**Noname manuscript No.**
(will be inserted by the editor)

# Algorithms and Framework for Computing 2-body Statistics on GPUs

**Napath Pitaksirianan · Zhila Nouri · Yi-Cheng Tu**

Received: date / Accepted: date

**Abstract** Various types of two-body statistics (2-BS) are regarded as essential components of low-level data analysis in scientific database systems. In relational algebraic terms, a 2-BS is essentially a Cartesian product between two datasets (or two instances of the same dataset) followed by a user-defined aggregate. The quadratic complexity of these computations hinders timely processing of data. Use of modern parallel hardware has thus become an obvious solution to meet such challenges. This paper presents our recent work on designing and optimizing parallel algorithms for 2-BS computation on Graphics Processing Units (GPUs). Although a typical 2-BS problem can be summarized into a straightforward parallel computing pattern, traditional knowledge from (general) parallel computing often falls short in delivering the best possible performance. Therefore, 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 finding the best parameters of our GPU programs. 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. We further present two case studies to demonstrate that code generated by this framework reaches a very high level of efficiency.

**Keywords** 2-Body Statistics · Parallel Computing · GPGPU · GPU · CUDA

Department of Computer Science and Engineering, University of South Florida
4202 E. Fowler Ave., ENB118, Tampa, FL 33613, USA
E-mail: {napath, zhila, tuy}@mail.usf.edu

## 1 Introduction

Handling analytical workloads efficiently is a major challenge in today's scientific domains. Recent studies show increasing interest in developing database systems for handling scientific data [1–4]. Traditional DBMSs still fall short of algorithm and strategies to satisfy the special needs of scientific applications, which are very different from those in traditional databases in their data types and query patterns. In addition, design of efficient algorithms for data query and analysis are still the main challenge in scientific areas. In addition, support of complex mathematical functions in DBMS have become an active research area. A good example is the integration of linear algebra with DBMS in a recent ICDE awarded paper [5]. In this paper, we are interested in a type of low-level analytics that is frequently used in various applications, namely, the 2-body statistics. Bearing many forms and definitions, 2-body statistics (2-BS) 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. Therefore, in relational algebraic terms, a 2-BS is essentially a Cartesian product between two datasets (or two instances of the same dataset) followed by a user-defined aggregate.

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 [6] also used quad-tree and a batching technique to reduce the complexity of SDH computing to $O\big(N^{1.5}\big)$. 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 problems, parallel computing techniques are extremely useful for two reasons: (1) particular types of 2-BS lack efficient algorithms. For example, kernel functions for Support Vector Machine (SVM) [7] and pairwise comparison in various applications [8, 9] can only be solved in quadratic time. Another example is relational joins – although sort-merge, hash, and index-based algorithms are preferred for processing equi-joins, 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 [10] 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 architec-

ture (CUDA) [11] and Open Computing Language (OpenCL) [12]. 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 7 for details). In addition to the surprisingly little attention paid to this topic, existing work lacks a comprehensive study of the necessary techniques to achieve high performance on GPUs. This is a non-trivial task because code optimization has to consider architectural and (system) software features of modern GPUs that are found to be more complex than those in 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 3.3.2), warp-level load balancing (Section 3.4.1), and shuffle-enhanced tiling (Section 3.4.2) are innovations not presented in previous work, to the best of our knowledge. Specifically, this paper makes the following contributions.

(1) 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 synchronization. As a result, we achieve the design of highly-optimized 2-BS algorithms that significantly outperform the best known GPU and CPU implementations;

(2) 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;

(3) We extend our basic algorithm design to the scenarios of input data being larger than global memory and utilization of multiple GPUs towards high scalability. This involves splitting the input into small blocks and pipelining data transmission with in-core computation;

(4) 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 introduce the technical background of our work; in Section 3, we demonstrate a suite of techniques to speed up pairwise computation and output writing on GPUs; we describe the 2-BS framework for automatic

**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 |
| $\zeta$ | Actual use of SM in a block |
| $C_L$ | Latency of writing without conflict |
| $C_{LP}$ | overhead latency of writing conflict |

code generation in Section 4; we evaluate our GPU algorithms in Section 5; we evaluate our automatic code generation framework by two case studies in Section 6; in Section 7, we review work related to 2-BS problems, and conclude this paper in Section 8.

## 2 Background

In this section, we give an introduction to the 2-BS problem and typical GPU architecture. Then, we introduce a straightforward GPU algorithm for computing 2-BS. This work focuses on NVIDIA GPUs and the CUDA framework, which are widely used for general purpose computing. The notations used in this paper are presented in Table 1.

### 2.1 2-Body Statistics (2-BS)

As we mentioned earlier in this paper, we refer to 2-BS as a computational pattern that evaluates all pairs of points among an N-point dataset. This computation pattern can be done either within a single dataset, or between two datasets. For every pair of data points, a 2-BS computes a distance function between the pair. By iterating over all of the pairs, the problem can be solved with total complexity $O(N^2)$.

The 2-BS type of functions are found popular in many scientific domains, with numerous concrete examples. The following are names of 2-BS that compute all pairs of points within a single dataset: 2-tuple problem [13], all-point nearest-neighbor problem [13], pairwise comparisons for time series motifs [14], nonparametric outlier detection and denoising [13], kernel density regression [13], two-point angular correlation function [15], 2-point correlation function [13], and spatial distance histogram (SDH) [6], to name a few. Another flavor of 2-Body statistic takes two different datasets as input, examples include Radial distribution function (RDF) [16] and relational joins [10]. Table 2 lists some examples of 2-BS.

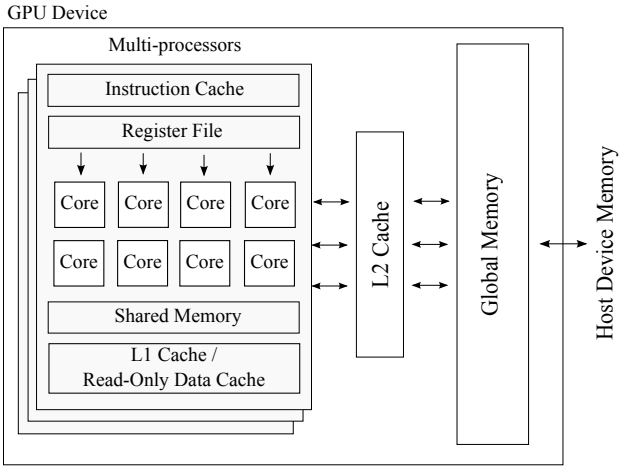


**Fig. 1** Architecture of a recent Nvidia (e.g., Maxwell, Pascal) GPU

In practice, there are many applications of 2-BS problems. A common practice in many application domains 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 recommendation systems for online advertising that predicts the interest of customers and suggests correct items. Jensen *et al.* reports a music predictive model [17] 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) [8,9]. Both require 2-BS computation: CB depends on pairwise comparisons between items and CF depends on those between users.

## 2.2 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-related computation. 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* built on high-speed memory technology such as GDDR5 and HBM2. 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 900 GB/sec [18]. Each multiprocessor also provides high-speed programmable *shared memory* of size 64-96KB. 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

**Table 2** A list of some 2-BS problems

| Type | 2-BS Problem Name | Output Description |
|---|---|---|
| Type-I | 2-point correlation function | Counting the number of pairwise distance that is less than a constant $r$ |
| | Kernel density regression | A statistic value that is defined by $K\sum_i w(\|x - x_i\|)$ where $K$ is a normalizing constant and $w$ is a weighting function |
| | Two-point angular correlation function | The probability of finding an object within a given angular distance |
| Type-II | Spatial distance histogram (SDH) | A histogram that shows the distribution of all pairwise distances within a set of atoms. |
| | Radial distribution function | SDH normalized by density of the system multiply by volume of the spherical shell |
| Type-III | Relational joins | List of tuple pairs that satisfy a given condition |
| | Similarity of all pairs | List of data point pairs that are similar according to a given distance function |
| | Recommendation systems | List of items that are related to (many) users |

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 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*) are guaranteed to run at the same time. Each warp contains 32 threads with consecutive thread IDs. 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[22], and Volta [18]. Newer architectures provide more computing resources. Moreover, new functionalities and features in the CUDA framework have been introduced over the different generations of GPUs. 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 via a mechanism called *dynamic parallelism*. Such features provide new opportunities for improving program efficiency and also impose extra challenges to algorithm design and implementation.

**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**

### 2.3 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 concurrent access to global and shared memory locations. 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.

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

[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.

registers for each thread. For example, 2-point correlation function [13], 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[13], 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) [6], which outputs a histogram of distances between all pairs of points; (2) Radial distribution function (RDF) [16], 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 [10], 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 [7].

Table 2 shows information of various 2-BS examples, and the category each example belongs to.

## 3 GPU Algorithms Design

In this section, we elaborate on the GPU algorithm design. First, we discuss input data representation of the GPU system. After that, we focus on pairwise computation which focuses on loading data in to multiprocessor via caching. Then, we explain techniques to handle the output from each thread, and introduce additional techniques that can be used by our algorithm in special cases. Finally, we present an algorithm for processing 2-BS problems on multiple GPUs.

### 3.1 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

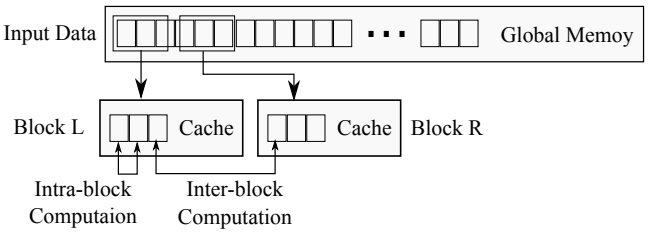


**Fig. 2** Tiling method requires loading data in blocks

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).

### 3.2 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.

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

**Algorithm 2:** Block-based 2-BS computation

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

response time of just a few clock cycles [25–27]. 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. Let us call this baseline technique as *LRshm*.

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, and each thread only accesses a particular datum throughout its lifetime. Therefore, it is more efficient 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 3.3). 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 another technique, named *Register-SHM*, which improves *LRshm* 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 technique we use 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 by storing the block in RoC. 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 3.3). 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.

## 3.3 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 synchronization required by supporting different threads write into the same memory location. To avoid incorrect results, 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 when threads frequently access the same memory address. For that, we present a *direct output buffer* (Section 3.3.3) mechanism to minimize such costs. Note that our paper focuses on 64-bit output data type.

### *3.3.1 Output privatization*

Data *privatization* is frequently used in parallel algorithms to reduce synchronization [15]. 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

**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: $Output[b][t] \leftarrow SHMOut[t]$

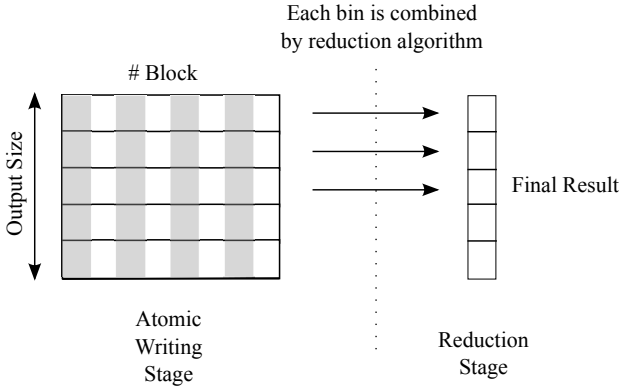


**Fig. 3** Combining private outputs in all blocks to obtain the final result

a parallel *reduction* algorithm such as the one presented in CUDA thrust library. Algorithm 3 shows a new version of Algorithm 2 enhanced by the output privatization technique.

In our initial implementation, we use one private copy of the output for each thread block. By this, synchronization 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 3.3.2. 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.

**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

### *3.3.2 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 synchronization still exists when different threads in the block write into the same output address. If the output size is small enough to allow multiple 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 determine the exact 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 even-numbered IDs in all warps to share the first output copy and those with 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

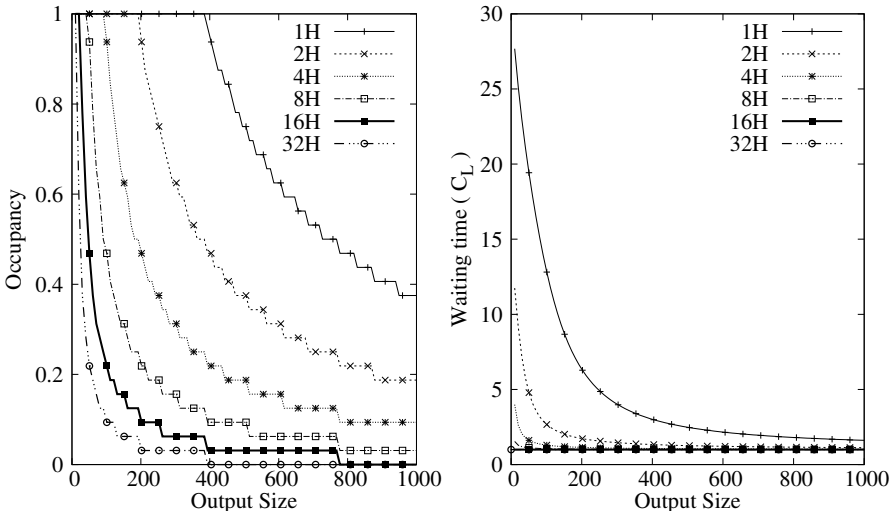


**Fig. 4** Modeling results under different numbers of private copies and sizes of the output. Left: thread occupancy as given by the model developed in [28]; Right: latency given by our model

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, $\alpha$ 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 small number of (i.e., 30 in a Titan XP) 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. Based on such results, we now develop the analytical model.

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 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

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

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:

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

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:

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

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 becomes 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 find one that leads to the best running time.

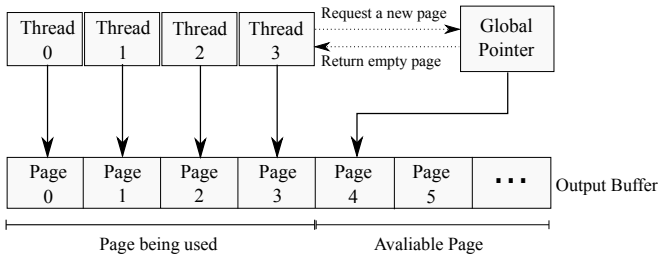


**Fig. 5** A case of direct output buffer for GPU threads, showing Thread 3 acquiring a new page (i.e., page 4) as its output buffer

### *3.3.3 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 [10] 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 take advantage of a *buffer management* mechanism proposed in our previous work on GPU-based joins [30] to handle unknown output size. This design only requires one single run of the kernel, with very little synchronization among threads. Figure 5 demonstrates the mechanism. 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. 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 algorithm augmented with this mechanism. The algorithm starts from initializing local buffer pointer by using *atomic add* operation from global buffer pool (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). Here we want to point out that the GP pointer, although defined in global memory, will be most likely cached at runtime therefore its accesses generate very little overhead.

## 3.4 Additional Techniques

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

**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: $buf \leftarrow$ atomicAdd($GP$, $b$)
2: $count \leftarrow 0$
3: **for** each pair of input datum **do**
4: $x \leftarrow$Pairwise Distance
5: $buf[count{+}{+}] \leftarrow x$
6: **if** $count == b$ **then**
7: $buf \leftarrow$ atomicAdd($GP$, $b$)
8: **end if**
9: **end for**

### *3.4.1 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, we can 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 6). Here, we introduce a *load balancing* method to eliminate divergence from the intra block computation. As we mentioned before, divergence occurs because the workload on each thread is different. Our technique thus enforces each thread to compute the same amount of work, i.e., half of 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, and this leads to divergence everywhere. With the load balancing technique, we let each thread pair with $B/2$ datum. In particular, at iteration $j$, the thread with index $i$ pairs with datum with index $(i+j)\%B$. Figure 6 illustrates the main idea. Note that, in the last iteration, only the lower half of all threads in a block needs to compute the output. This does not cause a divergence as the block size is a multiple of warp size.

### *3.4.2 Tiling with Shuffle instruction*

As seen in Section 3.2, 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).

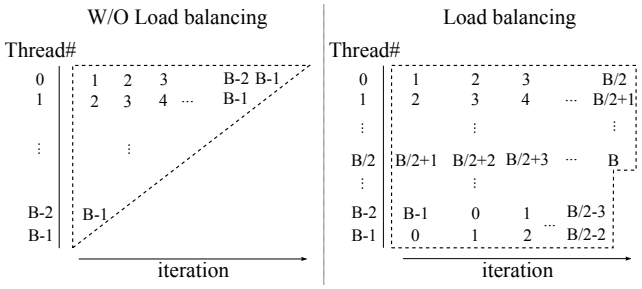


**Fig. 6** Two different ways of work assignment to threads in intra-block pairwise computation

**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
```

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 load 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 7 shows an example. Note that this method requires only two more registers and does not require shared memory or read-only cache.

### 3.5 Dealing with Large Data Inputs

So far, we have presented solutions to the 2-BS problem assuming the GPU global memory is big enough to contain all input data as well as program states. In fact, the 10GB-level global memory in a typical GPU can hold input data big enough to keep the GPU busy for a long time due to the quadratic nature of 2-BS computing. However, it is still meaningful to study algorithms that

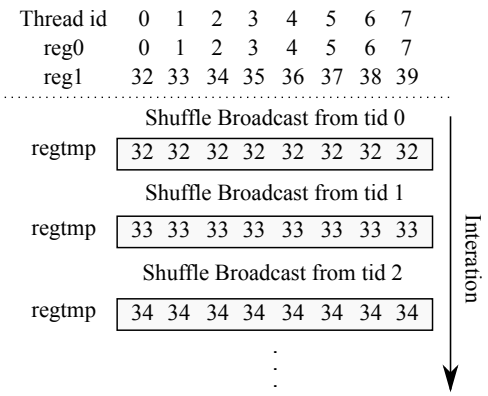


**Fig. 7** Tiling with shuffle instruction technique

handle large datasets with sizes greater than that of the global memory. In this section, we present an advanced algorithm that processes 2-BS on arbitrarily large data inputs.

In our algorithm, we first divide the input data into small blocks that can fit into GPU global memory, and execute the best algorithm we have developed so far for each pair of input data blocks sitting in global memory. The results of each pair of data blocks are then aggregated towards computing of the final result. It is well known that data transmission in/out GPU carries significant overhead due to the limited bandwidth of the PCI-E bus.[2] In our small-data algorithms, the input data only needs to be shipped into the GPU once, this translates into a linear overhead that is easily overshadowed by the quadratic on-board computational time. With the large data inputs, every pair of data blocks will have to be transmitted to the GPU. If the data has to be partitioned into $k$ blocks, we essentially have to ship $O(k^2)$ pairs of blocks. Therefore, a major optimization goal here is to reduce the data shipping overhead.

Our strategy is to hide the data transmission latency behind the GPU processing of the *in situ* data blocks. In CUDA, data transmission in/out of GPU is asynchronous therefore execution of concurrent kernels allows data transmission and in-core computation to be done at the same time. The mechanism for concurrent kernel execution is called *CUDA Streams*: a CUDA stream is basically a series of GPU operations (e.g., I/O, kernel execution) that have to follow a fixed order. However, kernels belonging to different CUDA streams can be executed concurrently: when input data is being transmitted in Stream 1, the 2-BS kernel in Stream 2 can run. Figure 8 illustrates this idea. For each pair of input data blocks, if the kernel running time is longer than data transmission time,[3] the latter can be effectively hidden.

[2] The NVlink bus found in newer GPUs provides a higher bandwidth but does not fundamentally change the fact that data transmission is the bottleneck.

[3] It is easy to find a block size to satisfy this condition due to the quadratic computational time.

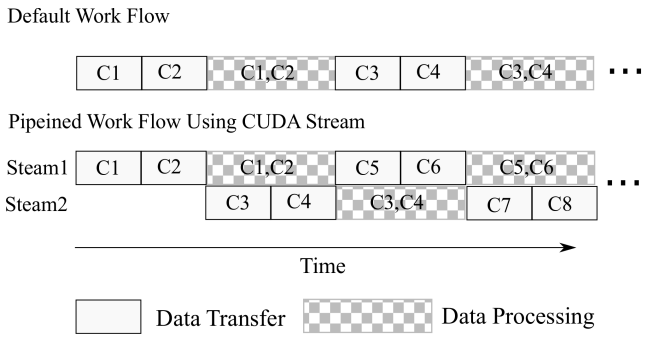


**Fig. 8** Pipelining data transmission and kernel execution via CUDA streams. Here C1, C3, C5, C7 represent blocks of dataset A and C2, C4, C6, C8 are those for dataset B. For simplicity we ignored the output data transmission, which can also be pipelined in the same manner

Apart from allowing 2-BS to be computed for large data inputs, the above approach provides an effective way to scale up the algorithm. Since all pairs of input data blocks can be processed independently, our method can be easily applied to a multi-GPU environment, in which each GPU can work on a different set of input data blocks.

A special note here is: the way to combine outputs from different block-level runs depends on the type of 2-BS. For Type-I and Type-II, we combine the results inside each GPU via parallel reduction when any pair of data blocks are processed. For Type-III, due to the use of direct output buffer, no action is needed to combine output within the GPU. To handle new chunks output, threads just require to acquire a new page from the Global Pointer. After GPU devices complete computation of all data blocks, they transfer output back to the host. At the end, the host combines all device-level outputs into the final output.

## 4 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 of different 2-BS problems. For users with new 2-BS problems with arbitrary characteristics, development of efficient code is still challenging. 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 new 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.

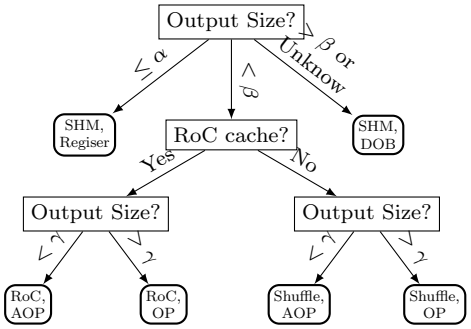


**Fig. 9** Decision tree of our 2-BS framework. Acronyms: SHM (cached input on shared memory), RoC (cache input on read only data cache), Register (cached output on register), DOB (Direct output buffer), OP (output privatization), AOP (Advance output privatization)

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 9. If output size is tiny or equal to a threshold $\alpha$ (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 $\beta$ 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 instructions will be used for caching input data. Then we check the output size again, if output size is greater than a threshold $\gamma$, regular output privatization will be used. Otherwise, warp-level output privatization will be used. The thresholds are set as follows: we set $\alpha$ 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 $\beta$ to the size of shared memory (i.e., 64K for Pascal); and $\gamma$ is given by the modeling results shown in Table 5.

We developed the framework with Python [31], and some implementation details can be found in Figure 10. For any 2-BS problem, the system takes as inputs an XML file holding all relevant problem-specific parameters and a CUDA file containing the distance function. The CUDA distance function is written as a regular C function. Blocks of CUDA code are pre-stored in a database and selected based on the aforementioned rules and modified and integrated into a file containing the CUDA kernel code as the output.

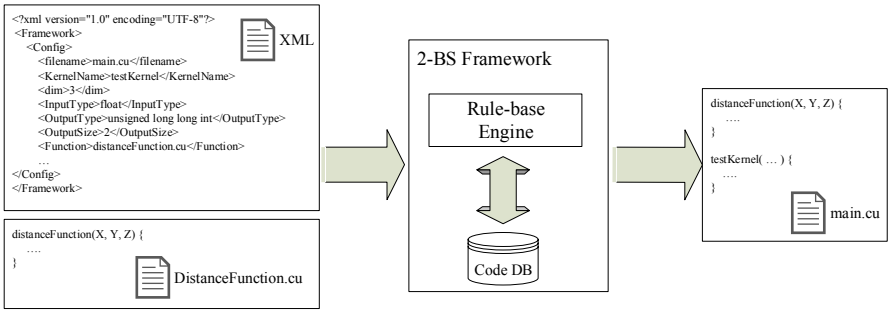


**Fig. 10** Implementation of the 2-BS code generation framework

## 5 Evaluation of GPU-Based 2-BS Algorithms

In this section, we present empirical evaluation of aforementioned algorithms. We run our experiments in a workstation running Linux (Ubuntu 14.04 LTS) with an Intel Xeon 6-core E5-2620 v3 CPU, 128GB of DDR4 1866-MHz memory, and an Nvidia Titan XP GPU with 12GB of global memory. The running time reported for all experiments is end-to-end latency, which includes time to transfer input data in and output data out of the GPU.

### 5.1 Data Representation Schemes

We first evaluate the 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 workload 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: 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[4].

Figure 11 shows the running time of all eight kernels. As we see, kernels that use Shared Memory show similar results by using Float2, Float3 and

[4] We also run experiments on skewed datasets. However, the performance of 2-BS algorithms is not affected by data distribution thus we omit those results.

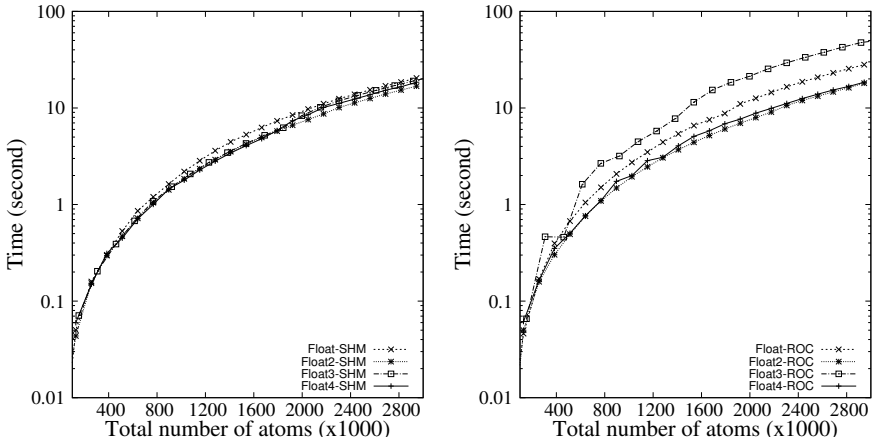


**Fig. 11** Performance of different data types of vectorized memory access for computing 2-PCF

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 performance because they significantly increased register use of kernel and thus 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

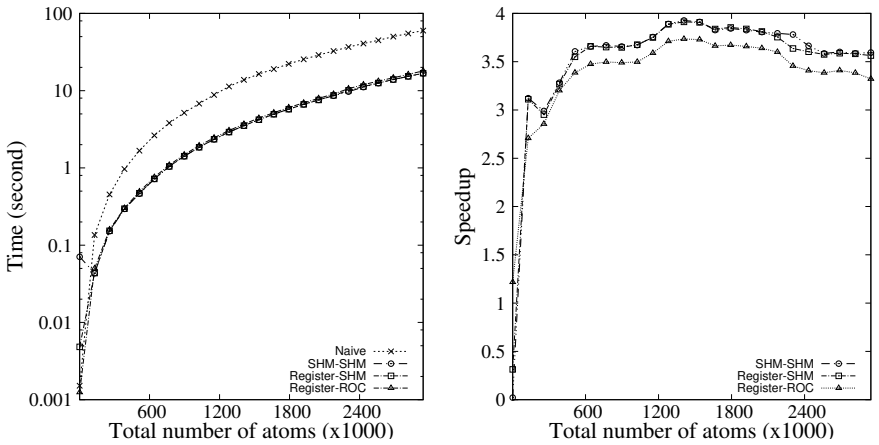


**Fig. 12** Performance of different GPU-based algorithms for computing 2-PCF: total running time and speedup over naive algorithm

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.

Figure 12 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

**Table 3** 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; ROC: Read-Only data cached

| 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% |

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 3 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 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, 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 thus can be placed in shared memory. On GPUs, due to the concurrent updating of output the problem is bound by memory bandwidth.

In this set of experiments, we compare six kernel functions: the first three are algorithms we studied in Section 3: Naive, Register-SHM, and Register-RoC. The output stage of those three algorithms is handled in a straightfor-

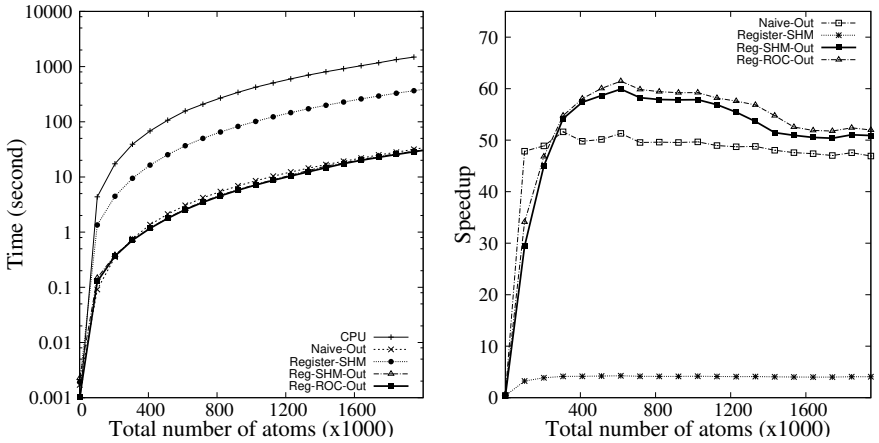


**Fig. 13** Performance of different GPU-based algorithms for computing SDH: total running time and speedup over CPU algorithm

ward 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 implemented 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. We also profile our multithread CPU code for computing SDH by pref tool. The profiler shows that the algorithm is bound by computation, uses all cores of the CPU, and makes good use of cache (cache miss rate is less than 2%). Therefore, our

**Table 4** 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; ROC: Read-only data cache

| 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% |

CPU code shows great spatial locality since most accesses are satisfied through cache. In summary, we believe our CPU code is of very high (if not optimal) performance.

**Experimental Results:** Figure 13 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 13. 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 4 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 14 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

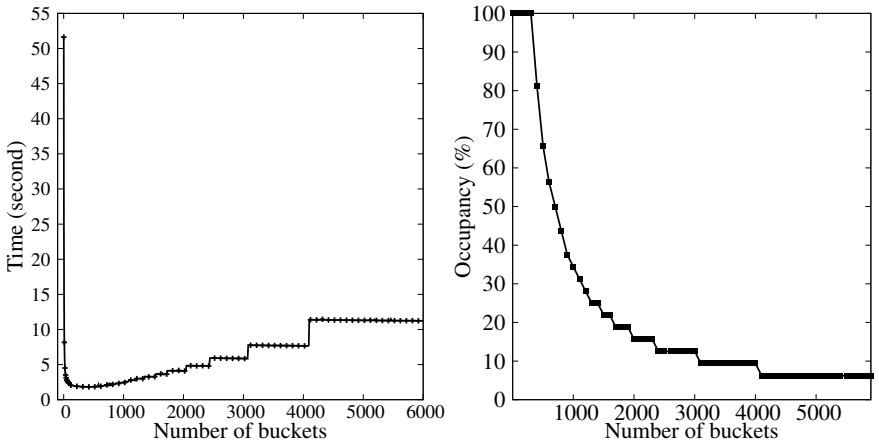


**Fig. 14** Performance of the Reg-RoC-Out kernel under different bin sizes: running time and occupancy

as a step function of output size. This is because the output size affects the performance via changing the occupancy of the kernel. Figure 14 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 technique 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 15 shows theoretical running time (left) obtained from our models shown in Section 3.3.2 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 can be easily found in Figure 15. Table 5 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

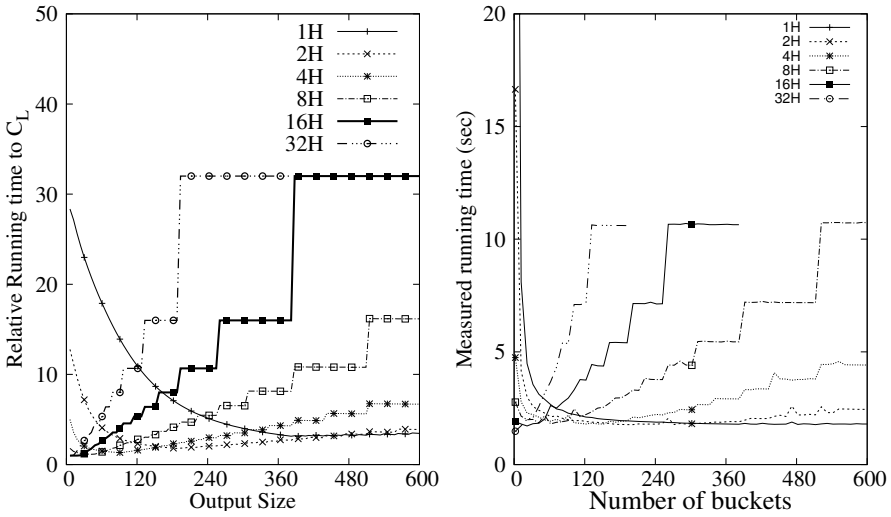


**Fig. 15** Performance with advanced output privatization: theoretical (left) and measured (right) running time of SDH kernel. Each line represents the case of one particular number of private output copies

**Table 5** 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 | $> 450$ | $> 300$ |

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. However, such selections under large output size do not significantly affect the performance of the algorithm. If we look closely into Figure 15, 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.

### *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. We define a similar function by using Euclidean distance. Naturally,

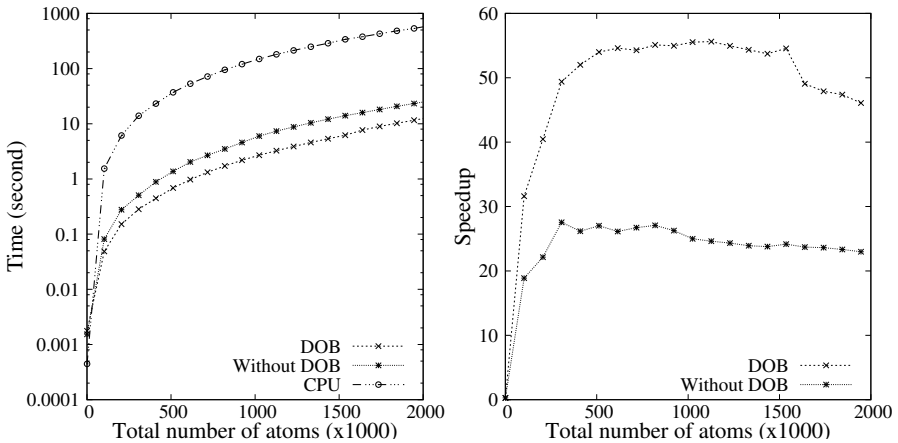


**Fig. 16** Output Buffer Management: total running time and speedup over CPU in computing the item similarity problem

the output size of this problem is unknown at the beginning. In our experiments, we use data with up to 2 million items and limit 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 16 demonstrates running time of the experiment. As we can see, our GPU code without the output buffer technique achieves an average 23X speedup over CPU program (Maximum speedup of 28X). When the direct output buffer is implemented, the speedup averagely increases to around 48X, 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 the kernel before and after applying the technique, and Figure 17 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 18 shows the results of the experiments. Clearly, tiling with shuffle

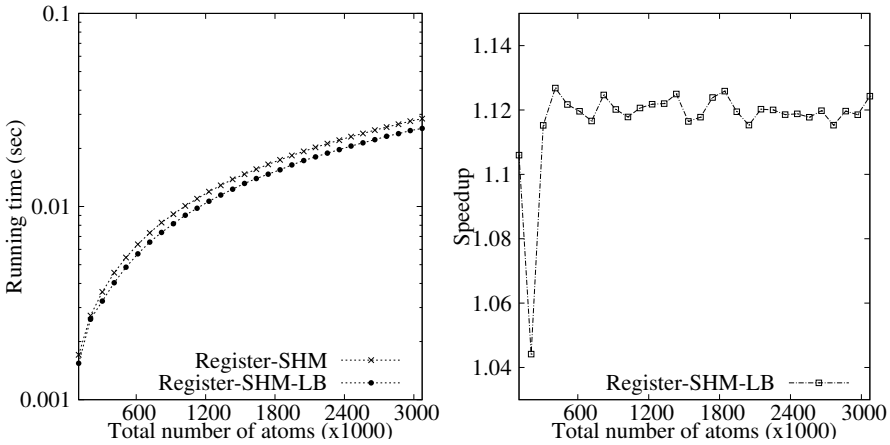


**Fig. 17** Performance of Reg-SHM kernel with and without load balancing method: total running time (left) and speedup (right) over Reg-SHM

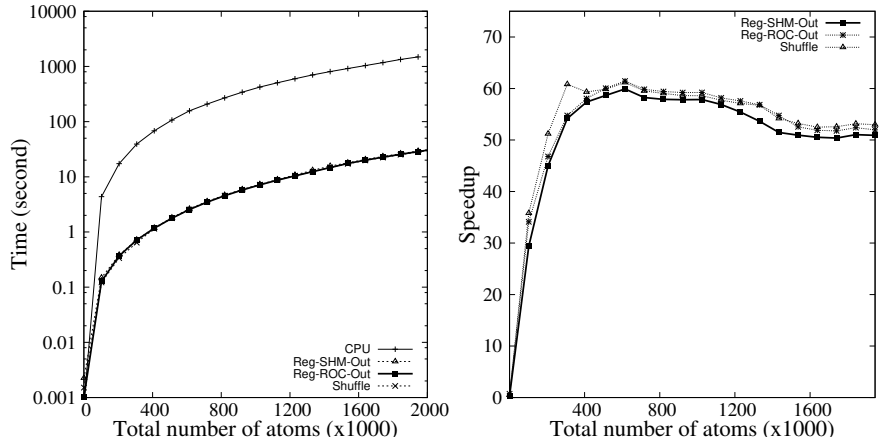


**Fig. 18** Performance of different GPU-based algorithms for computing SDH: total running time and speedup over CPU algorithm

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.

## 5.5 Results of Large Input Data

In this subsection, we present end-to-end performance of our multi-GPU 2-BS algorithm. We run our algorithm on multiple GPUs that reside in a single server running Linux with an Intel Xeon 6-core E5-2620 v3 CPU, 128 GB of

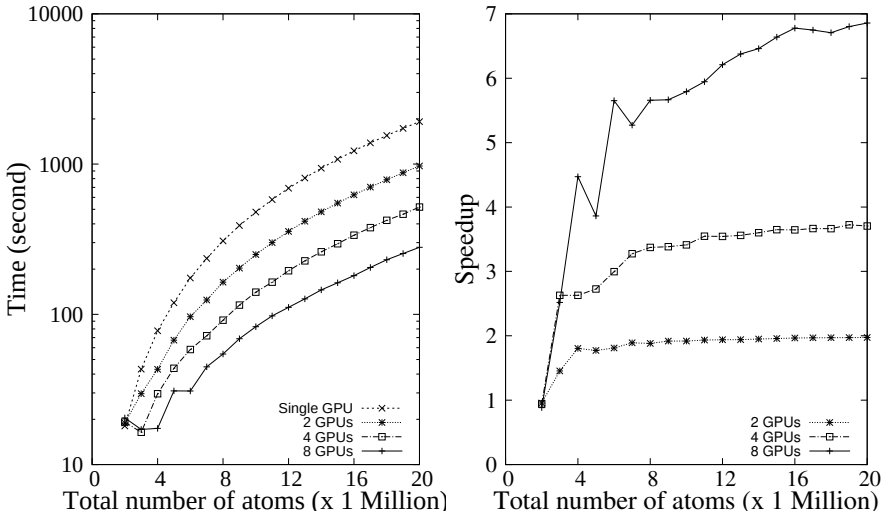


**Fig. 19** Performance of different setup for computing SDH: total running time and speedup over single GPU

DDR4 1866-MHz memory, and eight Nvidia Titan X Pascal GPUs with 12GB of global memory in each card.

In such experiments, we still use SDH as an example to demonstrate the performance of our algorithm. In particular, we implement the algorithm based on the best kernel for processing onboard input data as shown in Section 5.3.1, which is RoC-Out. We compare performance based on various numbers of GPU devices: single card, 2 cards, 4 cards, and 8 cards. We generate uniformly distributed datasets with a size ranging from 2 million to 20 million. The size of data blocks was set to 1 million points. We use this number because it is relatively large input size that requires a long time to compute SDH (even on GPUs).

Figure 19 shows the running time. We calculate the speedup over the single GPU setup. As we can see, the speedup is improved by almost the magnitude of a number of GPU cards. The multiple GPU system is up to 1.97 (2 GPUs), 3.72 (4 GPUs) and 6.85 (8 GPUs) times as fast as the single GPU system. The higher number of GPUs card tend to have more overhead, due to the time required to merge the output. However, the larger the input data is, the better the merging time can be hidden. As a result, the speedup of the algorithm increases with data size. In addition, we also used Nvidia Profiler to study the algorithm behavior. On 8 GPUs and 20 Million input data, the profiler shows that the average HtD (Host to device) transfer time is 847.10ms (0.04% of running time), and DtH (Device to host) transfer time is 10.881us (less than 0.01% of running time). This shows that the overhead of transfer time via PCIe is minimal.

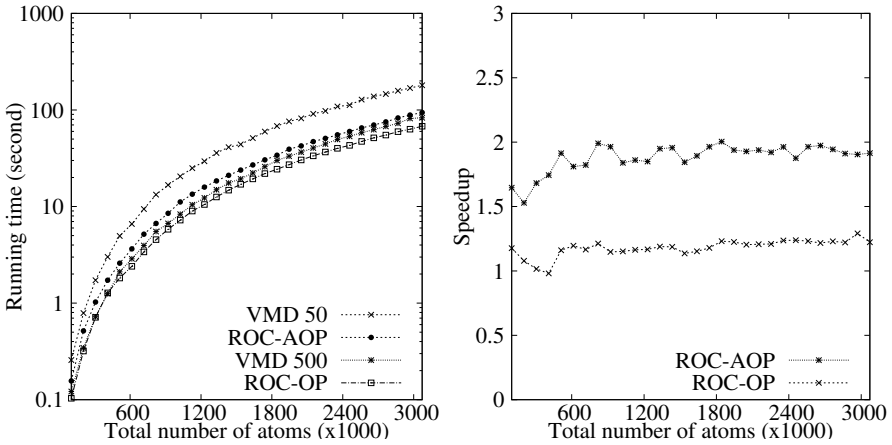


**Fig. 20** Performance of different GPU-based algorithms for computing RDF: total running time (left) and speedup over VMD code (right)

## 6 Case Study of Automated 2-BS Framework

In this section, we evaluate CUDA kernels generated by automated code generation framework via two case studies. We demonstrate Radial Distribution Function, a Type-II 2-BS, as Case Study I. Then, we present Nested-Loop join as case study II, which is a Type-III 2-BS.

### 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 well-known molecular visualization program used for displaying, animating, and analyzing large biomolecular systems. RDF is implemented on GPUs [16] 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. Parti-

cle coordinates are generated randomly following a uniform distribution. The RoC-OP has 45 lines of code while RoC-AOP is 51 lines long.

Figure 20 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 [32]. 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 21, 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 [32] 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.

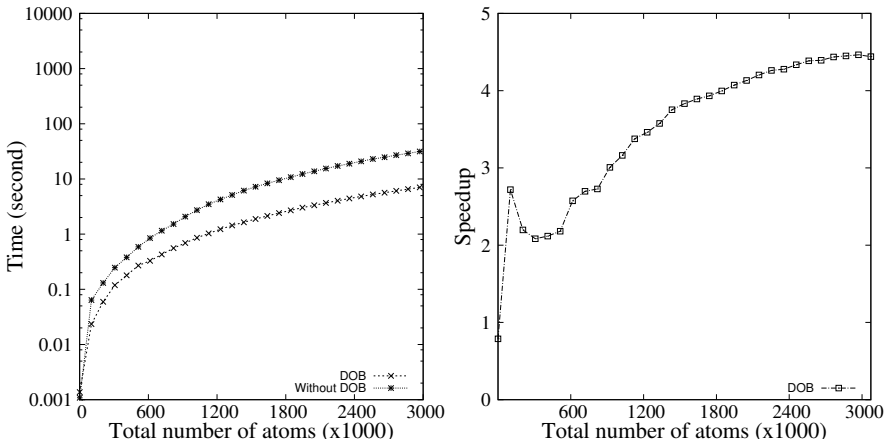


**Fig. 21** Performance of NLJ kernel generated from 2-BS framework as compared to best known NLJ kernel reported in [32]

## 7 Related Work

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 only a few reports on computing 2-BSs on GPUs. As a part of efforts to parallelize relational joins, He *et al.* implemented various functionally-equivalent CPU-base join algorithms on GPUs [10]. The algorithm was designed to take advantage of early generations of CUDA framework. They utilized on-chip memory on GPU and handled undefined size of join result by doing twice of computation: once to calculate the output size, and second to do actual output. The report showed a 7X speedup over CPUs. Similar results are presented in [33]. Rui *et al.* [32] utilized new feature of GPU for nested loop join (e.g., Read-only data cache, large L2 cache, and shuffle instructions) and reported speedup 20X over CPUs. The above work is often benchmarked against CPU-based parallel joins. For example, He *et al.* [34] proposed cache-oblivious nested-loop joins by grouping related data together to get better spatial locality and thus higher CPU cache hit. Kim *et al.* [35] implemented optimized sort-merge join and hash join on a multi-core CPU system. They improved data parallelism by taking advantage of the SIMD instructions. Albutiu *et al.* [36] designed a massively parallel sort-merge join on Multi-Core CPU where each thread only works on its local sorted partitions in a non-uniform memory access (NUMA) system.

Levine *et al.* [16] 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 via constant memory and shared memory to speed up the algorithm. Constant memory is high speed on-chip memory on GPUs with a drawback. In order to load new data to constant memory, the CUDA kernel needs to stop and be relaunched. Ponece *et al.* [37] used

tiling and privatization via shared memory with two point correlation function in cosmology application. They reported speedup of up to 100x over single-thread CPU code. However, no details of their implementation was reported in the paper. More recently, Stratton *et al.* sketched tiling and privatization techniques in computing two-point angular correlation function [15], but again, no technical details were reported.

Several work presented techniques related to handling data output in computing other related problems on parallel computing platforms. Karnagel *et al.* proposed GPU-based algorithms for computing group-by and aggregates that are often seen in database systems [38]. Similarly, they keep copies of output data in GPU cache. However, their implementation is based on caching the entire hash table, which is large and requires grid-level synchronization. For that, they proposed to use L2 cache that is slower as compared to other cache such as shared memory. Ye *et al.* [39] studied the same problem with multicore CPU as the implementation platform. They implemented a partition-and-Aggregate algorithm that focuses on CPU cache utilization.

There are other related work on improving performance of individual 2-BS computations via algorithms with complexity lower than quadratic. For computing SDH, the state-of-art work reduces point-to-point computations to pairwise computation among nodes in a spatial tree structure [40,41]. Such strategies can reduce complexity from quadratic to $\theta(N^{3/2})$ for 2D data and $\theta(N^{5/3})$ for 3D data. Their main idea is to group data in a local region into a tree node, then pairwise comparisons of tree nodes (instead of individual particles) are conducted. Therefore, the core procedure of pairwise comparison remains the same, and we could still parallelize the pairwise computation by GPUs.

Unlike the aforementioned work focusing on individual problems and techniques, our work aims at a comprehensive study of the multitude of techniques that can be used for the development and optimization of GPU-based 2-BS algorithms. We propose a series of novel techniques for minimizing overhead and increasing resource utilization, a few techniques are never seen in previous work. All types of GPU on-chip memory are exploited for the best results of caching input/output.

## 8 Conclusions

The 2-BS problems are popular and fundamental to many natural science domains. 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 develop load balancing and tilling techniques with shuffle instructions. Scalability of the GPU Algorithms is also studied to handle very large input/output data in 2-BS computation. Moreover, we develop a general 2-BS framework that can automatically generate CUDA code for user-defined 2-BS problems.

**Acknowledgements** This work is supported by an award (IIS-1253980) from the National Science Foundation (NSF) of U.S.A.. Equipments used in the experiments are partially supported by another grant (CNS-1513126) from the same agency.

## References

1. C. Türker, F. Akal, D. Studer-Joho, and R. Schlapbach, "B-fabric: An open source life sciences data management system," in *Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, New Orleans, LA, USA, June 2-4, 2009, Proceedings*, 2009, pp. 185–190.
2. M. Feig, M. Abdullah, S. L. Johnsson, and B. M. Pettitt, "Large scale distributed data repository: design of a molecular dynamics trajectory database," *Future Generation Comp. Syst.*, vol. 16, no. 1, pp. 101–110, 1999.
3. G. Finocchiaro, T. Wang, R. Hoffmann, A. Gonzalez, and R. C. Wade, "DSMM: a database of simulated molecular motions," *Nucleic Acids Research*, vol. 31, no. 1, pp. 456–457, 2003.
4. W. Xu, S. Ozer, and R. R. Gutell, "Covariant evolutionary event analysis for base interaction prediction using a relational database management system for RNA," in *Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, New Orleans, LA, USA, June 2-4, 2009, Proceedings*, 2009, pp. 200–216.
5. S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine, "Scalable linear algebra on a relational database system," in *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, 2017, pp. 523–534.
6. Y.-C. Tu, S. Chen, and S. Pandit, "Computing distance histograms efficiently in scientific databases," *ICDE*, pp. 796–807, 2009.
7. B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999.
8. L. Rokach and S. Kisilevich, "Initial profile generation in recommender systems using pairwise comparison," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 6, pp. 1854–1859, Nov 2012.
9. S. Jiang, X. Wang, and H. Zhu, "Learning pairwise comparisons of items with bigram content features for recommending," in *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on*, Oct 2013, pp. 446–449.
10. 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.
11. *NVIDIA: CUDA C Programming Guide Version 7.0.*

12. T. Group., "Opencl." [Online]. Available: https://www.khronos.org/opencl/
13. A. G. Gray and A. W. Moore, "N-body problems in statistical learning," *Advances in Neural Information Processing Systems (NIPS)*, pp. 521–527, 1993.
14. Y. Zhu, Z. Zimmerman, N. Shakibay Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh, "Exploiting a novel algorithm and gpus to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins," *Knowledge and Information Systems*, Dec 2017.
15. 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.
16. 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.
17. 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.
18. *NVIDIA GeForce Tesla V100 Whitepaper*.
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 GTX 1080 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, May 29 - June 2, 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. R. Rui and Y. Tu, "Fast equi-join algorithms on gpus: Design and implementation," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, 2017, pp. 17:1–17:12.
31. *2BS Framework*. [Online]. Available: https://github.com/napath-pitaksirianan/2-bodyFramework
32. 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.
33. 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.
34. B. He and Q. Luo, "Cache-oblivious nested-loop joins," in *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*, 2006, pp. 718–727.
35. C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, 2009.

36. M. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *PVLDB*, vol. 5, no. 10, pp. 1064–1075, 2012.
37. 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.
38. T. Karnagel, R. Müller, and G. M. Lohman, "Optimizing gpu-accelerated group-by and aggregation," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015.*, 2015, pp. 13–24.
39. Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable aggregation on multicore processors," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, 2011, pp. 1–9.
40. 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.
41. 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.