Journal of Engineering and Applied Science

RESEARCH Open Access

An experimental study of group-by and aggregation on CPU-GPU processors

Hua Luan[1*] and Lei Chang[2]

*Correspondence:
luanhua@bnu.edu.cn
[1]School of Artificial Intelligence, Beijing Normal University, Beijing, China
Full list of author information is available at the end of the article

Abstract

Hash-based group-by and aggregation is a fundamental operator in database systems. Modern discrete GPUs (graphics processing units) have been considered to accelerate the performance. However, the data transfer through the PCIe (peripheral component interconnect express) bus would reduce gains. On recent architectures, the GPU and the CPU (central processing unit) are built into the same chip which removes the data transmission and offers new performance opportunities. Yet there has been no systematic analysis of grouping and aggregation algorithms on such architectures. In this paper, we study the behaviors of various hash-based grouping and aggregation methods on coupled architectures to provide meaningful guidelines. We conduct an extensive experimental study and analysis on the single CPU, the coupled GPU, and both processors. Six dimensions are considered in analyzing the hashing methods carefully: (1) hashing scheme, (2) hash function, (3) data size, (4) group cardinality, (5) load factor, and (6) data distribution. Two additional dimensions are also explored: (7) shared and independent hash tables and (8) running on single processors and co-processing. We hope the results in our study could help database researchers to choose the right direction in terms of algorithm design and system optimization.

Keywords: Group-by and aggregation, Coupled CPU-GPU architectures, Experimental study

Introduction

Group-by and aggregation is an important and dominant primitive in analytical queries. Hashing is a common approach to implement the grouping operator. There has been a lot of research on hashing grouping on multi-core CPUs [1–3]. With the advent of the GPU accelerator, many efforts have appeared to accelerate the operation by offloading the workload to the GPU [4–7]. However, the discrete GPU is connected to the CPU and system memory through a PCIe bus. The data transfer over PCIe usually becomes the bottleneck to hamper the performance improvement [8–10]. In recent years, a new CPU-GPU architecture attracts growing interest of researchers, where a GPU coprocessor is integrated into the CPU chip which forms coupled CPU-GPU architectures. On this architecture, the CPU and the integrated GPU share the same memory and data transmission through the PCIe bus is eliminated. Researchers have focused on leveraging the

computing power of the coupled GPU in database systems [4, 7, 11–13]. Despite previous work, as far as hash grouping and aggregation is concerned, there does not exist an in-depth study on coupled CPU-GPU architectures.

In this paper, we carefully study hash-based group-by and aggregation in a multi-dimensional space on such heterogeneous architectures. We aim to investigate answers to the following questions: (1) How well do various hashing methods perform on the CPU and the emerging coupled GPU? (2) What are the most efficient methods and when should we consider them? (3) Whether and how does the coupled GPU help to increase the overall performance? (4) Where does time go when the group-by and aggregation operator is executed on heterogeneous architectures?

To this end, we conduct a detailed experimental study and analysis from a variety of dimensions. (1) Hashing scheme. Linear probing, two versions of chained hashing and two versions of Cuckoo hashing are considered. (2) Hash function. Two representative hash functions Multiply-shift and Murmur hashing are used and compared. (3) Data size. We consider different data sizes to examine the effects when the data is rather small and large. (4) Group cardinality. We observe the performance of hashing methods for a variety of group numbers. (5) Load factor. Six load factors between 25% and 90% are explored. (6) Data distribution. We consider two data distributions: uniform distribution and Zipf distribution. (7) Hash table. Shared and independent hash tables are investigated. The shared hash table is accessed by all the work items. If hash tables are independent, each work item has its own hash table and a global hash table is built at last. (8) Processor. We examine the performance of hash grouping and aggregation on the CPU and the coupled GPU to understand the ability of heterogeneous processors. In addition, two strategies including inter-operator parallelism and intra-operator parallelism are exploited in co-processing to explore potential benefits from the coupled architecture. Our main goal is to acquire a comprehensive understanding of hash-based grouping and aggregation on modern hybrid architectures, which could guide researchers to design and optimize future database algorithms and systems.

The remainder of this paper is organized as follows. First, we briefly describe the coupled CPU-GPU architecture, OpenCL, three kinds of considered hashing schemes and two hash functions. Then, we give our methodology and the experimental setup. Later, the experiment results and discussion are presented. Finally, the related work is described and we conclude our work.

Preliminaries

In this section, we first describe the coupled CPU-GPU architecture and the OpenCL standard. Then, we introduce three types of hashing schemes and two hash functions examined in this paper.

Coupled CPU-GPU architectures

The coupled CPU-GPU architecture is a hybrid architecture which integrates the CPU and the GPU in the same chip. On traditional discrete architecture, the GPU is equipped with a great number of cores and has excellent computing power. However, data need be transferred between the main memory and the GPU memory through the PCIe bus. This has become a nonnegligible overhead when using the discrete GPU as an accelerator. On the emerging coupled architecture, the integrated GPU has much fewer computing units

compared with the discrete GPU and is not as powerful as the latter. But the integrated GPU shares the same physical memory and the last level cache (LLC) with the CPU which reduces the data transfer cost between two processors since the slow transfer via the PCIe bus is not needed. It is meaningful to explore the benefits from the hybrid architecture in applications.

Figure 1 shows the architecture of modern Intel processors. Intel supported its integrated architecture on the processors such as Ivy Bridge. We use Intel Core i7-9700K as the major experimental platform in this paper which is the 9th generation processor and includes an integrated graphics processor, Intel UHD Graphics 630. The hardware configuration is described in Table 1. The CPU has 8 cores and L1, L2, and L3 cache. L3 cache of size 8 MB (Megabyte) is the last level cache which could also be used by the integrated GPU. On some laptop processors, an embedded dynamic random access memory (EDRAM) is supported as a larger cache. The GPU contains 24 Execution Units (EUs) and each EU can be regarded as a multithreaded SIMD (single instruction multiple data) processor. Usually, the CPU is much faster than the integrated GPU and the GPU clock speed of Core i7-9700K is 350MHz. However, the GPU could provide more parallel computing resources and the actual computational capability will be examined in this paper.

OpenCL

The OpenCL (Open Computing Language) [14], developed by an open and non-profit consortium *Khronos Group*, is a standard and framework for parallel programming on CPUs, GPUs and other processors and accelerators. It specifies a C-like language and APIs (application programming interfaces) to write and launch programs across various platforms. The same code in C and OpenCL could be executed on either CPUs or GPUs, which facilitates the software design on heterogeneous architectures. Hardware vendors like Intel, AMD, and NVIDIA have supported the OpenCL standard by providing specific OpenCL implementations.

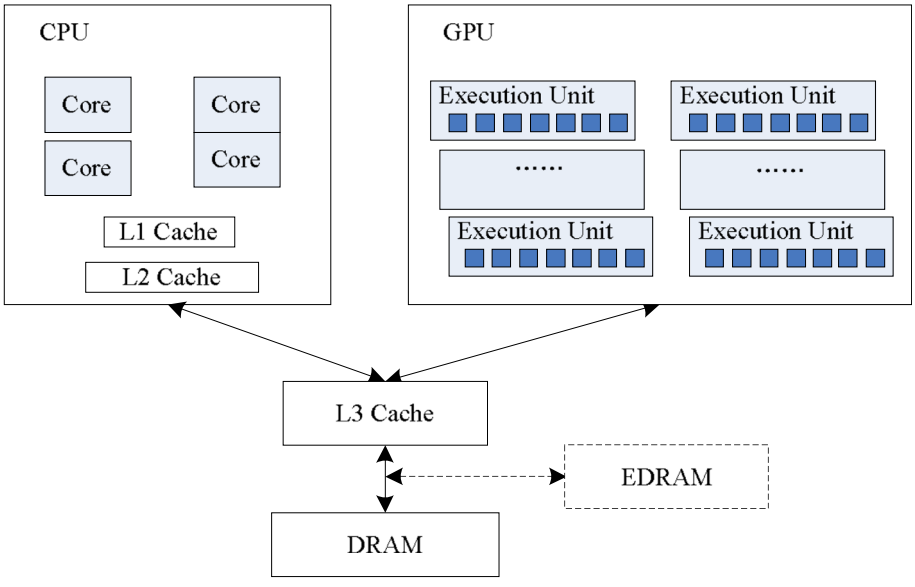


Fig. 1 Intel CPU-GPU architecture

Table 1 Hardware parameters

Processor	**Intel Core i7-9700K**	
Type	CPU	GPU
Name	N/A	Intel UHD Graphics 630
Basic units	Core	EU
Number of units	8	24
Base clock rate	3.6 GHz	350 MHz
TDP	95 W	
LLC	12 MB	

In the OpenCL model, there exist one *host* and one or more *devices* in computer systems. The host usually refers to the CPU where the host code could run. The host code is in charge of preparing and triggering OpenCL program execution on devices. The device covers a wide range of processors such as CPUs, GPUs and other accelerators. Each device contains more compute units and each compute unit is composed of more SIMD processing elements. The parallel work is submitted to devices by launching *kernels* which are coded in the OpenCL language. The instances of a kernel are broken up into *work groups*. Each work group includes the same number of *work items* and is scheduled to run on a single compute unit. Work items within the same work group are executed concurrently by processing elements and could synchronize with each other using barriers or memory fences.

Hashing schemes

When performing hashing, multiple keys may be mapped into the same value. How to handle key collisions is a problem. We study the performance of three kinds of hashing schemes: (1) linear probing, (2) chained hashing, and (3) Cuckoo hashing. Linear probing and Cuckoo hashing belong to the static hashing technique. The size of the hash table is fixed and if the hash table could not provide enough space, a new hash table has to be rebuilt from scratch. This is expensive and may be a factor which affects the use in practice. Chained hashing is a dynamic hashing scheme by maintaining linked lists which is able to resize the hash table without needing to rebuild the whole table.

Linear probing

Linear probing is a simple open-addressing hashing collision resolution technique. The used hash function is $h(k, i) = (h'(k) + i) \bmod S$, where i represents the i-th probing, $h'(k)$ is an initial hash function, and S denotes the hash table size. The slot $T[h(k, 0)]$ in the hash table T is first checked for a key k. If the slot is empty, the key is placed into it. Otherwise, the next free slots $T[h(k, 1)]$, $T[h(k, 2)]$, etc., are searched until an empty slot or the same key is encountered. The collision resolution technique is to probe through a set of alternate locations in the table in a cyclic manner.

The advantage of linear probing is its simplicity and sequential scanning. This method has been adopted in many studies [4, 5, 7]. The problem in this collision resolution is *primary cluster* [15], that is, a large group of consecutive slots are occupied when an empty slot needs to be searched for a key. The consecutive sequences form a *cluster* where keys whose hash values lie require excessive probing, which degrades the performance of linear probing. We will examine the performance of this scheme as one of the comparison methods in the experiments.

Chained hashing

Chained hashing is a classical method to deal with collisions, which is widely used in practice such as functions in C++ STL (standard template library) and Java [15]. Keys with the same hash value are placed in a bucket and all the slots in a bucket are linked together with a head pointer pointing to the first slot. The directory which stores the head pointers and pointers to the next slot in the linked list are extra space used by chained hashing compared with linear probing. When a collision happens, the corresponding bucket is traversed to judge whether the key has existed. If not, a new empty slot is added into the linked list of the bucket. The characteristics of the linked list bring two disadvantages: more memory space and irregular memory accesses. In order to reduce cache misses, a variant of chained hashing was presented in the previous study [15] where the directory is 24 bytes wide to keep key-value-pointer triplets so that one element could be directly stored in the directory and the linked list is not needed to be visited. In our study, we consider two versions of chained hashing: the standard scheme and the variant mentioned above.

Cuckoo hashing

Cuckoo hashing [16] is another type of open-addressing scheme, which maintains two hash tables T_0 and T_1 with their own hash function h_0 and h_1 in the simple version. Keys are stored in one of the hash tables. Every hash table is checked to find an empty slot or a slot containing the same key. If both tables have an empty slot, any of the two hash tables are chosen alternately to store the new key k. If neither of them has an empty slot in the corresponding location, the key k' in the slot $T_0[h_0(k)]$ in the first hash table is kicked out to make room for the new key k, which is inserted into the second hash table at location $T_1[h_1(k')]$. If the location in the second table is not free, the existing key is evicted again to probe its location in the first table. The advantage of Cuckoo hashing is two accesses are required at most to find a key. However, the insertion process may end up in a cycle and no empty slot could be found for a key especially for high load factors. Increasing the number of hash tables is one of the ways to alleviate the problem. In our experiments, besides the traditional Cuckoo hashing with two tables, the version using four hash tables is also explored to check the performance. The hash functions Multiply-shift and Murmur hashing described later are used respectively by two hash tables. Besides them, two additional functions such as $((a \times k + b) \bmod p) \bmod B$ with different parameters are utilized in the other tables.

Hash functions

Two representative hash functions are considered in our study which are widely used in practice: (1) Multiply-shift and (2) Murmur hashing. The detailed description is as follows.

Multiply-shift

Multiply-shift [17] is a universal hash function used in previous studies [7, 15] and the definition is:

$$h(k) = (k \times r \,\%\, 2^w) \,/\, 2^{w-d} \tag{1}$$

where k is a w-bit integer, r is an odd w-bit integer, and d means the hash table is of size 2^d. The operator "/" could be implemented by a right bit shift by w-d positions. In our

experiments, w is set at 32 that is keys are 32-bit integers, and r is an odd integer generated randomly.

Murmur hashing

Murmur hashing is a common hash function which is widely used in the literature [4, 5, 15]. This hash function is more complex than Multiply-shift and we use the Murmur3 implementation [18] for 32-bit keys in this paper with the finalization code shown as follows:

```
uint32_t fmix32 (uint32_t h) {
                    h ^= h >> 16;
                    h *= 0x85ebca6b;
                    h ^= h >> 13;
                    h *= 0xc2b2ae35;
                    h ^= h >> 16;
                    return h;
                }
```

Methods

In this paper, we aim to understand how well a hash-based group-by and aggregation operator works on the new coupled architecture. The following group-by query is used as a representative example:

```
SELECT c1, count(c2), sum(c2)
FROM T
GROUP BY c1
```

There are two 32-bit integer columns $c1$ and $c2$ in a table T. The query calculates the *count* and *sum* aggregates grouped by $c1$. The column $c1$ provides the keys on which a hash table is built. Two aggregates are stored and updated with the establishment of the hash table. The study in this paper is also suitable for other algebraic aggregates like *average*, *min*, and *max*.

Load factors

We assume the group cardinality G is known in advance which actually could be estimated by the optimizer in a DBMS (database management system) and the hash table size S is set at a power of 2. The ratio G/S is considered as the *load factor* of the hash table. For linear probing a hash table has S slots, and for chained hashing, there are also S slots in total and the directory stores the same number of pointers. In Cuckoo hashing, the size of each hash table is the minimum power of 2 which is greater than or equal to $S/2$ or $S/4$. In our experiments, the total size of multiple tables is just equal to S because the quotient is exactly a power of 2. We examine the performance of various grouping methods with six load factors: the low load factors 25%, 35%, and 45% and the high ones 50%, 70%, and 90%, which cover a wide range of collisions. At low load factors, collisions will be rare and become apparent when the load factor is increased.

Parallelization strategies

Two parallelization strategies are adopted to design hash tables: *shared* and *independent*. In the first strategy, a single globally shared hash table among all the work items in different work groups is used to perform grouping. Concurrent updates to the same bucket or slot are resolved with atomic access primitives. Compared with the independent strategy,

the memory requirements of a shared table are reduced. However, on the other hand, contentions may cause a performance problem especially when collisions occur often. The second strategy is a simple parallel approach and each work item aggregates into its own hash table. The advantage is that contentions on the same data are eliminated and expensive synchronization primitives are avoided. However, more memory space is needed to store many hash tables and a subsequent phase is required to merge the independent tables into a global one. We consider the shared approach when using linear probing and chained hashing. The independent hash table is combined with all the hashing schemes. The reason why we do not adopt a shared table for Cuckoo hashing is that for OpenCL there is no synchronization between work groups and it is hard to ensure the correctness of updates and accesses to a hash table. When Cuckoo hashing was implemented with a shared hash table, accesses to each sub table by all the work items from different work groups needed to be synchronized; otherwise, faster work items would modify any sub table which led to wrong results. However, the global synchronization operation of work items between work groups is not provided by the current OpenCL implementation. Thus, the shared hash table will not be combined with the Cuckoo method.

Co-processing

Since there exist two processors on a chip now, it is reasonable to utilize two kinds of computing resources together besides assigning a query on a single processor. In this work, we investigate task and data parallelism models of co-processing. When the task parallelism model is used, the CPU and the coupled GPU handle different group-by and aggregation queries simultaneously which forms a kind of inter-operator parallelism. The queries are of the same form as the representative example illustrated previously with the varied tuple number or group count. If intra-operator parallelism is adopted, the CPU and the GPU process part of data at the same time to build a global shared hash table. The other operations in the operator are performed by a separate processor.

Although the coupled GPU provides the extra computing ability, it adds to the pressure on the memory bandwidth since the CPU and the GPU access the unified memory. The concurrent visits to different memory regions may also bring the cache contention which results in more cache misses. In addition, the execution on the GPU acquires the control and coordination of the CPU which may affect the overall co-processing performance to some extent. In view of the above factors, we study the co-processing strategy in our experiments to explore the performance gains from the hybrid architecture.

Implementation

The implementation of the group-by operator adopts a column-based model, which is true for a columnar database suitable for analytical query processing. The raw data, intermediate structures, and the results are all stored by columns. The raw table is generated as a file on the disk and read into memory in advance.

The major process consists of four phases and involves three kernels. The first phase is to create and initialize data structures used in the building step. For linear probing and Cuckoo hashing, three arrays are allocated to store the distinct group-by key, the count of tuples and the sum of values respectively. For the traditional chained hashing, two extra arrays are needed in a hash table to keep the head and next pointers which are represented by array indexes in our implementation. Figures 2 and 3 show the structure

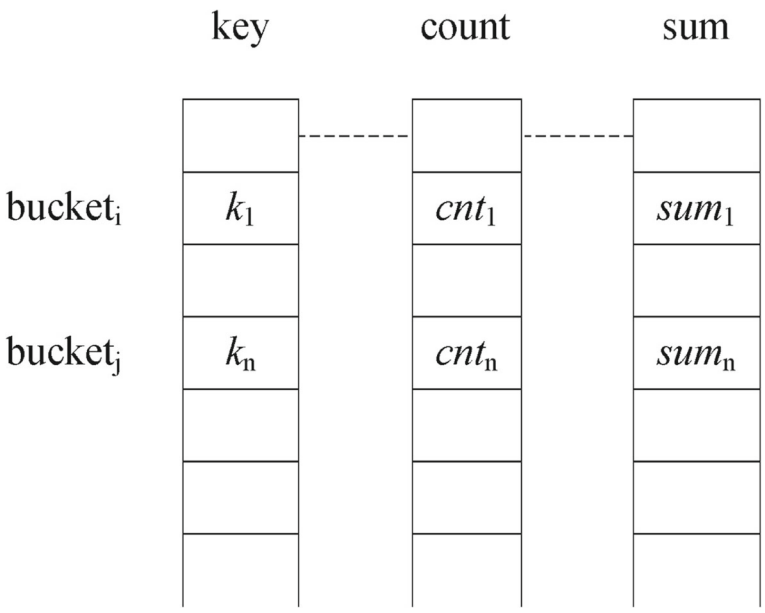


Fig. 2 Linear probing structure

and implementation of linear probing and the classical chained hashing. The variant of chained hashing relates three other arrays to the directory which are used to place the first key, count and sum directly. The work in this phase is done by the CPU.

The second phase is to construct shared or independent hash tables by launching a kernel *build_hash_table* on the CPU or the coupled GPU or both of them in co-processing. The concrete implementation of the kernel is associated with different methods. Algorithm 1 shows the pseudo-code of linear probing with shared tables as an example. The kernel acquires the global number of work items that is how many work items will deal with respective tuples simultaneously (Line 1). The value is the same for all the work items. Then, each work item gets its item ID which ranges from zero to the number of work items minus one (Line 2). Every work item accesses the tuples at intervals of the work item count (Line 3). For every tuple, the kernel calculates the bucket number based on the grouping key and checks the corresponding bucket until the insertion process is

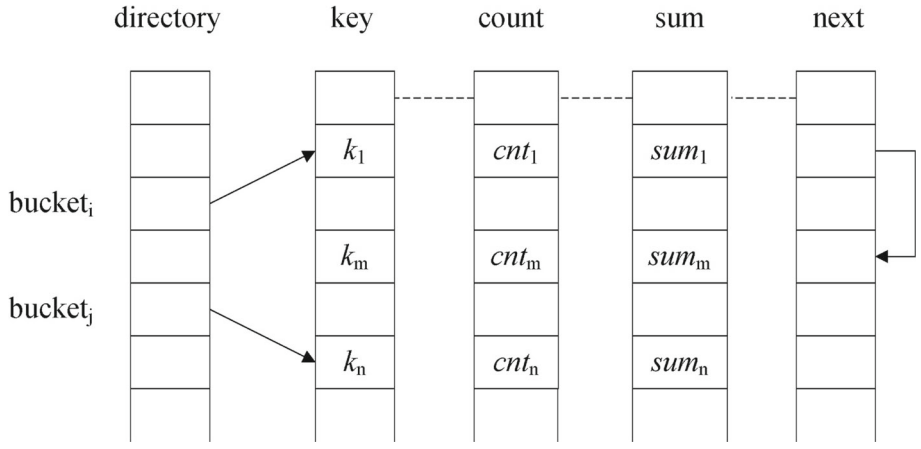


Fig. 3 Chained hashing structure

Algorithm 1 The kernel *build_hash_table* of linear probing with shared hash tables

```
_kernel void build_hash_table (_global int *c1, _global int *c2, _global int *key,
_global int *count, _global int *sum, ...)
 1: stride = get_global_size(0); // how many work items in total
 2: tid = get_global_id(0); // the id of the current work item
 3: for (i = tid; i < tupleNum; i += stride)
 4:   calculate the bucket number b based on k that is c1[i]
 5:   while (!success)
 6:     if (key[b] is empty)
 7:       ret = atomic_cmpxchg(key[b], empty, k);
 8:       if (ret is empty or ret is equal to k)
 9:         update the values in count and sum arrays atomically
10:         set success with true
11:       else
12:         b = (b +1) % the size of the hash table
13:   else if (key[b] is equal to k)
14:       update the values in count and sum arrays atomically
15:       set success with true
16:   else
17:       b = (b +1) % the size of the hash table
```

finished (Line 4 and Line 5). If the bucket is empty, the key will be filled in the bucket using the atomic_cmpxchg function (Line 7). If the bucket is set successfully or it has been set with the same key by another work item, the aggregates are updated atomically (Line 8 and Line 9). Otherwise, the next bucket has got as a candidate and will be examined later (Line 12). If the bucket is not empty at the beginning, there are two cases. One is that there exists a key which is the same as the grouping key to be inserted, the kernel revises the count and sum values directly (Line 14). Otherwise, the bucket has been occupied by another key and the kernel checks a new bucket to try to perform the insertion operation (Line 17). Suppose there are four work items and the first four tuples in the table are ⟨1, 10⟩, ⟨2, 20⟩, ⟨3, 30⟩ and ⟨4, 40⟩. An illustration of linear probing is shown in Fig. 4. The *build_hash_table* kernel is executed by all the four work items and each work item handles one tuple. If ⟨4, 40⟩ has the same initial bucket number as ⟨1, 10⟩, the fourth work item has to use the next free bucket to resolve the collision.

After the second phase, a shared hash table or multiple separate hash tables containing the distinct keys and their aggregates are established. In order to get the final results from the hash table, two subsequent stages are needed. The third phase differs in the functions for the two parallelization strategies. If a shared hash table is used, the second kernel *gather_item_num* counts the number of keys a work item will access in the last phase. By calculating a prefix sum on the numbers, each work item knows the offset from which it could put data in the result array. Then, in the third kernel *aggregate_result*, the final aggregation results are acquired and filled into arrays by the same work items in parallel. If the independent strategy is adopted, the last phase completes the similar function by executing the *aggregate_result* kernel. The difference lies in the second ker-

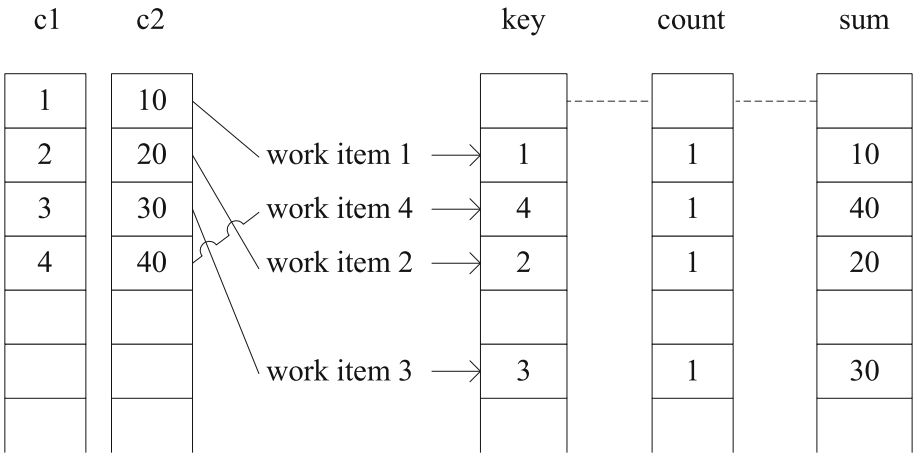


Fig. 4 An illustration of linear probing

nel *gather_item_num*. Before counting the key number for every work item, a global hash table is needed to be generated.

For two variants of chained hashing, the head pointer of keys belonging to a global bucket has the same position in each local directory. Thus, in the process of producing the unique hash table, every work item consolidates the corresponding sub-buckets in all of the independent hash tables. The sub-buckets in the later tables are merged into the ones in the first hash table. If keys are equal to each other, two aggregates are updated. If the key only exists in the later hash table, the slot is inserted into the appropriate location in the linked list. After a work item finishes the current bucket, it turns to the next bucket over an interval of the global size of work items. For linear probing and Cuckoo hashing, keys in a global bucket may be not in the same position in different local hash tables. A parallel method similar to the merge-sort is utilized. Each work item merges two adjacent local hash tables into an intermediate hash table. Two adjacent intermediate hash tables are processed by a work item further. Until the end, a final global hash table is established.

The second and third kernels could run on either the CPU or the coupled GPU. In our experiments, we will examine how the time is spent from the perspective of different phases to understand the performance of various methods on various platforms.

Experimental setup

Our experiments are conducted on Intel Core i7-9700K unless noted otherwise. The major hardware parameters are shown in Table 1. The machine is equipped with a 16-GB (Gigabyte) memory and Microsoft Windows 10. The code is implemented in C and OpenCL and compiled by Visual Studio 2017 with -O2 enabled in 64-bit mode. The raw data is read from disk into the memory and the time is not included. The OpenCL initialization overhead such as creating the context, queue, and kernels and building the program is not considered since these operations need be executed only once and the overhead could be amortized over queries. Each group-by and aggregation is run four times. The first run is used to exclude the warm up penalty and the average value based on the following three executions is used as the final time.

If the algorithms work on the CPU, the number of work groups is set to the core number of the CPU which is 8 on our platform. The number of work items in each work group

is set to 1. Thus, the global size on the CPU is 8. For the coupled GPU, the number of work groups is 32 and each work group has 32 work items to make full use of computing resources. All the data are 32-bit and there are two kinds of data distributions. The first is uniformly distributed and the number of keys in each group is similar. The second is skewed and generated using a Zipf distribution with the parameter set to 3, which means the most frequent value occurs in about 80% of the data.

Results and discussion

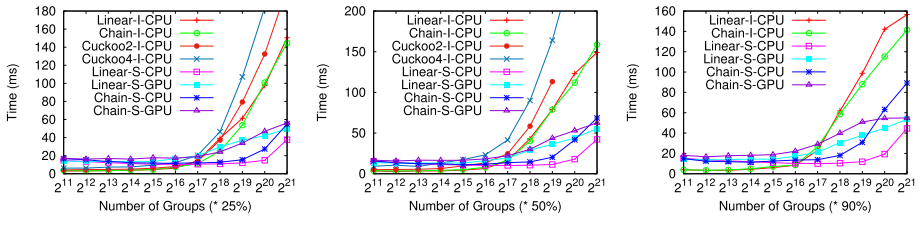
In this section, we provide the experimental results and performance analysis of various methods in grouping and aggregation. We use "Linear" to represent linear probing, and "Chain" and "Chain+" for the standard chained hashing and the variant of chained hashing. The terms "Cuckoo2" and "Cuckoo4" denote Cuckoo hashing using two hash tables and four hash tables respectively. "I" and "S" represent the independent and shared hash tables respectively. "CPU" and "GPU " denote the processor where the algorithm runs. In addition, "LF" is used to indicate the load factor. If not specified, chained hashing mostly refers to the standard chained hashing scheme, Multiply-shift is used as the hash function and the data is uniformly distributed.

Effects of group cardinalities

In the first set of experiments, we analyze the performance difference among 11 different configurations from five hashing schemes, two parallelization strategies and two processors with changing group cardinalities. Figure 5a to c show the execution time when there are 2^{21} (2M) tuples whose size is relatively small. Figure 6a to c give the performance when the data size is set to 2^{25} (32M) which is rather large. The values of X-axis represent the hash table size from which the actual group cardinality could be derived by multiplying by LF. We tested various methods under six load factors 25%, 35%, 45%, 50%, 70%, and 90%; however, due to the limited space, some of the similar results are omitted and we only present the results at the representative load factors in this paper. In addition, for linear probing, chained hashing and Cuckoo hashing, the execution times with independent hash tables on the coupled GPU are omitted in these and the following figures in order to show the performance difference of the other methods clearly because their times are much larger than the counterparts on the CPU, which demonstrates that independent hash tables are not suitable for the coupled GPU.

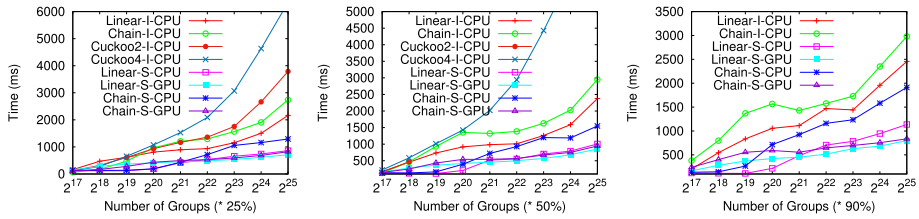
From the figures, we could obtain the following observations:

First, when the data size and group cardinality are small, all the methods show a relatively stable performance and the independent strategy performs better than the shared



(a) LF = 25% (b) LF = 50% (c) LF = 90%

Fig. 5 Performance as the number of groups changes with different load factors when the data has 2^{21} (2M) tuples



(a) LF = 25% (b) LF = 50% (c) LF = 90%

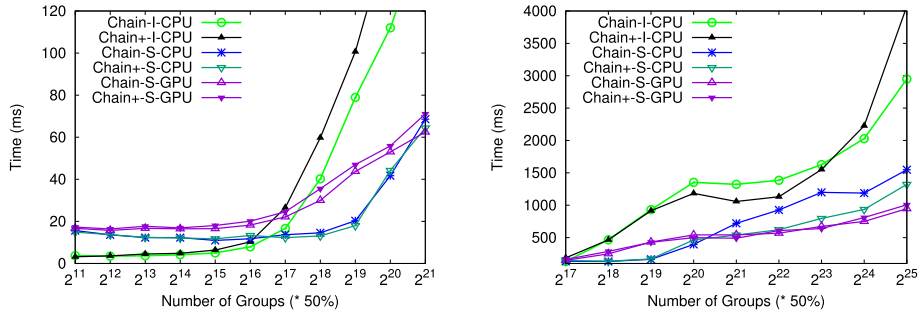
Fig. 6 Performance as the number of groups changes with different load factors when the data has 2^{25} (32M) tuples

one, as found in the left part of the curves in Fig. 5a, b and c. When the group cardinality becomes larger, the execution time goes up drastically. This implies the group cardinality does not affect the overall time too much within a certain range for the small data set. Outside the scope, a larger group cardinality, which means a bigger hash table when the load factor is fixed and more keys are stored in the table, has a great influence on the performance. Moreover, multiple independent hash tables bring a larger performance degradation than the shared table with the increase of the group count. Thus, the shared parallelization strategy should be considered as the primary choice for a larger data size or group cardinality. In Fig. 5b, the shared approach could achieve a speedup up to 6.9 and 3.9 on the CPU for linear probing and chained hashing respectively. In Fig. 6b, the maximum speedup is 6.5 and 5.7.

Second, two variants of Cuckoo hashing are slower than linear probing and chained hashing in most cases when using the independent hash table. In the results, in order to make the difference clear, some large time values are not plotted. Cuckoo hashing on two tables only works when the load factor is 25%. It begins to fail in some cases due to too many tries back and forth when the load factor is set to 35%, 45%, and 50%. If the load factor goes up to 70% and 90%, the method could not run at all. The version of Cuckoo hashing on four tables performs slower than that with two tables while it is feasible for a high load factor except 90%.

Third, linear probing and chained hashing on the CPU and the coupled GPU show the following behaviors: when the data size is 2M, the CPU algorithms run faster than the GPU counterparts except in several cases when the number of groups is large; if 32M tuples need to be processed, the trend is more obvious: the methods on the CPU perform better than those on the GPU when the group cardinality is low and the GPU methods show better performance if the group cardinality is increased. Thus, the coupled GPU has performance gains under time-consuming scenarios which makes it more useful to consider the accelerator in the grouping operator. In addition, linear probing outperforms chained hashing in most cases on either of the two processors. The performance gap of two kinds of hashing schemes on the CPU is larger than that on the GPU which sometimes is very close.

Fourth, in order to distinguish the performance clearly, two versions of chained hashing are illustrated separately in figures. We could find that for the configurations that run fastest, the two variants show similar performance. For example, in Fig. 7a, when the group cardinality is not bigger than 2^{16}, Chain-I-CPU is close to Chain+-I-CPU. When the number of groups is increased, the best methods Chain-S-CPU and Chain+-S-CPU achieve a close performance. This similar phenomenon could also be observed in Fig. 7b.



(a) 2M data (b) 32M data

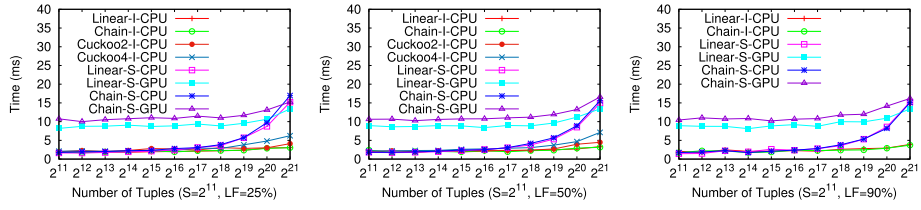
Fig. 7 Comparison between two kinds of chained hashing as the number of groups changes with different data sizes

In addition, on the whole, for the 2M data the standard chained hashing is a little better and when the data size is 32M, the variant of chained hashing demonstrates the advantage in many cases.

Fifth, there are clear breaking points where the execution time starts to increase linearly with the problem size especially for the methods using the independent hash tables. The size of hash tables becomes larger with the increase of the number of groups. For the independent strategy, every hash table grows exponentially and more memory space is needed. Thus, the increase of the group cardinality will lead to more LLC misses which reduce the performance. Take Fig. 5b for example, when the value on X-axis is less than or equal to 2^{15}, it is hard to detect LLC misses when Intel VTune Profiler, a performance analyzer Intel provides, is used to profile Linear-I-CPU. When the value is set to 2^{16}, 2^{17} and 2^{18}, the corresponding miss count is 140,000, 420,000, and 840,000, which may explain the occurrence of the breaking point.

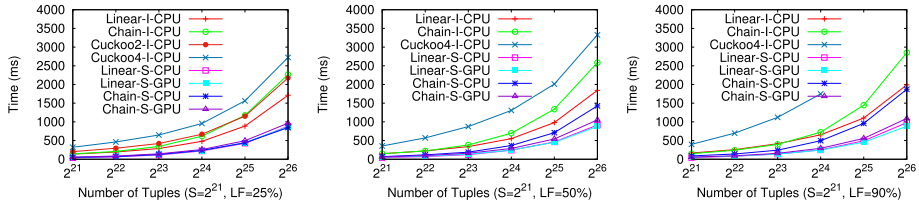
Effects of data sizes

In the second set of experiments, we compare various methods with the number of tuples varied. The hash table size is set to 2^{11} and 2^{21} respectively. The group cardinality responds to the change of the load factor. For space reason, the results at load factors 25%, 50%, and 90% are displayed in Figs. 8a to 9c. There exist 512, 1024, and 1843 groups in Fig. 8a, b and c respectively. The maximum number of tuples is 2^{21}. The group cardinality is 524,288, 1,048,576, and 1,887,436 in Fig. 9a, b, and c respectively, and the data size reaches 2^{26}.



(a) LF = 25% (b) LF = 50% (c) LF = 90%

Fig. 8 Performance as the number of tuples changes with different load factors when the number of groups is set at 2^{11} * LF with the hash table size of 2^{11}

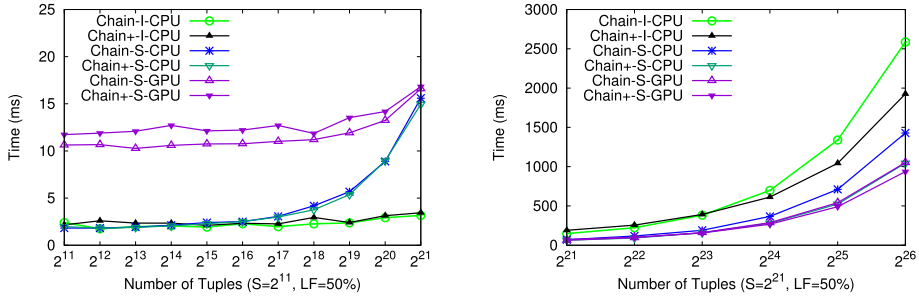


(a) LF = 25% (b) LF = 50% (c) LF = 90%

Fig. 9 Performance as the number of tuples changes with different load factors when the number of groups is set at 2^{21} * LF with the hash table size of 2^{21}

In the former group of figures where the data size and group cardinality are relatively small, the algorithms using the independent strategy are the top performers and the GPU methods offer the worst performance in most cases. The feature of Linear-S-CPU and Chain-S-CPU is that they have similar performance which is competitive with the best performers at the beginning and drops significantly with the increase of the data size. When the data size is 2^{21}, the shared CPU and GPU methods meet together. Cuckoo hashing does not work when the load factor is 90% and is slower than the other two hashing schemes for larger data sizes. In Fig. 9a to c, it could be pointed out that the four shared methods perform better than the independent ones. All the curves are in the upward trend in the execution time when the data size is increased. In terms of load factors, when the load factor is 25%, the four shared approaches show rather similar performance, and Chain-S-CPU falls behind obviously when the load factor goes up. A small gap among Linear-S-CPU, Linear-S-GPU, and Chain-S-GPU also begins to appear, and linear probing is slightly faster. The difference of two versions of chained hashing is shown in Fig. 10a and b. The behaviors are similar to those described above. When the data size is large, Chain-S-CPU is affected by increasing load factors. The other three shared methods are close and Chain+-S-GPU performs slightly better.

In Fig. 8, there are no LLC misses since the data size and group cardinality are small. There exist breaking points for Linear-S-CPU and Chain-S-CPU because their retired instructions are increased with the number of tuples. For example, in Fig. 8b, when there are 2^{15}, 2^{16}, 2^{17}, 2^{18}, and 2^{19} tuples, Linear-S-CPU executes 720,000, 1,440,000, 3,240,000, 5,400,000, and 12,240,000 instructions respectively. The CPI (cycles per instruction retired) is in the range of 6.7–9.5. As a result, the execution time increases approximately linearly with the problem size.



(a) S = 2^{11} (b) S = 2^{21}

Fig. 10 Comparison between two kinds of chained hashing as the number of tuples changes with different hash table sizes

So far, we could derive the following results from the two sets of experiments:

(1) The shared parallelization strategy is suitable for large data sets or group cardinalities, in other words the heavy workload. If the data size and group cardinality are small, the independent strategy works well.

(2) The coupled GPU on the hybrid architecture could help to increase the grouping and aggregation performance, which performs better for the time-consuming workload. Offloading the task to the coupled GPU is an option worth considering.

(3) Linear probing and chained hashing on the CPU are close to each other when the independent strategy is used to acquire good performance. When the shared hash table is adopted: linear probing on the CPU performs better than the CPU chained hashing in most cases and the gap is obvious with the increase of the group cardinality and load factor; for large data sets and group cardinalities, the GPU linear probing is competitive with and even slightly better than the CPU counterpart; chained hashing on the GPU is an attractive method, which is similar to the GPU linear probing and in the meantime it is a dynamic scheme.

(4) Cuckoo hashing is not an ideal candidate and slower than linear probing and chained hashing. Using four tables could increase the practicability of Cuckoo hashing in terms of load factors; however, the performance decreases.

(5) Neither of the two versions of chained hashing is the absolute winner. From the perspective of the best performance that could be provided, two methods perform similarly. Considering the extra memory requirement of the variant of chained hashing, we only present the standard chained hashing in the rest of this paper.

Effects of hash functions

In the previous two sets of experiments, Multiply-shift is the primary hash function. In the third set of experiments, the performance of Murmur hashing integrated with six major grouping methods is examined to show the effects of different hash functions. In order to see the gap clearly, the ratio of two execution times occupied by a certain method using Multiply-shift and Murmur hashing respectively is shown. The ratio larger than one indicates that Murmur hashing performs better than Multiply-shift. Due to space limitation, only the results when the load factor is 50% are illustrated in Fig. 11a to d. Tables 2 and 3 give the values in the form of m/n where m is the number of ratios greater than one and n the number of points that lie on the line. "L-I-C" is used as a shorthand notation for Linear-I-CPU, and so on.

The main observations are as follows:

(1) Multiply-shift outperforms Murmur hashing in most cases. Considering all the test scenarios in Tables 2 and 3, only in 22% cases Murmur hashing runs faster;

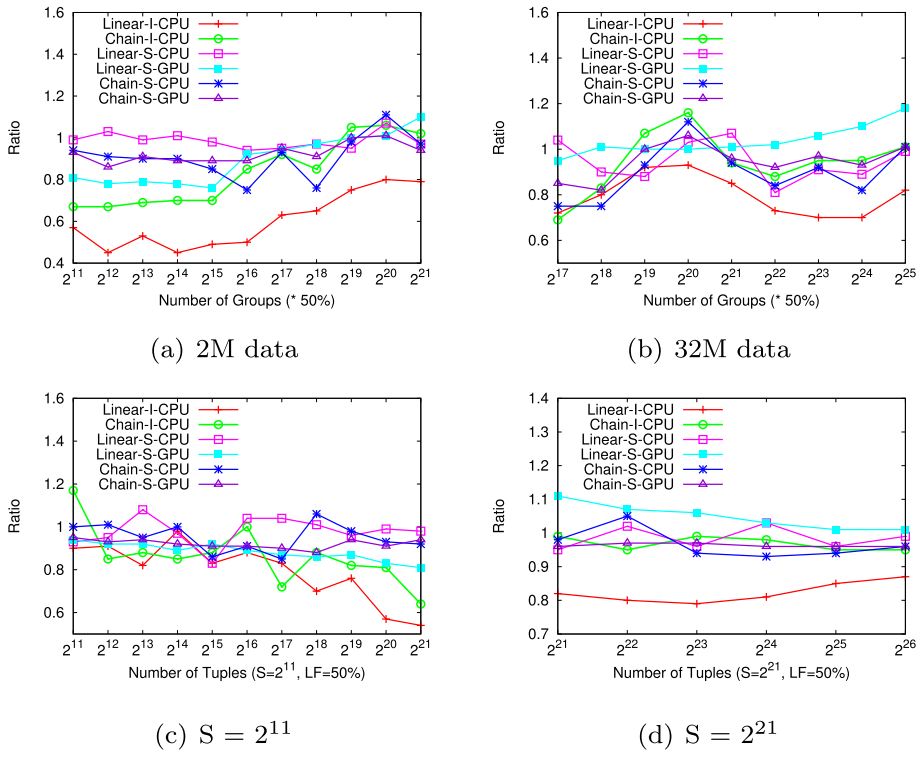
(2) Even if Murmur hashing is better, the performance gap between the two hash functions is much narrower than the performance degradation it brings in other cases;

(3) A pattern could be identified loosely that is the larger the load factor, the worse performance Murmur hashing will produce compared with Multiply-shift.

Thus Multiply-shift is a good candidate which is used in other experiments.

Effects of data distributions

In the fourth set of experiments, we compare six algorithms on the skewed data which obeys the Zipf distribution. In order to fully understand how these different methods per-



(a) 2M data (b) 32M data

(c) $S = 2^{11}$ (d) $S = 2^{21}$

Fig. 11 Time ratios of various methods combined with Multiply-shift and Murmur hashing

form when the group keys are skewed, the data sets are generated with the parameter of 3, which means that 80% of the tuples belong to one group. Under these circumstances, the actual group cardinality is much smaller than the expected one due to the characteristics of Zipf's law. In Fig. 12a to d, the values on X-axis and the axis title represent the parameters used to produce the test data sets which finally contain several hundred distinct keys. The tuple counts are the same with those in the previous experiments. In the case of the same hash table size, the number of collisions is reduced greatly.

From the figures, it could be found that the independent strategy combined with linear probing and chained hashing shows the best performance in most cases. The reason may be that the number of groups is significantly less than that on the uniform data. Many tuples are associated with one key. The independent strategy demonstrates its superiority on the low group cardinality. The performance in the third figure is similar to that in Fig. 8b which is reasonable since there exist fewer groups in the original test data. Using the shared hash table, linear probing and chained hashing on the GPU are faster than the CPU versions in many cases. Sometimes as shown in the first two figures, the two

Table 2 The performance of Murmur hashing on data sets of different sizes

Data set	Load factor	L-I-C	C-I-C	L-S-C	L-S-G	C-S-C	C-S-G
2M	25%	0/11	2/11	6/11	6/11	2/11	2/11
	50%	0/11	3/11	3/11	3/11	1/11	2/11
	90%	0/11	1/11	0/11	0/11	1/11	2/11
32M	25%	0/9	2/9	2/9	8/9	0/9	2/9
	50%	0/9	3/9	3/9	8/9	2/9	3/9
	90%	0/9	4/9	0/9	0/9	3/9	4/9

Table 3 The performance of Murmur hashing with groups of different numbers

Data set	Load factor	L-I-C	C-I-C	L-S-C	L-S-G	C-S-C	C-S-G
$S=2^{11}$	25%	2/11	2/11	8/11	0/11	3/11	0/11
	50%	0/11	2/11	4/11	0/11	4/11	0/11
	90%	2/11	3/11	2/11	0/11	1/11	0/11
$S=2^{21}$	25%	0/6	2/6	6/6	6/6	5/6	5/6
	50%	0/6	0/6	2/6	6/6	1/6	0/6
	90%	0/6	1/6	0/6	0/6	0/6	0/6

methods on the coupled GPU become the best candidates and we will investigate the reason in the next section.

Time analysis

Before the time analysis, we first give the standard deviation in calculating the average execution time on three runs in Table 4. The workload denoted by "Wi" includes the previous uniform and skewed data sets where the load factor is set to 50%. For example, "W0" corresponds to the workload in Fig. 5b. For each algorithm, the standard deviation at each point in the figure is first acquired and then the average value is used to represent the standard deviation under the workload. From the results, it could be found that the more the execution time, the larger the standard deviation is. For skewed heavy workloads such as W5 and W7, the shared strategy especially on the coupled GPU produces a bigger standard deviation. In addition, linear probing is more stable than chained hashing in most cases.

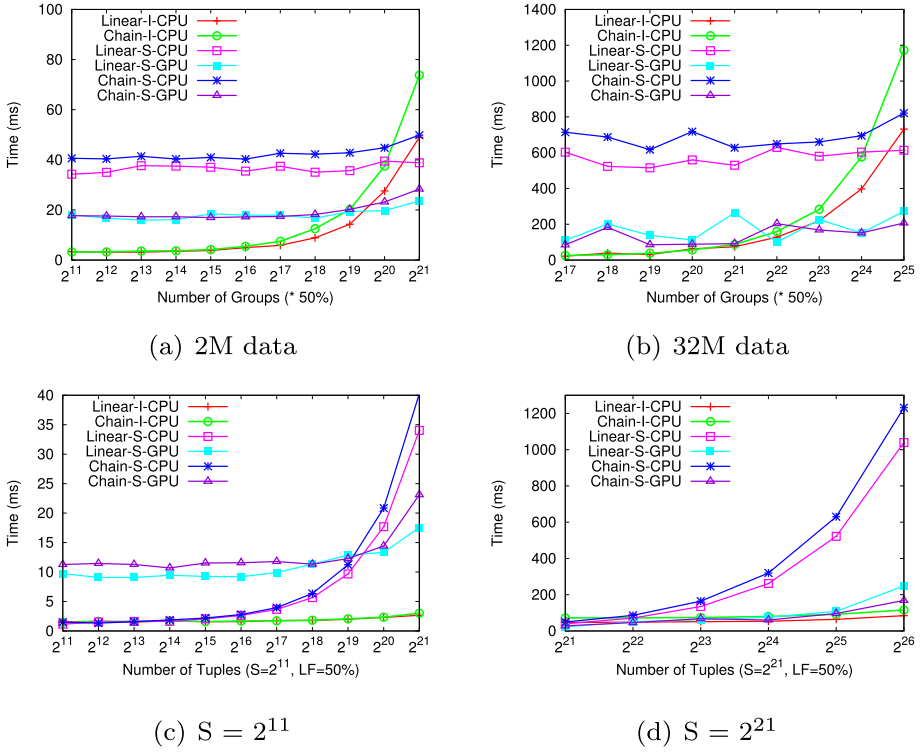


Fig. 12 Performance of various methods on the skewed data which obeys the Zipf distribution with the parameter of 3

Table 4 The standard deviation

Notation	Workload	L-I-C	C-I-C	L-S-C	L-S-G	C-S-C	C-S-G
W0	uniform data, 2M	0.51	1.30	0.34	0.40	0.54	0.37
W1	uniform data, 32M	12.07	17.56	9.78	2.22	14.25	2.56
W2	uniform data, S = 2^{11}	0.11	0.23	0.13	0.40	0.13	0.33
W3	uniform data, S = 2^{21}	3.80	10.33	5.95	0.45	7.02	0.47
W4	skewed data, 2M	0.74	0.33	0.34	1.51	0.63	0.39
W5	skewed data, 32M	5.51	5.88	22.51	62.62	54.08	38.08
W6	skewed data, S = 2^{11}	0.08	0.11	0.11	0.51	0.12	1.44
W7	skewed data, S = 2^{21}	2.28	4.32	2.56	14.60	7.95	9.37

Every grouping and aggregation algorithm is composed of four phases three of which are implemented by kernels. We break the execution time down to the time spent on each phase to exhibit the performance bottleneck. Figure 13a to d show the time breakdown in four cases which include two points (2^{18} and 2^{24} groups) in Fig. 6b and two (2^{21} and 2^{26} tuples) in Fig. 9b. Figure 14a to d are the results on the skewed data. In the figures, "build", "gather" and "aggregate" represent three kernels respectively, and "other" mainly refers to the process of creating and initializing data structures.

For the methods using the shared strategy, the first kernel *build_hash_table* that builds the hash table dominates the total execution. For the independent strategy, the first kernel is not the only time-consuming operation. When the hash table size is larger, the independent strategy spends more time on the first phase ("other") since multiple hash tables are needed to be created, which could be observed in Figs. 13b and 14b. This explains the

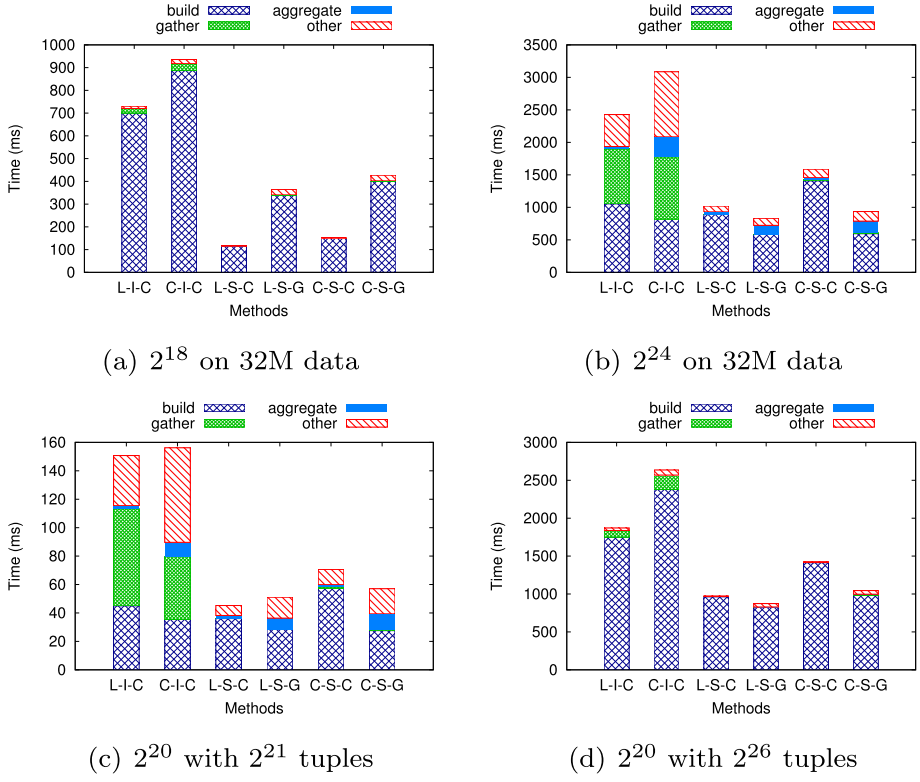


Fig. 13 Time breakdown in four cases which include two points (2^{18} and 2^{24} groups) in Fig. 6b and two (2^{21} and 2^{26} tuples) in Fig. 9b

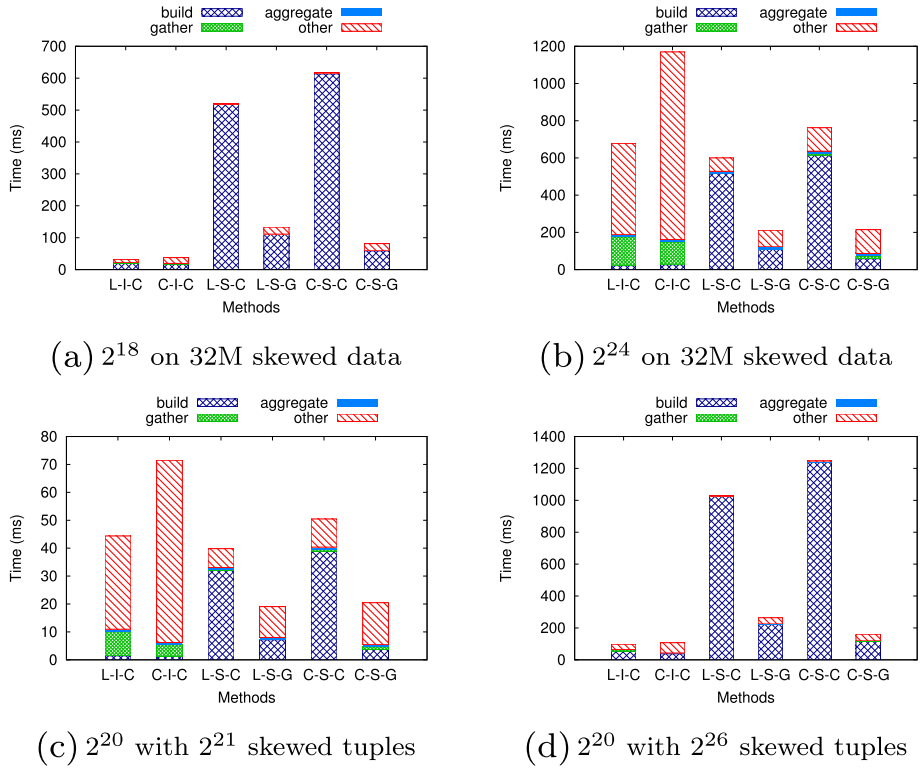


Fig. 14 Time breakdown in four cases on the skewed data

performance degradation of two independent methods on the skewed data in the previous experiments. The time on the first phase for two independent methods in Fig. 13c and d is similar and just the proportion in the total execution time is different. This is also true when the data is skewed. As the group cardinality is increased, the second kernel *gather_item_num* to generate a global hash table from separate tables occupies much time. Thus if the independent hash tables are used, three of the four phases could become the main contributors to the execution time.

Table 5 shows the LLC miss count of each method obtained from Intel VTune Profiler. Overall, the independent hash table exhibits more cache misses than the shared strategy on both the CPU and the coupled GPU. The relative performance in cache misses of various approaches is similar to that of the execution time on the uniformly distributed data sets. In the cases on skewed data, although Linear-S-CPU and Chain-S-CPU have few

Table 5 LLC miss count

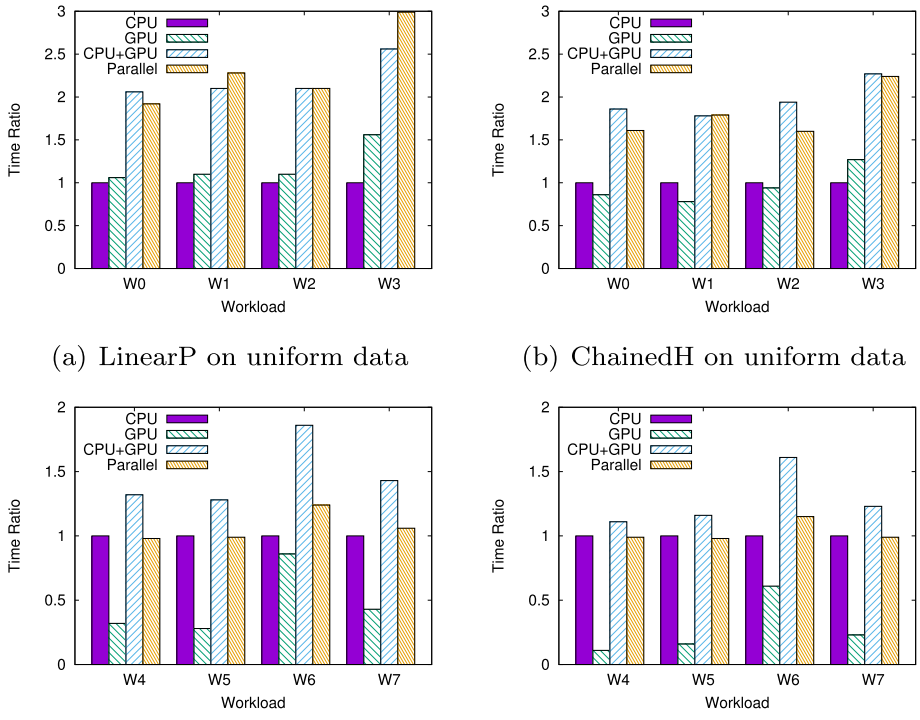
Data set	L-I-C	C-I-C	L-S-C	L-S-G	C-S-C	C-S-G
Figure 13a	39,246,666	43,680,000	233,333	280,000	280,000	303,333
Figure 13b	44,846,667	169,026,667	34,580,000	256,667	59,010,000	420,000
Figure 13c	3,220,000	7,700,000	735,000	93,333	2,310,000	210,000
Figure 13d	59,243,333	173,845,000	36,330,000	280,000	90,976,667	280,000
Figure 14a	443,333	466,666	140,000	70,000	163,333	70,000
Figure 14b	3,990,000	4,025,000	840,000	210,000	980,000	210,000
Figure 14c	385,000	210,000	233,333	70,000	70,000	70,000
Figure 14d	1,166,667	945,000	256,667	140,000	606,667	163,333

LLC misses, they spend more time than Linear-I-CPU and Chain-I-CPU on three kernels. The reason is that the two methods using shared hash tables produce more retired instructions or higher CPI. For instance, in Fig. 14d, the number of retired instructions of Chain-I-CPU and Chain-S-CPU is 2,550,960,000 and 5,271,120,000 respectively and the CPI is 1.22 and 8.77. There are more instructions for Chain-S-CPU and on average each instruction needs more cycles. Thus it runs slowly.

Co-processing

Since the modern architecture provides the additional GPU accelerator, we explore the co-processing potentials the coupled GPU could bring in the experiments. Firstly, an operator-level co-processing strategy is adopted to execute different queries on the CPU and the GPU simultaneously, which simulates real application scenarios of databases. We use the previously used eight kinds of workloads at 50% load factor, four of which are based on the uniform distribution and four are skewed data. "Parallel" means every two queries are assigned to the CPU and the GPU until no more two queries in the workload need to be executed. After two queries are finished, the next two queries start. The total sum of time is compared with the time when the corresponding part of queries are independently executed on the CPU and the coupled GPU respectively. The relative time ratio is shown in the figures. If the parallel time is less than the sum of the CPU time and GPU time denoted by "CPU+GPU", this illustrates the co-processing paradigm could help to increase the performance.

In Fig. 15a and b, the co-processing strategy is a little better than the mechanism on the single processor and the maximum speedup is 18%. Notice, in some cases for exam-



(a) LinearP on uniform data (b) ChainedH on uniform data

(c) LinearP on skewed data (d) ChainedH on skewd data

Fig. 15 Performance of the operator-level co-processing strategy

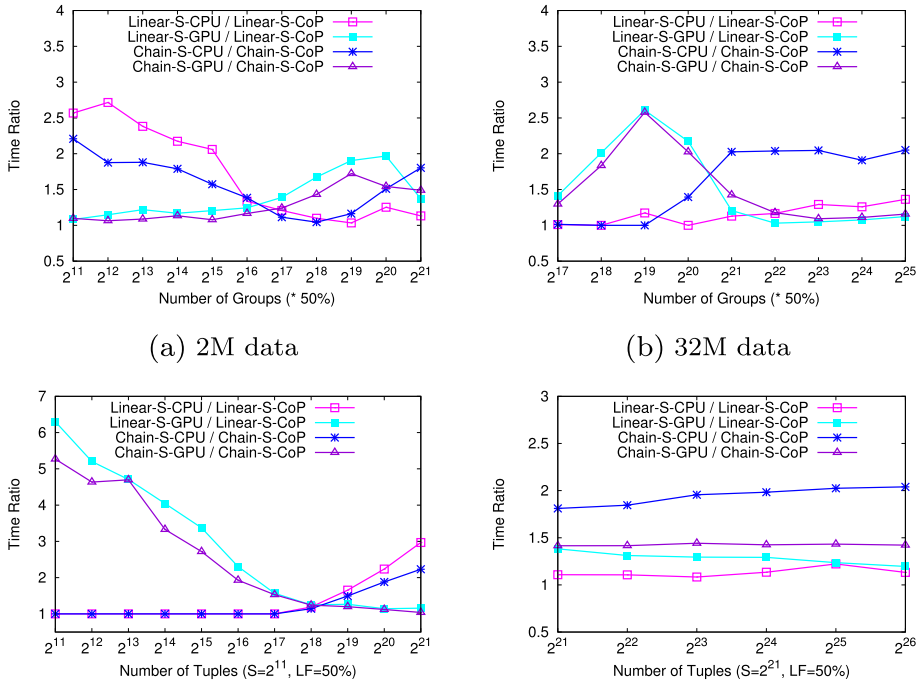
ple W1 and W3 workloads in Fig. 15a the parallel approach reduces the performance slightly. These workloads are heavier for both the CPU and the GPU than W0 and W2, that is there are more tuples or groups, and for these workloads, the CPU and the coupled GPU are likely to compete for cache and memory resources which will affect the overall performance. In Fig. 15c and d, the workloads contain skewed data and the performance gap between the CPU and the GPU is larger. Thus, the competition is reduced and co-processing could provide better performance. For linear probing, the speedup is between 23 and 33% and for chained hashing, it ranges from 11 to 29%.

Secondly, an intra-operator co-processing method is used to provide fine-grained parallelism. After the CPU completes the task of the first phase, the CPU and the coupled GPU handle different raw data in the second phase to establish a shared hash table. The workload distribution belonging to each processor is based on the previous execution time on separate processors. According to the percentage of the GPU time to the sum of the CPU and GPU time spent on the building phase, an initial CPU ratio is acquired. Then, considering the communication cost between the CPU and the GPU, the actual data ratio handled by the CPU is set empirically at an optimal value around the initial ratio which is a multiple of 0.05 in the experiments. We use this straight workload division way to verify the effectiveness of co-processing. At last, the left job is assigned to the CPU unless the second phase is totally finished by the GPU.

In order to build the hash table by all the work items of the CPU and the GPU, the methods are implemented using OpenCL 2.0 with the Shared Virtual Memory (SVM) feature and the corresponding experiments are done on Intel Core i7-8700 which is equipped with 6 cores and Intel UHD Graphics 630 and supports the SVM technology. SVM allows virtual addresses to be shared between the host and all devices in a context and enables pointer-based data structures. The SVM atomics could be used to guarantee concurrent accesses by the CPU and the GPU to the same memory address. The same workloads tested above are considered here and the ratios of single processor methods to the co-processing approach are shown in Figs. 16a to 17d. “CoP” indicates that the CPU and the GPU work in parallel within the operator.

The data sets used in Fig. 16a to d are uniformly distributed. If not considering Fig. 16c, in most cases the intra-operator co-processing method is faster than the corresponding version on the CPU and the coupled GPU for both linear probing and chained hashing. The average speedup over the CPU when linear probing is used is 1.7, 1.2, and 1.1 respectively in Fig. 16a, b, and d and the simultaneous speedup over the GPU is 1.4, 1.5, and 1.3. When chained hashing is adopted, the average time ratio is 1.6, 1.6, and 1.9 for the CPU and 1.3, 1.5, and 1.4 for the GPU. The higher the speedup, the corresponding processor runs more slowly than the counterpart and could acquire better performance improvement by adding extra computing power. In Fig. 16c, the absolute execution time is too short especially for the CPU and there exists a large gap between two processors when the number of tuples is small. Thus, the better choice is to use the CPU only to conduct the query not introducing the coupled GPU as well as the additional communication and contention cost. With the increase of the tuple count, the speedup acquired by the co-processing strategy over the CPU goes up and the maximum value is 3.0 and 2.2 for linear probing and chained hashing respectively.

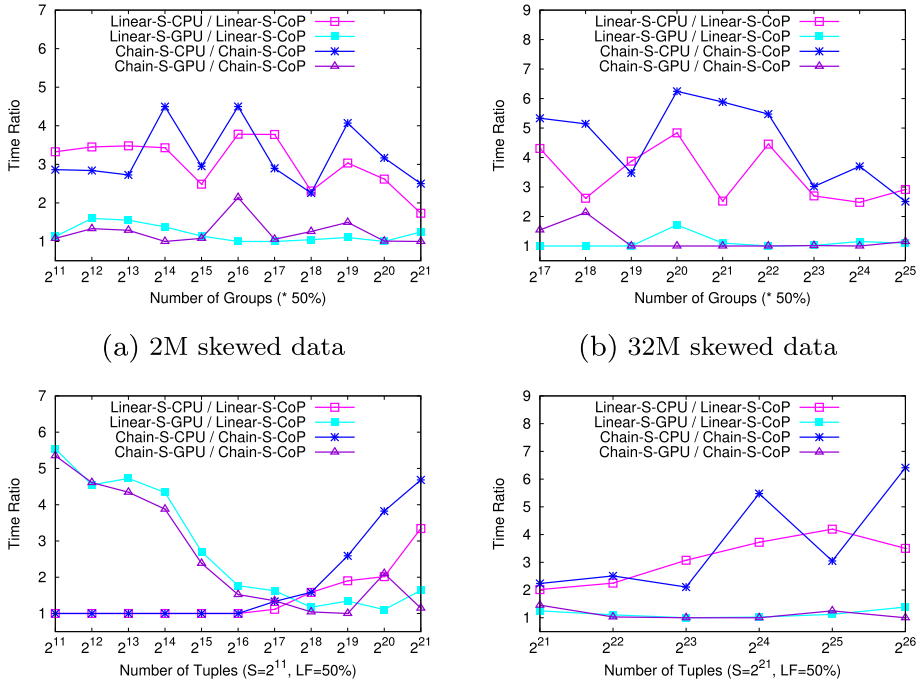
The results of the data under the Zipf distribution are shown in Fig. 17a to d. Since for the skewed data the GPU methods perform much better than the CPU counterparts, the



(a) 2M data (b) 32M data

(c) $S = 2^{11}$ (d) $S = 2^{21}$

Fig. 16 Performance of the intra-operator co-processing strategy on the uniform data



(a) 2M skewed data (b) 32M skewed data

(c) $S = 2^{11}$ (d) $S = 2^{21}$

Fig. 17 Performance of the intra-operator co-processing strategy on the skewed data

workload proportion belonging to the CPU is very small which is 5% in the majority of cases and 10% at most. Sometimes the coupled GPU itself shows the best performance. Except the third figure, the average speedup of the co-processing strategy over the GPU method in every figure is around 1.2. The behavior in the third figure is similar to that in Fig. 16c that is the CPU is the top performer at first.

Generally the empirical optimal workload ratio owned by the CPU is smaller than the initial CPU ratio and the performance of the co-processing mechanism is affected by the used value. Thus, an advanced optimizer in databases should be designed in future to produce adaptive execution strategies under considering comprehensive factors from CPU, GPU, and memory.

Discussion

The used example has two columns and each column entry is 4-byte integer. Assume there are n tuples in the table then the data size is $8n$. The CPU time is roughly expressed as $8n/B_m + T_o$, where B_m is the memory bandwidth, and T_o is the other execution time. If the query is memory-bounded, T_o could be ignored. For a discrete GPU, if through optimization the execution on the GPU could be overlapped with the data transfer via the PCIe bus, the runtime is $8n/B_p$, where B_p is the PCIe bandwidth. Usually, B_m is larger than B_p thus it is quite possible the CPU outperforms the discrete GPU. Although the coupled architecture is our main research object, ideally the CPU and two kinds of GPUs are not contradictory and the study on the coupled architecture could be combined into the technologies on discrete GPUs. A better performance may be gained if all the computing resources such as CPUs, coupled GPUs, and discrete GPUs are utilized fully and simultaneously in future.

Related work

On the traditional multi-core CPU, there have existed many studies on group-by and aggregations [1–3, 19]. Cieslewicz et al. [1] focused on chip multiprocessors to study different strategies and factors of affecting performance and introduced an adaptive aggregation operator to choose the correct strategy. The used hash function was multiplicative hashing and chaining was utilized to resolve hash collisions. The research [2] found that for aggregation the complexity of hashing was the same as that of sorting and mixed hashing and sorting routines in the same algorithmic framework to leverage the advantages of both approaches. Ye et al. [3] provided a solution to perform parallel aggregation on the Intel Nehalem architecture and proposed two algorithms to acquire good performance. Gubner et al. [19] studied optimizations using SIMD instructions and presented an in-register aggregation technique which can be vectorized easily. A seven-dimensional analysis of hashing methods [15] have been performed on Intel CPU processor using indexing as a use-case. Some dimensions are also considered in this paper such as hash functions and hashing schemes. The differences include the following aspects: the emerging coupled CPU-GPU architecture is our goal and we are interested in the performance of different methods on the coupled GPU and two processors besides only the CPU; the grouping and aggregation operation is our research purpose which is different from indexing; the parallel methods not the single-threaded hashing are studied carefully in our work.

As discrete GPUs have become popular, they have been used as accelerators in database research [5, 6, 8–10, 20–22]. Karnagel et al. [5] studied offloading the grouping and aggregation operator to a discrete GPU and provided the corresponding analysis and a model with a heuristic optimizer. Compression, query-compilation-like fusion and a scheduling mechanism were investigated by the research work [6] to use both the CPU and the GPU for TPC-H Q1. There have also appeared some studies on coupled architectures [4, 7, 11–13, 23–25]. Power et al. [4] explored scan-aggregate queries on the CPU, the discrete GPU, and the integrated GPU and found that an integrated GPU was faster than a discrete GPU and a CPU. Linear probing was used to implement the hash table and MurmurHash was the hash function. Rosenfeld et al. [7] examined the influence of different parameters on hash aggregation on five discrete GPUs and one integrated GPU and presented an algorithm to optimize execution parameters. Multiply-shift and linear probing were used in the work. The paper [25] provided the first step of the work to improve group-by and aggregation by using the coupled GPU. The research object was the grouping and aggregation based on chained hashing, one of the common hashing schemes, and in terms of hash functions, only Multiply-shift was used directly. The contribution was to present a co-processing approach which firstly implemented data parallelism using a shared hash table. Despite the previous work from researchers and us, a systematic analysis of grouping and aggregation algorithms on such architectures is still missing, which is the original intention of this paper. In this paper, eight different dimensions are considered in the extensive experimental study and analysis, including five hashing schemes, two hash functions, six low load factors and high ones, both shared and independent hash tables and two kinds of co-processing strategies. We hope the current work could provide more meaningful insights into the operator behavior on the coupled CPU-GPU architecture and some meaningful conclusions could help database researchers to design the high-performance algorithms on modern processors.

In terms of optimization strategies on hybrid heterogeneous architectures, many other research findings have been provided [26–36]. Heuristical and machine-learning-based load distribution and balancing models on a hybrid CPU-GPU database systems were proposed by the authors [26] and multiple models such as linear regression, random forest, and Adaboost were used to dynamically decide the processing unit. Nozal et al. [27] designed an OpenCL-based runtime system EngineCL to simplify the co-execution of data-parallel kernels on different devices of a heterogeneous system. The research work [28] implemented an abstract entity Multi-Controler in a library to coordinate the management of heterogeneous devices. In the paper [29] a field-programmable gate array was utilized to accelerate query processing in a searchable encrypted database system and a cache function was proposed to improve the access. Singh et al. [30] focused on an energy-efficient run-time mapping and thread partitioning method to execute concurrent OpenCL applications on the CPU and the GPU cores. The authors [31] integrated the FPGA (field-programmable gate array) into EngineCL and effective cooperation among the CPU, GPU and FPGA were implemented. An OpenCL runtime FluidiCL was proposed by Pandit et al. [32] to execute a program coded for a single device on both the CPU and the GPU. Most of these studies are related to OpenCL. Wang et al. [33] studied how to migrate a CUDA-based ultrasound imaging code to oneAPI code to run on different hardware. Intel oneAPI was equipped with co-processing strategies in the work [34] to execute the same kernel between different devices and static and dynamic policies were

exploited. Hammond et al. [35] provided a comparative analysis of Kokkos and SYCL programming models. The study [36] analyzed the feasibility of solving large-scale Markov decision processes with value iteration in low-power heterogeneous platforms using different programming approaches and scheduling strategies. SYCL, a Khronos standard for heterogeneous programming build on C++, is the model used in these studies. Overall, interesting and helpful insights in the acceleration technologies on hybrid heterogeneous architectures could been found.

Conclusions

In this paper, we have studied in depth group-by and aggregation on CPU-GPU processors. Our observations are described throughout the paper. The overall conclusions are as follows: (1) If the workload is light, for example, the data set and/or group cardinality are small, the independent hash table combined with linear probing or chained hashing on the CPU is preferred. As the group cardinality or the data set gets bigger, linear probing on the CPU using a shared hash table is the best. If the workload keeps getting heavier, the shared GPU linear probing is a candidate that could be considered, and the shared GPU chained hashing is also a good choice in practice in terms of dynamically resizing the hash table. (2) Cuckoo hashing, the variant of chained hashing and Murmur hashing have some disadvantages and are not the first choices. (3) For both linear probing and chained hashing, the grouping and aggregation could benefit from co-processing on the hybrid architecture. The coupled GPU plays a positive role no matter whether it is used as a stand-alone processor or jointly with the CPU.

For the performance optimization, various co-processing mechanisms should be explored further on the coupled architecture, such as adaptive strategy selection or even different workloads and hashing methods on different processors. For future work, we will consider optimization techniques in the direction and design fine algorithms on choosing correct strategies.

Abbreviations

GPU	Graphics processing unit
PCIe	Peripheral component interconnect express
CPU	Central processing unit
LLC	Last level cache
MB	Megabyte
EDRAM	Embedded dynamic random access memory
EU	Execution unit
SIMD	Single instruction multiple data
OpenCL	Open computing language
API	Application programming interface
STL	Standard template library
DBMS	Database management system
GB	Gigabyte
SVM	Shared virtual memory
FPGA	Field-programmable gate array
CPI	Cycles per instruction retired

Acknowledgements

Not applicable

Authors' contributions

The corresponding author, H.L., designed and performed the experimental study and was a major contributor in writing the manuscript. L.C. participated in the discussion about the performance analysis and reviewed and edited the article. The authors read and approved the final manuscript.

Funding

This research was funded by the National Key R&D Program of China grant number 2020YFC1523300 and a grant from the Capital Science and Technology Innovation Vouchers of China.

Availability of data and materials
All data generated or analyzed during this study are included in this published article.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]School of Artificial Intelligence, Beijing Normal University, Beijing, China. [2]Oushu Inc., Beijing, China.

Received: 14 February 2022 Accepted: 30 May 2022
Published online: 22 June 2022

References

1. Cieslewicz J, Ross KA (2007) Adaptive aggregation on chip multiprocessors. In: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, 23-27 September 2007. ACM, New York. pp 339–350
2. Müller I, Sanders P, Lacurie A, Lehner W, Färber F (2015) Cache-efficient aggregation: Hashing is sorting. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 04 2015. ACM, New York. pp 1123–1136
3. Ye Y, Ross KA, Vesdapunt N (2011) Scalable aggregation on multicore processors. In: Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, 13 June 2011. ACM, New York. pp 1–9
4. Power J, Li Y, Hill DM, Patel MJ, Wood AD (2015) Toward GPUs being mainstream in analytic processing. In: Proceedings of the 11th International Workshop on Data Management on New Hardware, DaMoN 2015, Melbourne, Victoria, Australia, May 31 - June 04 2015. ACM, New York. pp 11:1–11:8
5. Karnagel T, Müller R, Lohman MG (2015) 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, 31 August 2015. pp 13–24
6. Tomé GD, Gubner T, Raasveldt M, Rozenberg E, Boncz AP (2018) Optimizing group-by and aggregation using GPU-CPU co-processing. In: International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, 27 August 2018. pp 1–10
7. Rosenfeld V, Breß S, Zeuch S, Rabl T, Markl V (2019) Performance analysis and automatic tuning of hash aggregation on GPUs. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019. ACM, New York. pp 8:1–8:11
8. Kaldewey T, Lohman MG, Müller R, Volk P (2012) GPU join processing revisited. In: Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, 21 May 2012. ACM, New York. pp 55–62
9. Yuan Y, Lee R, Zhang X (2013) The yin and yang of processing data warehousing queries on GPU devices. Proc VLDB Endowment 6(10):817–828
10. Shanbhag A, Madden S, Yu X (2020) A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, 14-19 June 2020. ACM, New York. pp 1617–1632
11. He J, Lu M, He B (2013) Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. Proc VLDB Endowment 6(10):889–900
12. Luan H, Chang L (2017) An evaluation of analytical queries on CPUs and coupled GPUs. Concurrency Comput Pract Exp 29(5):e3982
13. Delorme MC, Abdelrahman TS, Zhao C (2013) Parallel radix sort on the AMD fusion accelerated processing unit. In: 42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, 1-4 October 2013. IEEE Computer Society, Washington, DC. pp 339–348
14. Khronos The OpenCL specification. https://www.khronos.org/registry/cl/specs. Accessed Oct 2021
15. Richter S, Alvarez V, Dittrich J (2015) A seven-dimensional analysis of hashing methods and its implications on query processing. Proc VLDB Endowment 9(3):96–107
16. Pagh R, Rodler FF (2004) Cuckoo hashing. J Algoritm 51(2):122–144
17. Dietzfelbinger M, Hagerup T, Katajainen J, Penttonen M (1997) A reliable randomized algorithm for the closest-pair problem. J Algoritm 25(1):19–51
18. Appleby A Murmurhash project. https://github.com/aappleby/smhasher. Accessed Oct 2021
19. Gubner T, Boncz PA (2017) Exploring query execution strategies for JIT, vectorization and SIMD. In: International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, 1 September 2017. pp 9–17
20. Pirk H, Manegold S, Kersten LM (2014) Waste not... efficient co-processing of relational data. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4 2014. IEEE Computer Society, Washington, DC. pp 508–519
21. Wang K, Zhang K, Yuan Y, Ma S, Lee R, Ding X, Zhang X (2014) Concurrent analytical query processing with GPUs. Proc VLDB Endowment 7(11):1011–1022
22. Sioulas P, Chrysogelos P, Karpathiotakis M, Appuswamy R, Ailamaki A (2019) Hardware-conscious hash-joins on GPUs. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, 8-11 April. IEEE Computer Society, Washington, DC Vol. 2019. pp 698–709

23. Huang H, Luan H (2020) Optimizing B^+-tree searches on coupled CPU-GPU architectures. In: Algorithms and Architectures for Parallel Processing - 20th International Conference, ICA3PP 2020, New York, USA, 2-4 October 2020. Springer, Cham. pp 401–415
24. Huang H, Luan H (2021) Rethinking insertions to B^+-trees on coupled CPU-GPU architectures. In: 19th IEEE International Symposium on Parallel and Distributed Processing with Applications, IEEE ISPA 2021, New York, USA, September 30 - October 03 2021. IEEE Computer Society, Washington, DC. pp 993–1001
25. Luan H, Fu Y (2021) Accelerating group-by and aggregation on heterogeneous CPU-GPU platforms. In: 2021 17th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery, ICNC-FSKD 2021, Guiyang, China, 24-26 July 2021. Springer, Cham. pp 980–990
26. Abdennebi A, Elakas A, Tasyaran F, Öztürk E, Kaya K, Yildirim S (2021) Machine learning-based load distribution and balancing in heterogeneous database management systems. Concurrency Comput Pract Exp 34(4):e6641
27. Nozal R, Bosque J, Beivide R (2020) EngineCL: Usability and performance in heterogeneous computing. Futur Gener Comput Syst 107:522–537
28. Moreton-Fernandez A, Llanos D (2019) Multi-device controllers: a library to simplify parallel heterogeneous programming. Int J Parallel Prog 47(1):94–113
29. Okada M, Suzuki T, Nishio N, Waidyasooriya H, Hariyama M (2020) FPGA-accelerated searchable encrypted database management systems for cloud services. IEEE Trans Cloud Comput 10(2):1373–1385
30. Singh A, Prakash A, Reddy B, Merrett G, Al-Hashimi B (2017) Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs. ACM Trans Embed Comput Syst 16(5s):147:1–147:22
31. Dávila-Guzmán M, Nozal R, Tejero R, Villarroya-Gaudó M, Gracia D, Bosque J (2019) Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. J Supercomput 75(3):1732–1746
32. Pandit P, Govindarajan R (2014) Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In: 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, 15-19 February 2014. ACM, New York
33. Wang Y, Zhou Y, Wang Q, Wang Y, Xu Q, Wang C, Peng B, Zhu Z, Takuya K, Wang D (2021) Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI. In: IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, 17-21 June 2021. IEEE Computer Society, Washington, DC. pp 360–370
34. Nozal R, Bosque J (2021) Straightforward heterogeneous computing with the oneAPI coexecutor runtime. Electronics 10(19):2386
35. Hammond J, Kinsner M, Brodman J (2019) A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications. In: Proceedings of the International Workshop on OpenCL, IWOCL 2019, Boston, MA, USA, 13-15 May 2019. ACM, New York. pp 15:1–15:2
36. Constantinescu D, Navarro A, Corbera F, Asenjo R (2021) Fernández-Madrigal, J. J Supercomput 77(1):44–65

Publisher's Note