

To implement the above algorithm, an important decision to make is: *which cache do we use to hold both blocks L and R?* There is no straightforward answer since there are 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 (RoC), both have TBps-level bandwidth and a response time of just a few clock cycles [25], [26], [27]. According to [25], [27], programmable shared memory has the lowest latency in GPUs (i.e., about 21 clock cycles in NVidia Maxwell), 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 registers 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 a local variable 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 4.3). Plus, latency of accessing registers is just one clock cycle [24]. 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 10 to 13 in Algorithm 2: such computation requires threads to access all datum 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 for L, we overwrite the space we just used for block R. By that, the total shared memory used is still one block. 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 (e.g., outputting results, Section 4.3). This solution basically does not change the code structure of the second solution (with use of registers). However, we use the RoC instead of shared memory to store blocks R (for inter-block computation) and L (for intra-block computation). RoC has higher latency than the shared memory [25] (i.e., about 64 clock cycles higher in NVidia Maxwell) but it is still an order of magnitude faster than global memory. As a side note about implementation, RoC 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 RoC.

4.3 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) – we simply use automatic (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 medium-sized output (e.g., Type-II), we use shared memory to cache the output. We present novel *data privatization* techniques to handle these output. For problems with very large output size (e.g., Type-III), 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. To avoid incorrect results caused by race condition, atomic instructions are used in GPUs to have protected accesses 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. For that, we present a *direct output buffer* (Section 4.3.3) mechanism to minimize such costs. Note that, our paper focus on 64-bit output data type.

Algorithm 3: SDH with Output Privatization

Local Var: t (Thread id), b (Block id)
Global Var: B (Block size), M (total number of blocks)
1: $SHMOut \leftarrow$ Initialize shared memory to zero
2: $reg \leftarrow$ the t-th datum of b-th input data block
3: **for** $i = b + 1$ to M **do**
4: $R \leftarrow$ the i-th input data block loaded to cache
5: syncthreads()
6: **for** $j = 0$ to B **do**
7: $d \leftarrow$ DisFunction(reg, $R[j]$)
8: atomicAdd($SHMOut[d]$, 1)
9: **end for**
10: **end for**
11: $L \leftarrow$ the b-th input data block loaded to cache
12: syncthreads()
13: **for** $i = t + 1$ to B **do**
14: $d \leftarrow$ DisFunction(reg, $L[i]$)
15: atomicAdd($SHMOut[d]$, 1)
16: **end for**
17: syncthreads()
18: $Output[b][t] \leftarrow SHMOut[t]$

4.3.1 Output privatization

Data *privatization* is frequently used in parallel algorithms to reduce race condition [2]. For our problems, we store private copies of the output data structure to be used by a subset of the threads in the on-chip cache of GPUs. The RoC 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 using a parallel *reduction* algorithm such as the one presented in CUDA thrust library. Algorithm 3 shows a new version of Algorithm 2 enhanced with the output privatization technique.

In our initial implementation, we use one private copy of the output for each thread block. By this, the race condition only happens within a thread block, and the bandwidth of the shared memory can effectively hide the performance overhead. We will discuss advanced techniques that involve more private copies in a block in Section 4.3.2. 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.

4.3.2 Advanced Output Privatization Method

So far we have presented a straightforward privatization method in which one private copy of the output is used per thread block. Note that race condition still exists when different threads in the block write into the same output address. If the output size is small enough to allow multiple

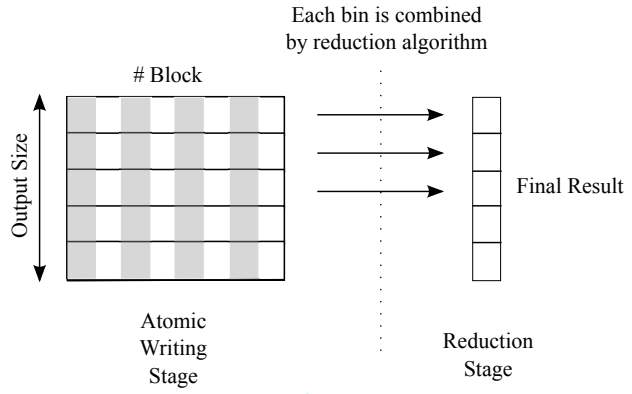


Fig. 3. Combining private outputs in all blocks to obtain the final result

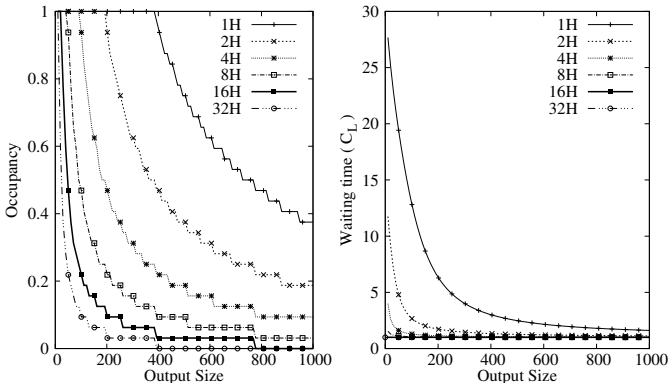


Fig. 4. Modeling results under different numbers of private copies and sizes of the output. Left: thread occupancy as given by the model developed in [28]; Right: latency given by our model

Algorithm 4: 2-BS with Advanced Output Privatization

Local Var: t (Thread id), b (Block id), l (lane id)
Global Var: H_{size} (Output size), H_{num} (number of private copies)
1: $laneID = t$ & 0x1f
2: initial $Output$
3: **for** each pair of pairwise computation **do**
4: $x \leftarrow$2-BS Computation Stage
5: atomic update Output[$H_{size} * (laneID\%H_{num}) + x$]
6: **end for**
7: Output Reduction Stage

private copies for the same block of threads, the probability of collision in atomic operations will decrease, leading to better efficiency of parallelization. To realize this idea, there are two problems to solve: (1) how to assign threads (within a block) to the multiple private copies; and (2) how to exactly determine the number of required copies.

As to the first problem, it is natural to assign threads with continuous thread IDs to a copy of temporary output. For example, with two private copies in each threads block of size B, threads with IDs in $[0, B/2)$ share the first copy and those with IDs in $[B/2, B)$ access the second copy. However, we found that this method does not further improve the performance of the kernel. This is due to the run-time thread scheduling policy in CUDA: every 32 threads with consecutive IDs (called a *warp*) is the basic scheduling unit. In other words, only threads in the same warp are guaranteed to run at the same time thus face the penalty of collision due to atomic operations. Threads in different warps do not suffer from this issue; thus, assigning them to different output copies does not help. Therefore, our idea is to *assign threads with the same offset of IDs in the warp to an output copy*. Going back to the previous example, we now assign threads with the even-numbered IDs in all warps to share the first output copy and those with the odd-numbered IDs to the second copy. Algorithm 4 shows details of this enhanced method: each private output is shared by threads whose IDs have the same 5 least significant bits (called *laneID*). Upon completing a distance computation, each thread updates its corresponding copy of the output (line 5).

The second problem (i.e., finding the best number of private outputs per block) is non-trivial: more copies will decrease the chance of collision in atomic writes, but may decrease the number of threads running simultaneously due to the limited size of shared memory. The impact of the latter has been studied in our previous work [28]. Based on that, we develop an analytical model to quantify both effects and find the balance point that leads to maximal kernel performance. We start with the following performance model for compute-intensive kernels shown in [28].

$$R = \left\lceil \frac{T}{\alpha \times M_P} \right\rceil \tag{1}$$

where R is the number of rounds that takes to schedule all thread blocks in the hardware, $T = M \times B$ is the total number of threads, α is the number of threads that can be run in each round in a single multiprocessor (a.k.a. *thread occupancy*), and M_P is the total number of multiprocessors in a GPU. Note that CUDA allows a large number of blocks to be launched yet there are only a few multiprocessors in a GPU device. Thus, the number of rounds is obviously determined by the occupancy, as all other quantities in Equation (1) are constants. The occupancy, in turn, is affected by the use of common resources for each block, which in our case is shared memory and is determined by the number of private output copies. Due to page limitation, we skip the model describing the relationship between occupancy and shared memory use developed in [28]. Instead, we plot the occupancy calculated from the model in Figure 4 (left subfigure). As we can see, kernel occupancy drops dramatically with the increase of number of copies and output size.

Let L be the latency of running a single round, we obviously have $R \times L$ to be the total kernel running time. In our case, latency is dominated by the time each thread idles due to the conflict of atomic operations. Let k denote the number of threads sharing the same private output in a warp (thus causing a conflict), latency can be then defined as a function of k

$$L(k, C_L) = \begin{cases} C_L & k = 1 \\ p_k C_L + (1 - p_k)P & k > 1 \end{cases} \tag{2}$$

Specifically, if each thread in a warp has its own private output ($k = 1$), there should be no conflict and we can denote

the latency under this (ideal) situation as C_L. If multiple threads share a private output, latency is determined by the probability of seeing a collision-free warp (p_k) and a penalty of collision P, which can be defined as

$$P = L(k-1, C_L + C_{LP}) \quad (3)$$

In other words, L becomes a recursive function defined over a higher latency time $C_L + C_{LP}$ and fewer conflicting threads $k-1$. Again, p_k is the probability that all threads in the same warp access different address locations in the outputs. This can be modeled as a classic birthday problem [29], and we have:

$$p_k = \frac{H_S-1}{H_S} \times \frac{H_S-2}{H_S} \times \frac{H_S-3}{H_S} \times ... \times \frac{H_S-(k-1)}{H_S} \quad (4)$$

This says that the first thread can update any address, the second thread can update any address except the first thread's output address, and so on. By using Taylor series expansion of e^x, the expression approximates to:

$$p_k \approx e^{\frac{-k(k-1)}{2H_S}} \quad (5)$$

Figure 4 (right subfigure) shows latency derived from our model under various values of k and H_S. Note that the latency data plotted here is of unit C_L instead of absolute time in seconds. Here C_L is a hardware-specific value that can be obtained via experiments. Clearly, the latency decreases when there are more private copies of outputs. In case of only a single output (1H), and when the output size is small the latency time is very high. As a side note, the output size obviously plays a role in both sides of Figure 4: with the increase of output size, we have lower occupancy (due to higher shared memory consumption) but lower latency (due to less conflict in accessing the output).

With the above model, we can find the optimal number of private copies. Given any output size (this is a user-specified parameter for a 2-BS problem), we can use different values of k to solve both Equations (1) and (2) to get the estimated total running time. Luckily, k is an integer ranging from 1 to 32 (i.e., CUDA warp size), therefore we can try all such k values to get the one k value that leads to the best running time.

4.3.3 Direct Output buffer

Now we present a technique to handle a common problem in Type-III 2-BSs: allocating GPU (global) memory for output whose size is unknown when the kernel is launched. The problem is due to the fact that CUDA only allows memory allocation for data with a static size. Such a problem has been a real difficulty for not only 2-BS computation but many other parallel computing patterns as well. A typical solution [5] is to run the same kernel twice – the first time is for determining the output size only, and the memory for output is actually allocated and updated in the second run. This obviously imposes a big waste of time.

We propose a *buffer management* mechanism to handle unknown output size, and it only requires one single run of the kernel. Plus, this mechanism does not require any synchronization among threads. First, we allocate an output buffer pool with a fixed size. Then, we divide it into small chunks called *pages*. We keep a global pointer GP that holds the position of the first available page in the buffer pool.

Algorithm 5: 2-BS with Direct Output Buffer

Local Var: buf (current buffer page), $count$ (page usage)
Global Var: GP (next free page), b (Page size)

```
1: buf ← atomicAdd(GP, b)
2: count ← 0
3: for each pair of input datum do
4:    x ←Pairwise Distance
5:    buf[count++] ← x
6:    if count == b then
7:       buf ← atomicAdd(GP, b)
8:    end if
9: end for
```

Each thread starts with one page and fills the page with output by keeping its own pointer to empty space in that page. Once the page is filled, the thread acquires a new page pointed to by GP via an atomic operation. By using this mechanism, conflicts among threads remains minimal because GP is only updated when a page is filled. Algorithm 5 shows the 2-BS augmented with this mechanism. The algorithm starts from initializing local buffer pointer by using *atomic add* operation from global buffer pool and set count to 0 (line 1-2). Then, the algorithm adds each update to local buffer page (line 5). If local buffer page is filled, the algorithm requests a new page by another atomic operation (line 7).

4.4 Additional Techniques

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

4.4.1 Load balancing technique

Code divergence is the situation when different threads follow different execution paths in an SIMD architecture. As a result, such threads will be executed in a sequential manner and that leads to performance 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 10 to 13 in Algorithm 2). This is because each thread goes through a different number of iterations (Figure 5). 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 the 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 that 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 5 illustrates the main idea. Note that, in the last iteration, only the lower half of all threads in a block need to compute the output. This does not cause a divergence as the block size is a multiple of warp size.

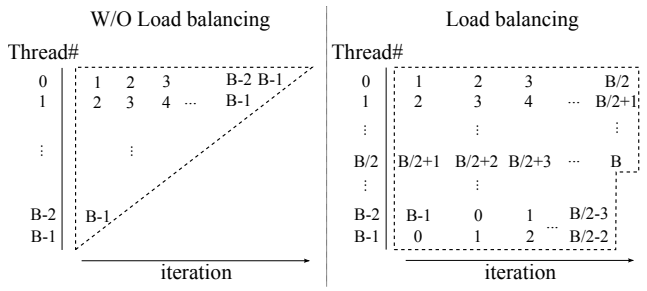


Fig. 5. Two different ways of work assignment to threads in intra-block pairwise computation

Algorithm 6: Block-based 2-BS with shuffle instruction

```
Local Var: t (Thread id), b (Block id), W (warp size)
Global Var: B (Block size), M (total number of blocks)
1:  reg0 ← the t-th datum of b-th input data block
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
5:      for k = 0 to W do
6:        regtmp ← reg1 broadcasted from the k-th thread
7:        d ← DisFunction(reg0, regtmp)
8:        update output with d
9:      end for
10:   end for
11: end for
```

4.4.2 Tiling with Shuffle instruction

As seen in Section 4.2, tiling via shared memory or RoC is the key technique to improve performance of Type-I 2-BS programs. However, under some circumstances, both the shared memory and RoC 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 in recent versions of CUDA allows sharing of register content among all threads in the same warp (not in the same block). Therefore, we augment Algorithm 2 with using shuffle instructions and show the pseudocode in Algorithm 6. In particular, we allocate three registers to store input data: *reg0* (line 1) is used to store datum from L which is the same as algorithm 2; *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. We let each thread loads a datum to *reg1* (line 4). Then, in each iteration, shuffle broadcast instruction is used to load data from other thread's register (line 6) to *regtmp*. After *regtmp* value becomes valid, *reg0* and *regtmp* can be used to calculate distances (line 7). Figure 6 shows an example. Note that this method requires only two more registers and does not require shared memory or read-only cache.

5 Evaluation GPU Algorithm

In this section, we present empirical evaluation of aforementioned algorithms. We run our experiments in a workstation

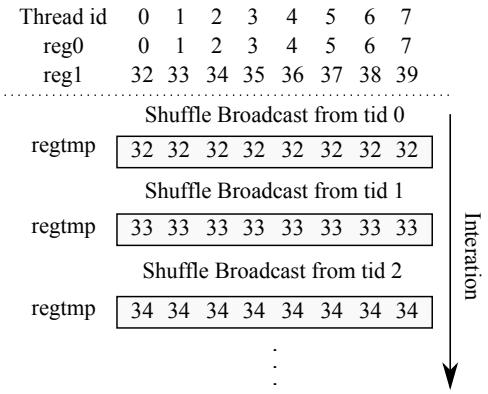


Fig. 6. Tiling with shuffle instruction technique

running Linux (Ubuntu 14.04 LTS) with an Intel Xeon CPU E5-2620 v3, 64GB of DDR3 1333-MHz memory and an Nvidia Titan XP GPU with 12GB of global memory.

5.1 Data Representation Schemes

We first evaluate performance of vectorized memory accesses via different data types (i.e., float2, float3, and float4). 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 computation is almost exclusively on the distance computation, which requires intensive data loading from global memory.

We select two different caching techniques to conduct this experiment, which are register with shared memory and register with RoC. In particular, we implemented and compared eight kernels with different data types and caching techniques. There are four kernels that are based on "Register + Shared Memory" (i.e., named Float-SHM, Float2-SHM, Float3-SHM, and Float4-SHM) and other four kernels are based on "Register + RoC" (i.e., named Float-ROC, Float2-ROC, Float3-ROC, and Float4-ROC). We experimented on input data sizes ranging from 512 to 3 million particle coordinates. Particle coordinates are generated following a uniform distribution in a spatial region.

Figure 7 shows the running time of all eight kernels. As we seen, kernels that use Shared Memory show similar result by using Float2, Float3 and Float4. However, when input data is greater than 1.8 Million atoms, Float2-SHM shows the best performance, which is 5% faster than scalar load (i.e., Float-SHM). The float3 and float4 cases did not show significant advantage. On the other hand, kernels that use RoC demonstrate a more clear trend that Float2-ROC outperforms all other kernels. The Float2-ROC kernel is 11% faster than scalar load (i.e., Float-ROC). However, larger vector width (i.e., float3 and float4) did not further improve performance, the Float3-ROC kernel is even slower than the scalar load case. To get insights on such results, we analyzed the runtime statistics of the kernels by using the NVidia visual profiler, a tool for analyzing runtime characteristics of CUDA kernels. The profiler results show that vectorized memory access via float3 and float4 yield lower perfor-

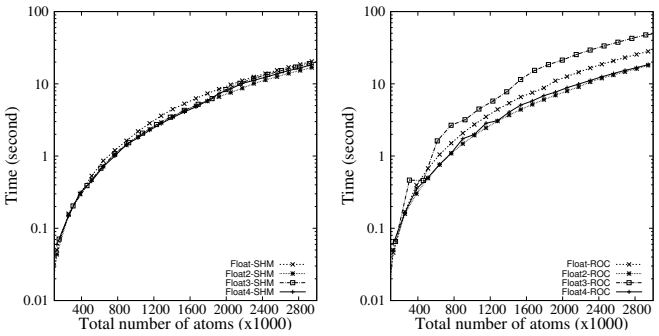


Fig. 7. Performance of different data types of vectorized memory access for computing 2-PCF

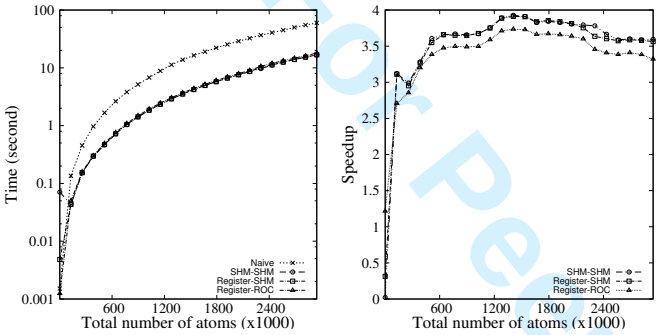


Fig. 8. Performance of different GPU-based algorithms for computing 2-PCF: total running time and speedup over naive algorithm

mance because they significantly increased register use of kernel and reduced warp occupancy (i.e., 75% on float3 and 50% on float4).

Based on the above results, in the rest of experiments, we vectorize all input data into float2. The only exception is the naive algorithm, in which we still use scalar load.

5.2 Evaluation of Pairwise Algorithms

To evaluate the performance of the aforementioned solutions in distance computation, we implemented them in CUDA and experimented using synthetic data with different sizes. We still used 2-PCF whose workload is exclusively on the distance computation, as the sample problem. We implemented and compared the following kernel functions that correspond to the different solutions mentioned above: (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. Note that, in all of the kernels except Naive, input variable is vectorized by float2. In addition, all kernels are compiled by -Xptxa-dlcm=ca flag, which enables the compiler to use L1 cache.

We experimented on input data size ranging from 512 to 3 million particles. Particle coordinates are generated following a uniform 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 512, which is derived from an optimization model developed in our previous work [28]. The model guarantees best kernel performance among all possible parameters by minimizing *running round* (i.e, number of rounds all the specified threads of a kernel are actually scheduled). The model also shows that running round is limited by three factors – shared memory consumption, register use, and number of concurrent warps. As our kernel in 2-PCF uses a small amount of resources in all three categories, we can use a relatively large block size of kernel (i.e, 512 thread per block) and achieve the best performance.

TABLE 2
Utilization of different GPU resources in running different 2-PCF kernels under a data size of 512k. Ari: arithmetic operation; Con: control operation; Mem: memory operations; SM: shared memory; GM: global memory

Kernel	GPU Cores			Memory Bandwidth		
	Ari	Con	Mem	SHM	L2	ROC
Naive	20%	5%	15%	10%	90%	40%
SHM-SHM	67%	14%	4%	20%	10%	10%
Reg-SHM	68%	14%	4%	20%	10%	10%
Reg-RoC	55%	12%	12%	10%	20%	50%

Figure 8 shows the total running time of each experimental run. We observed 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 and SHM-SHM kernel show similar results, which is the best performance under all data sizes – it achieves an average speedup of 3.9X (maximum speedup of 3.5X). The Register-RoC kernel shows the least improvement over naive algorithm, with an average speedup of 3.3X and maximum speedup of 3.7X. 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 utilization of GPU resources 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 can feed the cores with sufficient data, the cores will show a high utilization, which indicates 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 algorithms mentioned so far, 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. Table 2 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 a roughly 67% utilization of arithmetic units in GPU cores, indicating near-optimal performance. That number for Register-RoC is only 55%, verifying the result that its performance is not as good as the other two. Without a surprise, it reaches a high utilization (1.5 TB/s or 50%) of RoC bandwidth.

5.3 Evaluation of Complete Algorithms

In this subsection, we present experimental results on running the algorithms optimized for both (i.e., distance computation and output transmission) stages. We study the

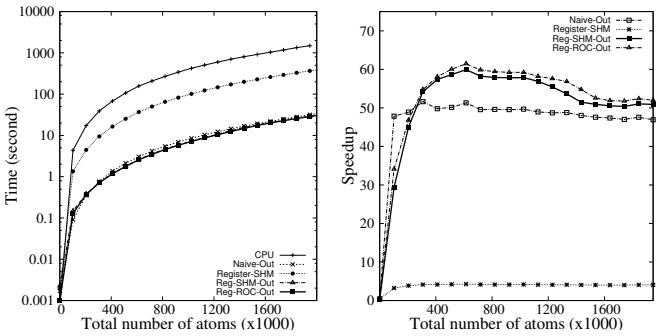


Fig. 9. Performance of different GPU-based algorithms for computing SDH: total running time and speedup over CPU algorithm

impact of each output technique (i.e., output privatization, advanced output privatization, and Direct output buffer) separately.

5.3.1 Output privatization

We use the Spatial Distance Histogram (SDH) as an example for implementing our output privatization algorithm. Classified as a Type-II 2-BS, the SDH is a problem similar to 2-PCF. SDH also requires computing all pairwise Euclidean distances, but it outputs a histogram that shows the distribution of all computed distances. 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 a few 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 4: Naive, Register-SHM, and Register-RoC. The output stage of those three algorithms is handled in a straightforward way: we directly output to a shared data structure in global memory via atomic operations. The other three algorithms, named Naive-Out, Reg-SHM-Out, and Reg-RoC-Out, are based on the first three algorithms, but we enhance the output stage with the privatization technique. In addition, we compare all GPU 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. We set the total number of threads as the data size and the value of threads per block to 64, which is derived from an optimization model developed in our previous work [28]

Design and Implementation of CPU-based Algorithm: We implement a highly-optimized parallel algorithm for computing SDH in multi-core Intel Xeon using OpenMP in C. To improve performance, various techniques are applied to the CPU version. First, we optimize the 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 will be conducted after all distance function calls are returned. Second, we compare the effects of OpenMP thread affinity schedulers (e.g., *scatter*, *compact*, and *balanced*) and choose the one (i.e., *balanced*) that is most beneficial to overall performance. Third, parallel loops can be executed in different scheduling modes, and selecting a scheduling mode is usually a trade-off between overhead and load imbalance. Among the available modes (e.g., *static*, *dynamic*, and *guided*) in OpenMP, we choose *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.

TABLE 3
Utilization of different GPU resources in running different SDH kernels for a 512,000-point dataset. Ari: Arithmetic Operation; Con: Control Operation; Mem: Memory Operation; SM: shared memory; GM: global memory

Kernel	GPU Cores			Memory Bandwidth		
	Ari	Con	Mem	SM	L2	ROC
Register-SHM	10%	10%	10%	10%	10%	10%
Naive-Out	23%	5%	7%	95%	10%	10%
Reg-SHM-Out	50%	20%	20%	98%	10%	10%
Reg-RoC-Out	60%	20%	10%	100%	10%	30%

Experimental Results: Figure 9 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 the same speed. Therefore, we just plot one of the three (i.e, Register-SHM) in Figure 9. It is easy to see that 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 data output dominates the running time due to atomic operations (against a global memory). 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 read-only cache for distance function 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 13.48 times as fast as Register-SHM. Even the Naive-Out algorithm, without any optimization for pairwise distance computation, shows a 12.05 speedup over Register-SHM.

Further profiling of the involved kernels support discussions made above. Table 3 shows the bandwidth utilization in different GPU cache systems by the tested kernels. Clearly, cache bandwidth is the limiting factor of the three output-optimized kernels. Among them, Reg-RoC-Out achieves very high bandwidth utilization in both shared memory (9TB/s) and read-only cache (750GB/s), leading to the best kernel performance. The other two kernels, Reg-SHM-Out and Naive-Out, have lower utilization in either shared memory or RoC. All GPU kernels beat the CPU program running on a Intel Xeon, showing GPUs being a superior platform for computing 2-BSs. The best GPU program (i.e., Reg-RoC-Out) is about 52 times as fast as the CPU program. Even the least optimized Reg-SHM kernel is about 3.86 times as fast as the CPU code.

We also study the effects of output size on the performance of the output-optimized kernels. Figure 10 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

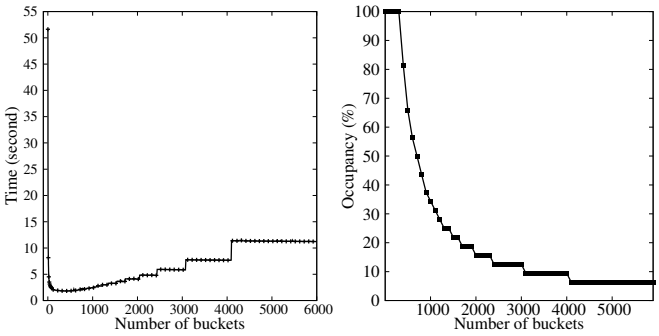


Fig. 10. Performance of the Reg-RoC-Out kernel under different bin sizes: Running time and Occupancy

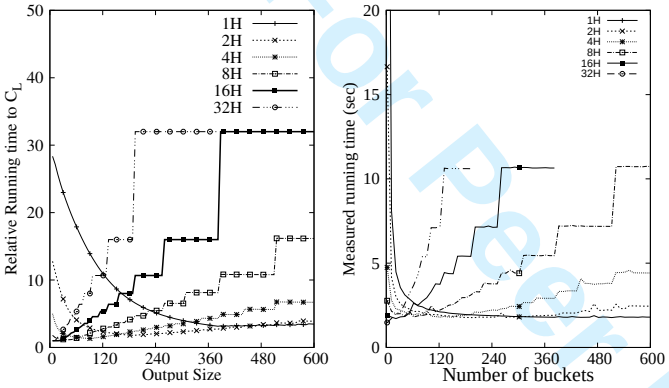


Fig. 11. Performance with advanced output privatization: theoretical (left) and measured (right) running time of SDH kernel. Each line represents the case of one particular number of private output copies

function of output size. This is because the output size affects the performance via changing the occupancy of the kernel. Figure 10 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. In the following section, we will show that advanced output privatization techniques is the remedy for this problem.

5.3.2 Advanced Output privatization

We present our empirical evaluation of our output privatization and verify our running time model. We continue using SDH to evaluate our algorithm. According to the experiments in Section 5.3.1, when output size is too small, SDH renders more shared memory accesses for updating outputs and suffers from long running time. Therefore, we implement our advanced output privatization technique on top of Algorithm 3, and evaluate it with a 512,000-point dataset under output size from 1 to 600 buckets.

Figure 11 shows theoretical running time (left) obtained from our models shown in Section 4.3.2 and actual running time (right) of implemented algorithm. It is easy to see that our model matches the empirical running time very well. Recall that our modeling work aims at finding the optimal number of private output copies. Given any output size, this

TABLE 4
Number of private copies H and the range of output size for which H is found to be the optimal choice

Number of copies	Theoretical Results	Actual Results
32	1 - 10	1 - 10
16	11 - 30	11 - 35
8	31 - 65	36 - 92
4	66 - 150	93 - 152
2	151 - 450	153 - 300
1	$>$ 450	$>$ 300

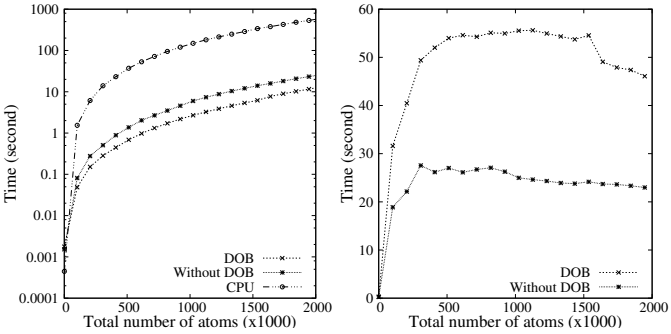


Fig. 12. Output Buffer Management: total running time and speedup over CPU in computing the item similarity problem

can be easily found in Figure 11. Table 4 shows how well the theoretical results predict the best choice in real-world. In particular, we see that any number of private copies is picked under a continuous range of output size (e.g., for output size 1-10, 32 copies are found to be optimal by both modeling and experiments). When the output size is small (i.e., less than 65), our model is very accurate in selecting the best number of private copies. When the output size is between 65 and 152, our model gives more wrong selections among H values of 8, 4, and 2. If we look closely into Figure 11, the running time for 8, 4, and 2 copies are almost the same for both the theoretical prediction and actual values (i.e., less than 5% difference) for a wide range of output sizes. Therefore, modeling errors won't be large enough to affect final choice of H.

5.3.3 Direct output buffer

We also conduct experiments to evaluate the direct output buffer technique. For that, we use a Type-III 2-BS problem: the *item similarity* problem. It computes all pairwise distances and saves those pairs that are found to be similar. Naturally, the output size of this problem is unknown at the beginning. In our experiments, we use data with up to 2 million items and output size around 4 million pairs. We compare our technique with the multi-thread CPU program and our kernel without using the Direct output buffer technique. The CPU program is a variance of the one described in Section 5.3.1. On the other hand, the baseline GPU code encapsulates all optimizations described in previous sections but it handles output by running the kernel twice (i.e., first run only calculates output size). Figure 12 demonstrates running time of the experiment. As we can see, our GPU code without the output buffer technique achieves an average 48X speedup over CPU program (Maximum speedup of 56X). When the direct output buffer is

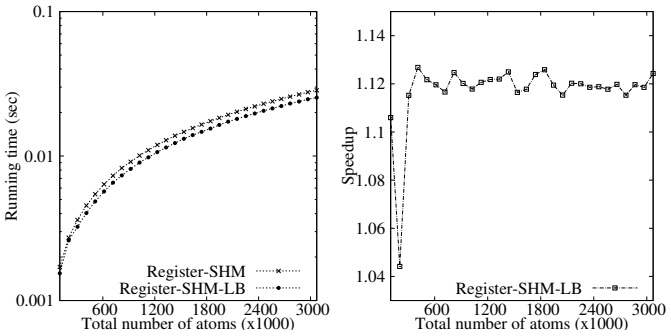


Fig. 13. Performance of Reg-SHM kernel with and without load balancing method: total running time (left) and speedup (right) over Reg-SHM

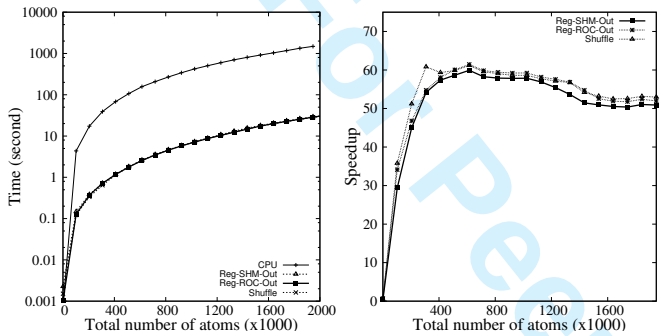


Fig. 14. Performance of different GPU-based algorithms for computing SDH: total running time and speedup over CPU algorithm

implemented, the speedup averagely increases to around 23X, this translates into a 2X speedup generated by the output buffer mechanism.

5.4 Additional Techniques

In this subsection, we present empirical evaluation of additional techniques on SDH problem.

Load balancing technique: In such experiments, we only record the time for processing intra-block 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 5.2. We compare the running time of kernel before and after applying the technique, and Figure 13 shows such results – a 12%-13% improvement can be seen.

Tiling with Shuffle instruction: To evaluate the this technique, we run experiments in a way similar to those mentioned in Section 5.3.1. We compare the shuffle instruction with tiling via shared memory and tiling via RoC. Figure 14 shows the results of the experiments. Clearly, tiling with shuffle instruction has almost the same performance as tiling with RoC or with shared memory. This shows that the technique based on shuffle instruction can be an alternative method when shared memory and RoC are not available, and we expect the algorithm to show the same level of performance.

6 Automated Code Optimization for 2-BS

We have so far presented a multitude of techniques to optimize 2-BS code on GPUs. It is clear that different techniques are effective at different stages for different 2-BS

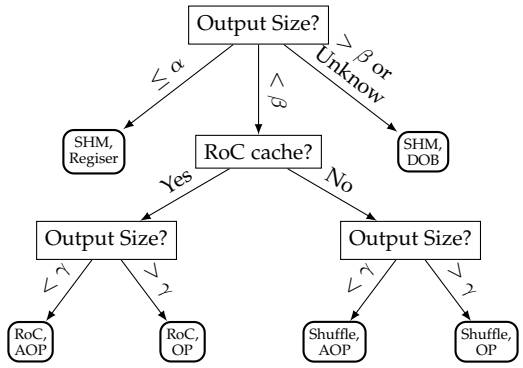


Fig. 15. Decision tree of 2-BS Framework: SHM (cached input on shared memory), RoC (cache input on read only data cache), Register (cached output on register), DOB (Direct output buffer), OP (output privatization), AOP (Advance output privatization)

problems. This proves a burden in code development to users with new 2-BS problems with arbitrary characteristics. In this section, we introduce a framework that encapsulates all aforementioned techniques and automatically generates optimized GPU code for user-defined 2-BS problems. To develop code for a 2-BS problem, our framework only requires the following inputs: (1) a distance function; (2) information about the type, number of dimensions of the input data and the number (1 or 2) of input datasets; (3) an output data structure and its (estimated) size; and (4) specifications of the target GPU device. Based on such information, our framework outputs almost complete CUDA kernels that reflect the optimized strategy for computing the given 2-BS.

Our framework stores chunks of template code reflecting the individual techniques mentioned above as well as the models we developed for kernel optimization. In addition, we develop a **rule-based engine** that integrates different chunks of code into executable CUDA kernels. For example, critical components of the rule-based engine are about decision-making with the different sizes of the output data. This can be seen as a decision tree in Figure 15. If output size is tiny or equal to a threshold α (i.e., Type-I), the code will be generated based on caching inputs into shared memory and outputs into registers. Otherwise, if output size is larger than a threshold value β or unknown (i.e., Type-III), the code will cache inputs in shared memory and use direct buffer to handle output. For Type-II problems, we will check available RoC size. If there is enough RoC to hold input data, RoC will be used as input cache. Otherwise, shuffle instruction techniques will be used for caching input data. Then we check the output size again, if output size is greater than a threshold γ, regular output privatization will be used. Otherwise, warp-level output privatization will be used. The thresholds are set as follows: we set α to 16 bytes – the size of the largest primitive type (i.e., float4, int4) supported of CUDA. This is because anything larger than that (e.g., an array) will be placed in global memory. We set β to the size of shared memory (i.e., 64K for Pascal); and γ is given by the modeling results shown in Table 4.

We developed the framework with python [30] and evaluate its effectiveness with the following two case studies.

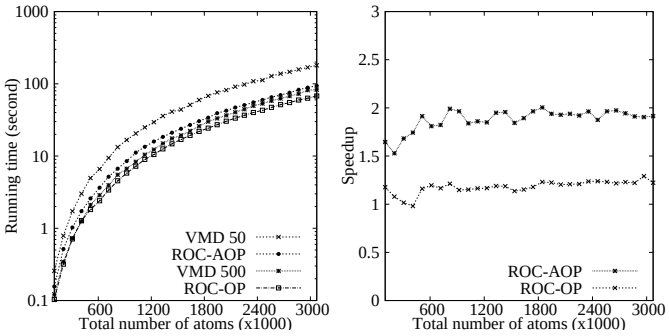


Fig. 16. Performance of different GPU-based algorithms for computing RDF: total running time (left) and speedup over VMD code (right)

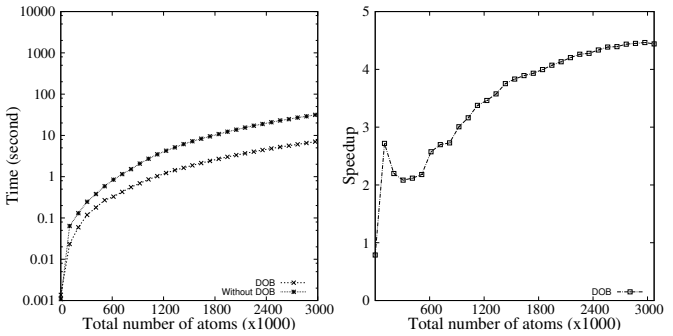


Fig. 17. Performance of NLJ kernel generated from 2-BS framework as compared to best known NLJ kernel reported in [31]

6.1 Case Study I: Radial Distribution Function (RDF)

RDF is an essential physical feature of molecular systems. The RDF algorithm receives two sets of atom coordinates as the input and calculates distances between two atoms from different sets in a pairwise manner. The output is a histogram of the distances between two atoms. Therefore, RDF can be classified as a Type-II 2-BS. In this study, we compare the performance of our RDF code (generated automatically) with the best known code extracted from the Visual molecular dynamics (VMD) software. VMD is a molecular visualization program used for displaying, animating, and analyzing large biomolecular systems. RDF is implemented on GPUs [4] and the code is included in the current VMD release.

We create two sets of experiments to test our framework and compare with existing VMD kernel. In the first experiment, we set the size of output histogram to 500 buckets (i.e., 2KB). In this case, the framework generates a kernel (named RoC-OP) with tiling cached in RoC and regular output privatization. In the second experiment, we set the output size to 50 buckets (i.e., 200 bytes). The framework generates kernel (named RoC-AOP) with tiling by RoC but with warp-level output privatization. In both experiments, we use two datasets with a size ranging from 100k to 3 million particles. Particle coordinates are generated randomly following a uniform distribution. The RoC-OP has 45 lines of code while RoC-AOP is 51 lines long.

Figure 16 shows the running time of relevant kernels under different input sizes. The results of original VMD code under two output sizes are labeled as *VMD50* and *VMD500*, respectively. As can be seen, the running time grows quadratically with increase of data size. We compare the performance of RoC-OP with VMD500 and RoC-AOP with VMD50. For output size of 500, the RoC-OP kernel is faster than the VMD code by up to 18%. Under output size of 50, the RoC-AOP kernel achieves an average speedup of 1.88X and maximum speedup of 2X. This clearly shows that our framework is more adaptive to different scenarios of the same problem. Although the VMD code does a good job under large output, it does not capture the opportunity to handle smaller output more efficiently via the warp-level privatization.

6.2 Case Study II: Nested-Loop Join

We also use Nested Loop Join (NLJ) as an example to verify the effectiveness of our 2-BS framework. As mentioned earlier, being the preferred algorithm for processing joins with complex non-equality conditions, NLJ is important in database systems. In particular, NLJ requires to compare all pairs of tuples from two tables. The output size cannot be determined at the beginning of the run: the size ranges from 0 to N^2 where N is the table size.

In this experiment, each tuple contains a randomly generated integer ID and a key value. Tables' sizes range from 1M to 3M tuples. We limit the output size to be roughly the same as the input table size. We feed such parameters and the join function to our 2-BS framework to generate the CUDA kernel. Our framework classifies NLJ as a Type-III 2-BS problem and chooses to cache input in shared memory and use direct buffer output to handle the output of the application. As a result, around 70 lines of kernel code are generated. We run the generated kernel, and compare its performance to a GPU-based NLJ program developed in previous work [31]. The latter is believed to be the most efficient GPU-based NLJ development, beating all other NLJ programs in performance by a large margin.

As shown in Figure 17, the kernel generated from the 2-BS framework clearly outperforms the state-of-art program, with a speedup up to 4.4X. Looking into the details, the code in [31] is designed to tile input table into on-chip memory (i.e., shared memory and RoC) and it does not use direct output buffer. Note that we also experiment input size beyond 3M tuples, but the speedup stays at 4.45X. This shows that 2-BS framework can automatically generate kernel with very high performance with very little effort from the developer.

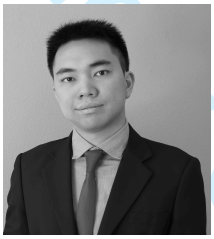
7 Conclusions

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 divide the problem into two stages: pairwise computation and writing output. In order to increase the performance, we present modifications to the algorithm by integrating various novel techniques in each stage. In the pairwise computation stage, we optimize the algorithm by blocking and tiling data into multiprocessors using different data paths, shared memory, read-only

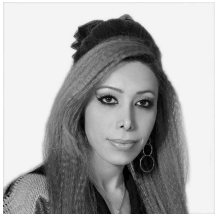
data cache, and register. We evaluate this stage by 2-PCF problem. The results show that tiling via shared memory and register outperform other techniques for this type of 2-BS problems by up to 4 times. Considering the writing output stage, we utilize on-chip shared memory to privatize output and use parallel reduction method to combine each private output. Experiments show that privatizing output can improve speed up to 13 times and lead to a 52X speedup over a highly-optimized parallel CPU version for the same problem. We also introduce direct buffer output for 2-BS with large outputs whose size is unknown at compile time. To further improve the efficiency of the algorithm, we also introduce load balancing and tilling techniques with shuffle instructions. Moreover, we develop a general 2-BS framework that can automatically generate CUDA code for user-defined 2-BS problems. 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] 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.
[3] Y.-C. Tu, S. Chen, and S. Pandit, "Computing distance histograms efficiently in scientific databases," *ICDE*, pp. 796–807, 2009.
[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] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Procs. ACM Intl. Conf. Management of Data (SIGMOD)*, 2008, pp. 511–524.
[6] 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.
[7] 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.
[8] 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.
[9] Y. Hairu, J. Zhengyi, and Z. Haiyan, "Exploring a method for ranking objects based on pairwise comparison," in *Computational Sciences and Optimization (CSO), 2011 Fourth International Joint Conference on*, April 2011, pp. 382–386.
[10] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2008, pp. 677–695.
[11] B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999.
[12] *NVIDIA: CUDA C Programming Guide Version 7.0.*
[13] T. Group., "Opencl." [Online]. Available: https://www.khronos.org/opencl/
[14] A. Kumar, V. Grupcev, Y. Yuan, J. Huang, Y. Tu, and G. Shen, "Computing spatial distance histograms for large scientific data sets on-the-fly," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2410–2424, 2014.
[15] V. Grupcev, Y. Yuan, Y. Tu, J. Huang, S. Chen, S. Pandit, and M. Weng, "Approximate algorithms for computing spatial distance histograms with accuracy guarantees," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 9, pp. 1982–1996, 2013.
[16] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Procs. ACM Intl. Conf. Management of Data (SIGMOD)*, ser. SIGMOD '04, 2004, pp. 215–226.
[17] R. Ponce, M. Cardenas-Montes, J. J. Rodriguez-Vazquez, E. Sanchez, and I. Sevilla, "Application of gpus for the calculation of two point correlation functions in cosmology," in *ADASS XXI (Paris, 2011) conference proceedings*, 2012.
[18] "Nvidia geforce gtx 1080 whitepaper," NVidia Developer Technology, Tech. Rep.
[19] "Nvidia's next generation cudatm compute architecture:fermi," NVidia Developer Technology, Tech. Rep.
[20] "Nvidia's next generation cudatm compute architecture:kepler gk110," NVidia Developer Technology, Tech. Rep.
[21] *NVIDIA. GTX 980 whitepaper*.
[22] *NVIDIA GeForce Tesla V100 Whitepaper*.
[23] 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, pp. 194–205, 2011.
[24] *NVIDIA. CUDA C Best Practices Guide, version 7.5*.
[25] *Analyzing GPGPU Pipeline Latency*, 2014. [Online]. Available: http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf
[26] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, 2010, pp. 235–246.
[27] J. Wang, X. Xie, and J. Cong, "Communication optimization on GPU: A case study of sequence alignment algorithms," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, 2017*, 2017, pp. 72–81.
[28] 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.
[29] D. Bloom, "A birthday problem." *Am. Math. Mon. 80*, pp. 1141–1142, 1973.
[30] "2BS Framework." [Online]. Available: https://github.com/napath-pitaksirianan/2-bodyFramework
[31] R. Rui, H. Li, and Y. Tu, "Join algorithms on GPUs: A revisit after seven years," in *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, 2015, pp. 2541–2550.



Napath Pitaksirianan is currently a PhD candidate in the Department of Computer Science and Engineering, University of South Florida. He received an MS in computer Engineering from University of South Florida in 2015, and his B.E. in computer Engineering from Mahidol University, Thailand, in 2012. His current research is in parallel and distributed computing.



Zhila Nouri is currently a PhD candidate in the Department of Computer Science and Engineering, University of South Florida. She received her Master's degree in Software Computer Engineering from University of Isfahan, Iran, in 2009, and her Bachelor's degree in Software Computer Engineering from Azad University of Najaf Abad, Iran, in 2006. Her current research is in big data, parallel and high performance computing.

Yi-Cheng Tu received a Bachelor's degree in horticulture from Beijing Agricultural University, China, and MS and PhD degrees in computer science from Purdue University. He is currently an associate professor in the Department of Computer Science & Engineering at the University of South Florida. His research interest is in energy-efficient database systems, scientific data management, high performance computing and data stream management systems. He received a CAREER award from US National Science Foundation (NSF) in 2013. He is a senior member of both IEEE and ACM.

TC Responses to reviewers' comments and suggestions

We would like to extend our gratitude to the reviewers for their time and valuable comments. In this paper, we made the following revisions:

1. We reorganized the paper structure by grouping all experimental results into one section.
2. We implemented vectorized memory access and evaluated it with experiments.
3. We rerun all experiments on a Titan XP GPU and an Intel Xeon CPU, both represent a newer generation of the hardware.
4. We shrank and eliminated various parts of the previous draft that are considered less important.

We believe we addressed all concerns and comments made by the reviewers, and corrected typos as pointed out by the reviewers and found in our proofreading process. The following is our responses to individual comments.

Reviewer 1

[A1] I strongly recommend to compare against the best available libraries in the RDF example. Specifically, against the NAMD's GPU implementation in addition to VMD.

Response: *We have looked into NAMD code and also consulted our collaborators in the molecular simulations field. What we found is that NAMD does not have its own RDF code. Instead, it calls the relevant functions in VMD for such computations. Thus, we believe VMD still represents the state-of-the-art in this topic. Plus, it is the only work with a published article describing its design.*

[A2] Line 55, col 2, page 5: "For kernel parameters, we set the total number of threads as data size, and the value of threads per block to 1024, which is derived from an optimization model developed in our previous work [25]". I suggest that this optimization model should appear (summerized) here. I am very surprised that the maximum performance is achieved when using 1024 threads per block. Although warp occupancy is maximized, block parallelism is often reduced due to SM common resources consumption.

Response: *In our previous draft, the 2PCF kernel uses only a small number of registers and very little shared memory. That is why we can configure 1024 threads per block for running the kernel. Our experiments clearly show that a block size of 1024 achieves the best performance. In our revised draft, when we store input data in vectors of type float2, the consumption of registers increases and we found 512 is the best block size for the 2PCF kernel.*

A take-home message we wish to deliver is: code optimization in CUDA needs to consider many factors. As a result, rules-of-thumb based on a single factor often do not work the best. This was the main reason why we worked on a quantitative performance model in our previous work.

[A3] There are several important references from previous authors' work that appear in the conference paper but not here, or directly missing.

Response: *We cited a couple of representative papers that are relevant to this topic in this revision. Those are refs [14] and [15].*

[A4] Other relevant references for this paper:

* R.Ponce, M.Cardenas-Montes, J.J.Rodriguez-Vazquez, E.Sanchez, I.Sevilla Application of GPUs for the Calculation of Two-Point Correlation Functions in Cosmology Astronomical Society of the Pacific, Conference Series Volume 461, P. 73-76, 2012

Response: *We cited this paper in the revision. Although the paper is directly related to ours, there is essentially no way for us to compare with their work because the paper focused on the application rather than on the computational methodology. They only said a few words about their algorithm design and their code is not available.*

[A5] Rather than evaluating each output strategy separately in Section 4.4, it would be interesting to use the same problem example (SDH or item similarity) in the three strategies and compare the achieved performance for different problem sizes.

Response: *We selected different problems to test different output strategies because each output strategy is designed for a particular type of 2-BS with unique output behavior. We used SDH for Output privatization and Advance output privatization, but the direct output buffer is geared towards problems with unknown output size therefore we used item similarity as a sample problem.*

[A6] In section 3.2, it would be interesting to see a "Type-IV: output is too big for a single GPU [global] memory and several GPUs have to be employed". Have authors considered how to extend their solution to solve this kind of problems?

Response: *In this paper, we set the scope of work to be handling data in a single GPU device. Large data input/output is obviously a meaningful extension of our work. It however involves very different techniques (i.e., network traffic scheduling) as the I/O between GPU and host becomes a bottleneck. We are actually working on such optimizations in the context of relational joins (which can also be seen as a 2-BS problem). Due to the complexity and magnitude of such work, it is only reasonable to report it in another working paper.*

[A7] Line 53, col 1, page 4: "each thread loads one datum into the cache to ensure coalesced access to global memory". While SM resources available, I suggest authors to load several data in float4/int4/double4 or float2/int2/double2 CUDA datatypes. Coalescing is still present and they can reduce significantly the number of memory transactions.

Response: *We very much appreciate the reviewer's suggestions. In this revision, we have studied this and the details can be found in Section 4.1. The results show that using float2 reduced the memory transactions thus led to better performance than using float. However, when we try float3/4 the performance gain diminishes as the register consumption increases (details in Section 5.1).*

[A8] Reference 26 is quite old and outdated.

Response: *We have removed that reference and cited CUDA thrust library instead.*

[A9] An interesting proof may be compiling the naive algorithm with -Xptxas-dlcm=ca. This flag enables the use L1 cache. Note that whereas the CPU L1 cache is designed for spatial and temporal locality, the GPU L1 is just optimized for spatial. Frequent accesses to a cached L1-memory location does not increase the probability of hitting the datum, but it is attractive when several threads are accessing to adjacent memory spaces.

Response: *The –Xptxa-dlcm=ca was enabled in all our experiments. We did not mention that in our previous draft but added a sentence in Section 5.2 in the revised draft.*

[A10] Line 57, col 1, page 7. "we read and write to global memory without atomic instructions in the reduction stage. Atomics writes are done in shared memory, which bears a much lower overhead". Please, observe that GP100 improved this by providing an FP64 atomic add instruction for values in global memory (rather than a compare-and-swap loop, which is generally slower than a native instruction).

Response: *Our philosophy is to allow efficient computing of 2-BS problems in a wide range of hardware, especially those that can be easily acquired. The P100, with its $6000 price tag, is not our target platform. This was particularly important for making a fair comparison between CPUs and GPUs - we chose devices that are of (roughly) the same retail price. That was the reason why we picked GTX Titan and Titan XP for our experiments.*

As a side note, we did conduct experiments on P100 to verify the performance of 64-bit atomic add. It is true that the FP64 instruction greatly improved the performance of atomic operations. As a result, our kernel with output privatization, Register-ROC-out, is only about 2% faster than one that uses the FP64 instruction. Such results could be interpreted in different ways, but we believe this shows the privatization technique, as a software solution, did an even better job in bypassing the performance barrier of atomic operations than the hardware solution of using FP64.

[A11] Section 4.3.2, "our idea is to assign threads with the same offset of ID in the warp to an output copy [..] Each private output is shared by threads that have same lane ID". Another approach to reduce the number of atomics would be warp-aggregated atomics (https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/)

Response: *We were aware of this technique and there are other articles reporting its success. The idea behind warp-aggregated atomics is using a leader of a warp to perform the atomic operation. This technique improves performance when many threads are expected to modify one single memory location. However, for Type-II 2BS, each thread may update any shared location in memory. That is why warp-aggregated atomics cannot apply to our problems.*

[A12] Section 4.3.2, have authors considered the "tail effect" in their performance model when calculating the number of rounds algorithm takes? If yes, it should appear in text, otherwise it should be analyzed.

Response: *If by "tail effect" the reviewer means the time spent to run the "last" round, which may not contain enough blocks to occupy all multiprocessors, we believe it is taken care of by our model – Equation (1) has a ceiling function in it so we are not ignoring the last round. Note that the latency of the last round, even if there are idling multiprocessors, should not be any different from that of the previous rounds.*

[A13] Line 34, col 1, page 8, "occupancy is affected by the use of shared memory for each block". Register consumption should be mentioned as a limiting factor in occupancy too.

Response: *We did not mean to make a general statement. The context was to compute Type-II 2-BS, in which the register utilization is very low comparing to shared memory consumption. Since the*

register is never a bottleneck, we just mentioned share memory. In this draft, we have updated this sentence (Section 4.3.2) for better clarity.

[A14] Authors should explain in greater detail the reason of the big difference between theoretical and actual results for H=1 in Table 4.

Response: *Our model is probabilistic and there are low-order dynamics we did not capture. These include the conflict-free latency and uncertainties in the actual conflict in the atomic operations, which depends on the input data. It is true that we started to pick H1 later (starting from output size 450) than expected. If we take a closer look at such cases (Fig. 11), the performance difference between H1 and H2 is very small (i.e., less than 5%) under large output sizes. We updated the draft with such descriptions.*

Reviewer 2

[B1] My personal opinion is that the methodology and the experimental results should not be presented in a mixed way. It makes the presentation flow less clear and fragmented.

Response: *We agree, therefore in this revision we separated the methodology and the experimental results. Section 5 now has all the experimental results.*

[B2] The article presents, in many points, memory latencies and bandwidth values that are specific to a single GPU device (Global Memory - 350 cycles, shared memory - 28 cycles, etc.). The authors should consider that, currently, the GPU devices present very different specifications and characteristics. In addition, many of these values presented in this work refer to reference [23], which is outdated (2010).

Response: *Since NVidia never published "official" latency, we depend on 3rd party experiments and publications. Those were the best results we could find at the time. In this draft, we cited a very recent paper published in IPDPS 2017, which reported latest results about on-chip memory latency of the NVidia Titan X Maxwell card. Indeed, the latency is lower in all memory systems for the newer cards, but it does not affect our algorithm design as the (orders of magnitude) different between shared memory and global memory still exists.*

Background:
[B3] This section should point out which GPU architectures/models support the read-only cache.

Response: *We added this in the revision. (Section 3.1)*

[B4] Methodology:
- The tiling technique presented in Section 4.1, second paragraph, is quite common at the state-at-the-art and not novel. Since the work is clearly focused on performance, the author should consider to apply other optimizations like loading more items per thread into the shared memory instead only one, and also to consider the occupancy/available shared memory trade-off in addition to a straightforward approach.

Response: *We tested the technique of loading multiple input data points by float2/3/4. See our responses to comment A7 for details.*

[B5] Section 4.1, fourth paragraph: The CUDA programming language does not provide the "register" keyword. All variables declared in the kernel scope are placed into registers by default, or eventually spilled in the L1 cache if the available resources are not enough.

Response: *We apologize for the wrong statement we made. We have removed such from our revision.*

[B6] In general, Section 4.1 proposes trivial techniques that serve as a baseline for other optimizations. The authors dedicated too much importance to these concepts that can be presented in a more concise way.

Response: *We agree, we significantly shrank Section 4.1 (Section 4.2 in new draft).*

[B7] Section 4.2. Table II reports very low global memory bandwidth. Given the simple problem formulation, the authors should consider adopting additional techniques to improve this aspect such as vectorized memory accesses and data-level parallelism. Furthermore, the arithmetic units utilization can be improved by using instruction-level parallelism (ILP) http://heather.cs.ucdavis.edu/~matloff/158/JO171.pdf).

Response: *We agree, we accommodated vectorized memory access and ILP in our efforts of using float2/3/4 to load data. See our responses to comment A7.*

[B8] The presentation of different techniques in Section 4.2 shows a confusion about the understanding of the CUDA programming model. The shared memory is reserved for storing data when its indices can be easily computed (as in the 2-body problem), while RoC should be used when the access pattern shows good locality but cannot be predicted.

Response: *Yes, we understand what desired access patterns the shared memory and RoC were meant to be used for. However, the interesting thing about this paper is that we show there are not many "golden rules" in CUDA code optimization – same for what RoC should be used for. As we can see, the 2PCF experiment (which is type-I of 2-BS) in Section 5.2 shows very good results by using shared memory. However, in Section 5.3 (type-II of 2-BS), we demonstrated that tiling by using RoC can gets better performance. This happened because using shared memory to cache output is more beneficial than using it for caching input data. As a result, the use of RoC to cache input shows its benefits by freeing the shared memory, even if access patterns to such inputs are not a perfect fit to RoC.*

[B9] Although the analytical model is still useful, many conclusions can be easily identified (global memory vs. shared memory) or computed with few runs on small benchmarks.

Response: *We agree, we removed this analytical model in this revision.*

[B10] Section 4.3. The methodology presented in this section presents some flaws:
- The "data output phase" basically falls in building a histogram of the evaluated distances. Several techniques have been proposed in the literature to efficiently compute histograms of arbitrary sizes. I'd suggest the authors to take a deep look at the following article, which addresses the problem in details: S. Ashkiani, A. Davidson, U. Meyer, J. Owens, "GPU multisplit, PPoPP'16

Response: *In our paper, the data output phase is not only focused on building histograms - we consider all kinds of outputs (e.g., tabular output from relational database join, scalar from 2PCF).*

We read the Ashkiani paper and learned a lot. However, we did not find anything that can be directly applied to the 2-BS problems. Even though the partial output of a multisplit is a histogram, the problem is still fundamentally different. In their computation of histogram (warp level histogram), they take multiple buckets that contain items as inputs. Then, they assign each thread in a warp to computing a particular bucket of the histogram. Each thread counts the item in a single bucket and updates the count locally. Finally, they copy local output to global output. This technique seems sound, but cannot be applied to 2-BS problems. Because, all of our threads have to calculate pairwise distances and the target bucket to modify is unpredictable. It is not possible to assign a particular thread to handle a single bucket of histogram.

[B11] In the "atomic writing stage" (fig 4.), the threads involve very scattered memory accesses (column-major order). I'd suggest the authors to consider additional techniques to improve this aspect or to apply a matrix-transpose to the output and then assign a warp for each row to compute the reduction.

Response: *We did not put a lot of efforts into optimizing this stage as it only corresponds to about 3% of total running time. In this revision, we changed atomic writing pattern to a row-major order, which is coalesced access. The change is made in Section 4.3.*

[B12] [26] is a very old reference for the reduction procedure. The CUDA C programming guide, CUB and ModernGPU libraries are definitely better and should be considered as more recent references.

Response: *In this revision, we removed this reference and cited CUDA thrust library instead.*

[B13] The work should compare the execution time of "output privatization" and "advanced output privatization" and better explain when one procedure is preferred over the other one (and not only in the experimental results).

Response: *The single histogram (1H) refers to regular "output privatization". Our analytical model in Section 4.3.2 tells us exactly when we should select output privatization (1H) and when to select advanced output privatization (2H-32H), as described in the last paragraph of Section 4.3.2.*

[B14] Section 4.3.1. Algorithm 4. The pseudocode should include synchronization barriers and atomic operations to guide a correct implementation.

Response: *In this draft, we have added synchronization barriers and mentioned atomic operations in all pseudocode.*

[B15] The "direct output buffer" technique is not novel. The idea has been widely used in other contexts such as graph algorithms to represent a queue updated in parallel. Examples:

- L. Luo, M. Wong, W.M. Hwu, "An effective GPU implementation of breadth-first search"
- D. Merrill, M. Garland, A. Grimshaw, "Scalable GPU graph traversal"
- F. Busato, N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures"

Response: *We thank the reviewer for the pointers, yet we humbly disagree that any of the three papers mentioned the same idea as ours.*

L. Luo et al. use Hierarchical Queue to handle output. They use on-chip memory to buffer the output, which reduces atomic operation on global memory. Then, at the end, they copy output to

global memory. Their solution does not tell exactly how to handle big output data, i.e., when output is bigger than on-chip memory. On the other hand, our solution allocates a page for each thread. If the page is full, the thread can request a new page without terminating the grid.

The other two papers use prefix sum to find the outputting location for each thread in a large output buffer. That is exactly the kind of approach we were trying to improve on. We can find the output location for each thread using a prefix sum, but to compute 2-BS following that idea, we have to run the kernel twice, and our experiments have shown that the direct output buffer approach is much better in terms of performance.

[B16] Section 4.5.2. Why "tiling with shuffle instruction" should be used instead of the shared memory approach as it provides lower performance?

Response: *The performance of tiling with shuffle is the same as or slightly better than the shared memory approach. Although this paper assumes the 2-BS kernel is the only application running in the GPU device, there is an increasing number of projects that consider running multiple grids / kernels concurrently (via CUDA Streams, for example). In such systems, a system mechanism selects grid parameters and even kernels (code) for each concurrent task towards a global optimization goal (e.g., total workload processing time). In such systems, the "tiling with shuffle instruction" kernel can be used to free up the shared memory and/or RoC for the use of other tasks.*

[B17] Section 5. "as anything larger than that cannot be placed in register in CUDA". The CUDA model supports arbitrary data type size and structures.

Response: *It is true that any data has to be in registers before the core can work on it. However, in CUDA, any data larger than something that can be defined as a primitive type (i.e., not arrays) will be placed in global memory. We have updated this sentence in Section 6 of this draft.*

[B18] Section 5.1. "This shows that 2-BS framework can automatically generate kernel with very high performance with very little effort from the developer". Is the code publicly available?

Response: *Yes, we have included our Github link in this revision (Ref [30]).*

Reviewer 3

[C1] Overall, this manuscript has enough addition to the original conference paper. The effort to evaluate the proposed approaches in both analytically and experimentally is impressive. However, the contribution of the proposed approaches looks weak. For example, this reviewer was able to come up with the 'register + shuffle instruction solution' and 'constant memory solution for the intra-block computation that also can take advantage of broadcasting' before reading section 4 and found that the authors proposed exactly the same solutions.

Response: *Let us share our experience in working on this seemingly 'simple' problem of 2-BS. Like the reviewer, we certainly had our hypotheses before we try different solutions. However, some of them turned out to be not the best, and that surprising factor is one of the reasons why we seek publication of this work, as readers could benefit from our findings (instead of relying on their intuition). The approaches proposed by the reviewer are very good examples of such – they sound reasonable but was not be the best. In fact, the 'register + shuffle instruction solution', as clearly shown in our paper (Section 5.4), is slower than the best solutions using on-chip cache, although it can be an alternative solution when all on-chip memory is in use. As to the second solution, we*

want to clarify that our solution does not depend on constant memory – we used Read only data cache instead and those are 2 different types of cache.

[C2] This reviewer didn't check related papers but the proposed approaches look straightforward for those who have basic CUDA programming knowledge. Privatization has been widely used for many algorithms so not a new idea. The advanced privatization method might be new but still doesn't look like a significantly novel invention.

Response: *We can only ask the reviewer to trust us that a thorough literature search has been done and the only published work in this topic we found were about individual 2-BS problems – these include the RDF code in VMD and various relational join work, as studied in our paper. Here we tend not to argue about the novelty. We just want to point out that our solutions beat those in performance with a big margin. Note those work were all published in decent venues, including J. Computational Physics, SIGMOD, and IEEE Big Data.*

[C3] The direct output buffer approach needs more explanation. The algorithm 6 shows only the distance computations by using as many pages as the number of pairs. Where is the summarization part (i.e. finding the pairs that have similar distances)? And, it is hard to understand how the mapping information between new page and thread is maintained (it seems that the mapping information is needed to finalize the similar distance pairs). The proposed page level buffer management doesn't look efficient; if its only purpose is to handle dynamic size heap allocation, "malloc" is also supported for CUDA architectures that are under the compute capability 2.x or higher.

Response: *The direct output buffer is a technique specifically designed for the output stage – that is the reason why we ignored details of the pairwise computation stage in Algo. 5 (Algo. 6 in old draft). In the pseudocode, the "x" is an abstract of any kind of output from each pairwise computation – the main problem here is how to output each "x." The mapping between thread and output page is given by the GP pointer. We modified the pseudocode in this draft to hopefully clarify the idea.*

The reviewer seems to have misunderstood the purpose of having the direct output buffer. The problem of malloc is that you need to know exactly how much memory to allocate yet in Type III 2-BS this can never be known beforehand, and that is the problem we are trying to solve. Even in problems where the maximum output of each thread is known, malloc is rarely used due to its enormous latency penalty and the code divergence it brings. In terms of efficiency, our solution based on Direct Output buffer delivers nearly a 2X speedup comparing to traditional solutions with a 2-pass kernel execution.

[C4] It is suggested for the authors to add more reference papers (recent papers after 2013) or even CUDA programming manual and teaching materials and clearly explain the contribution of their approaches (NVIDIA has been publishing lecture slides and books that actually include very good optimization techniques for various algorithms.)

Response: *Please see our responses to comment C3.*

About the techniques and programs provided by NVidia, our experience (and that of many other colleagues) is mixed – some are very well optimized, others are not well done at all. In the past few years, the authors have dedicated a lot of efforts into GPGPU research and education therefore have high confidence in saying so. In the particular case of 2-BS problems, we did not find any code from the CUDA libraries.