In-Cache Query Co-Processing on Coupled CPU-GPU Architectures

Jiong He   Shuhao Zhang   Bingsheng He
Nanyang Technological University

ABSTRACT

Recently, there have been some emerging processor designs that the CPU and the GPU (Graphics Processing Unit) are integrated in a single chip and share Last Level Cache (LLC). However, the main memory bandwidth of such coupled CPU-GPU architectures can be much lower than that of a discrete GPU. As a result, current GPU query co-processing paradigms can severely suffer from memory stalls. In this paper, we propose a novel *in-cache* query co-processing paradigm for main memory On-Line Analytical Processing (OLAP) databases on coupled CPU-GPU architectures. Specifically, we adapt CPU-assisted prefetching to minimize cache misses in GPU query co-processing and CPU-assisted decompression to improve query execution performance. Furthermore, we develop a cost model guided adaptation mechanism for distributing the workload of prefetching, decompression, and query execution between CPU and GPU. We implement a system prototype and evaluate it on two recent AMD APUs A8 and A10. The experimental results show that 1) in-cache query co-processing can effectively improve the performance of the state-of-the-art GPU co-processing paradigm by up to 30% and 33% on A8 and A10, respectively, and 2) our workload distribution adaption mechanism can significantly improve the query performance by up to 36% and 40% on A8 and A10, respectively.

1. INTRODUCTION

Query co-processing paradigm on GPUs has been an effective means to improve the performance of main memory databases for OLAP (e.g., [15, 17, 22, 28, 25, 13, 30, 29]). Currently, most systems are based on discrete CPU-GPU architectures, where the CPU and the GPU are connected via the relatively slow PCI-e bus. Recently, some emerging processor designs that the CPU and the GPU are integrated in a single chip and share LLC. For example, the AMD Accelerated Processing Unit (APU) architecture integrates CPU and GPU in a single chip, and Intel released their latest generation Ivy Bridge processor in late April 2012. On those

emerging heterogeneous architectures, the low speed of PCI-e is no longer an issue. Coupled CPU-GPU architectures call for new data processing mechanisms. There have been studies on more collaborative and fine-grained schemes for query co-processing [19, 38] and other data processing workloads (e.g., key-value stores [21] and MapReduce [7]).

Despite the effectiveness of previous studies on query co-processing on coupled architectures, both CPU and GPU execute *homogeneous* workloads in previous studies [19, 7, 38, 21]. However, due to the unique architectural design of coupled CPU-GPU architectures, such homogeneous workload distribution schemes can hinder query co-processing performance on the GPU. On the one hand, the GPU in the coupled architecture is usually less powerful than the one in the discrete architecture. On the other hand, the GPU in the coupled architecture accesses main memory (usually DDR3), which has a much lower bandwidth than the discrete GPU memory (usually GDDR5). These two factors lead to severe underutilization of the GPU in the coupled architecture because of memory stalls. The inherent GPU design of Single Program Multiple Data (SPMD) execution model and the in-order nature of GPU cores make the GPU in the coupled architecture more sensitive to memory stalls. In this paper, we investigate how to reduce memory stalls suffered by the GPU and further improve the performance of query co-processing on the coupled CPU-GPU architecture.

On the recent coupled CPU-GPU architectures, the computational capability of the GPU is still much higher than that of the CPU. For example, the GPU can have 5 and 6 times higher Giga Floating Point Operations per Second (GFLOPS) than the CPU on AMD APUs A8 and A10, respectively. The superb raw computational capability of the GPU leads to a very similar speedup if the input data is in cache. However, due to the above-mentioned impact of memory stalls on the GPU co-processing, the speedup is as low as 2 when the data cannot fit into cache (more detailed results can be found in Section 3.1). Thus, the natural question is whether we can and how to ensure that the working set of query co-processing can fit in the cache as much as possible to fully unleash the GPU power.

In this paper, we propose a novel *in-cache* query co-processing paradigm for main memory databases on coupled CPU-GPU architectures. Specifically, we adapt CPU-assisted prefetching to minimize the cache misses from the GPU and CPU-assisted decompression schemes to improve query execution performance. No matter whether or not the decompression is involved, our scheme ensures that the input data to the GPU query co-processing has been prefetched. Thus,

the GPU executions are mostly on in-cache data, without suffering from memory stalls. Specifically, unlike homogeneous workload distributions in previous query co-processing paradigms [19, 7], our workload distribution is *heterogeneous*: a CPU core can now perform memory prefetching, decompression, and even query processing, and the GPU can now perform decompression and query processing. We further develop a cost model guided adaptation mechanism for distributing the workload of prefetching, decompression, and query evaluations between the CPU and the GPU. Fine-grained resource allocation is achieved by *device fission* that divides the CPU or the GPU into smaller scheduling units (either by OpenCL runtime or our software-based approaches).

We implement a system prototype and evaluate it on two recent AMD APUs A8 and A10. The experimental results show that 1) in-cache query co-processing is able to effectively improve the performance of GPU query co-processing by up to 30% and 33% on A8 and A10, respectively, and 2) our cost model can effectively predict a suitable workload distribution, and our distribution adaption mechanisms significantly improve the query performance by 36-40%.

The remainder of this paper is organized as follows. In Section 2, we introduce the background and preliminary on coupled architectures and OpenCL. In Section 3, we elaborate the design and implementation of in-cache query co-processing, followed by the cost model in Section 4. We present the experimental results in Section 5. We review the related work in Section 6 and conclude in Section 7.

2. PRELIMINARIES AND BACKGROUND

This section introduces the background and preliminary on coupled architectures and OpenCL.

2.1 Heterogeneous System Architectures

Heterogeneous architectural designs are emerging in the field of computer architecture. Researchers have been proposing different heterogeneous designs in the modern/future processors, which attempt to improve the performance, reduce the energy consumption or both [2, 26]. This paper focuses on the coupled CPU-GPU architecture.

The design of the coupled architecture is illustrated in Figure 1. Both the CPU and the GPU are integrated in the same chip which removes the PCI-e bus. Besides, the CPU and the GPU share the L2 cache in this study, which enables the possibility of data reuse between them. In current AMD APUs, all data accesses should go through a unified north bridge (UNB) that connects the CPU, the GPU, and the main memory. Table 1 presents the hardware configurations of two generations of AMD APUs (i.e., A8-3870K and A10-7850K). For comparison, we also list the configuration of the latest Radeon R9 270 as an example of discrete GPU. The GPU in the coupled architecture has a much smaller number of cores at lower clock frequency because of chip area constraints. In the previous AMD APUs like A8 3870K, memory sharing is achieved by a relatively small zero-copy buffer. The latest Kaveri APUs like A10-7850K support Shared Virtual Memory (SVM) which extends memory sharing to the entire main memory space [23]. The memory bandwidth is relatively low (29.8GB/s), because DDR3 is specially designed for memory latency sensitive applications. For discrete GPUs, GDDR5 can provide up to 264GB/s bandwidth. As a matter of fact, a customized architecture design implemented in PlayStation 4 uses GDDR5 as main memory,

Table 1: Configuration of AMD Fusion A10-7850K.

	A8 3870K		A10 7850K		Radeon R9 270
Core type	CPU	GPU	CPU	GPU	GPU
# Cores	4	400	4	512	1280
Core frequency(MHz)	3000	600	4000	720	925
Shared memory (GB)	0.5		32(whole)		N/A
Peak memory bandwidth (GB/s)	5.6	24.5	7.8	28.9	179.2
Cache size(MB)	4		4		N/A
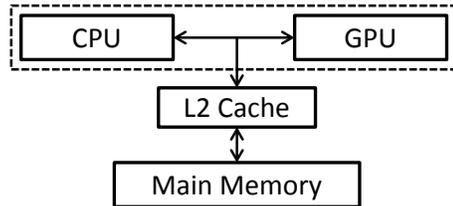
Figure 1: Overview of the coupled CPU-GPU architecture.

mainly for improving the graphics performance [33]. In this paper, we focus on coupled CPU-GPU architectures using DDR3 as main memory, which is more common in the current commodity market.

2.2 Unified Programming Interface

Open Computing Language (OpenCL) is a unified programming language for heterogeneous architectures. OpenCL programs can be coded once and run on any OpenCL-compatible devices. Existing studies [11, 34] have shown that programs in OpenCL can achieve very close performance to those in platform-specific languages such as CUDA for NVIDIA GPUs and OpenMP for CPUs. For example, Fang et al. [11] demonstrate that the CUDA-based implementations are at most 30% better than OpenCL-based implementations on NVIDIA GPUs. On CPUs, OpenCL even outperforms OpenMP in many scenarios [34].

All OpenCL-compatible devices are mapped to the same logical architecture, namely *compute device*. Each compute device consists of a number of Compute Units (CUs). Furthermore, each CU contains multiple processing elements running in the SPMD style. On the APU, the CPU and the GPU are programmed as two compute devices. Each CPU core is mapped as one CU, and each GPU CU is equivalent to one multi-processor. The piece of code executed by a specific device is called a *kernel*. A kernel employs multiple *work groups* for the execution, and each work group contains a number of *work items*. A work group is mapped to a CU, and multiple work items are executed concurrently on the CU. The execution of a work group on the target architecture is vendor specific. For instance, AMD usually executes 64 work items in a *wavefront* and NVIDIA with 32 work items in a *warp*. In this paper, we use AMD's terminology. All the work items in the same wavefront run in the Single Instruction Multiple Data (SIMD) manner.

Device fission is a new feature of OpenCL to support dividing a single device into multiple subdevices. With this feature, two different tasks can run on the same device concurrently. Thus, hardware resources on the same device can be shared among multiple tasks to achieve fine-grained resource allocation. Currently, device fission is fully supported on most OpenCL-compatible CPU devices. However, OpenCL does not support device fission on GPU devices.

(a) In-cache processing benefit   (b) Memory stalls on GPUs   (c) Wavefront efficiency
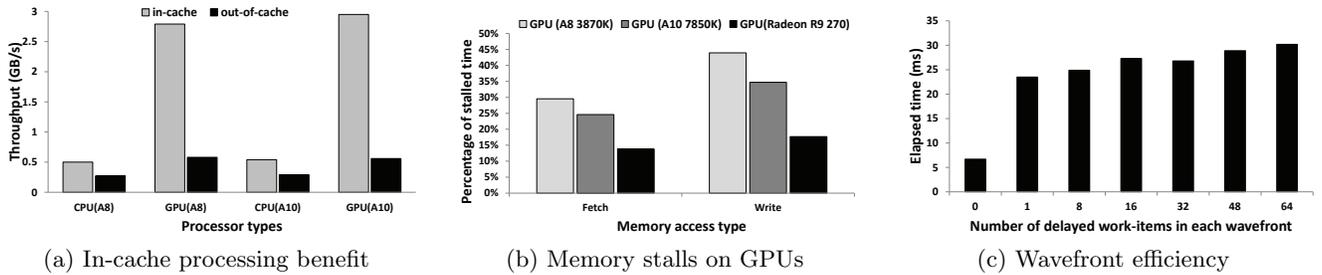
Figure 2: Motivations for in-cache query co-processing on APUs.

Thus, we develop a software-based approach to emulate device fission on the GPU (Subsection 3.2.3).

3. DESIGN AND IMPLEMENTATION

In this section, we first present the motivation for developing the in-cache query co-processing paradigm. We experimentally evaluate the memory performance of the GPU on two recent AMD APUs. The detailed experimental setup can be found in Section 5. Basically, we consider the following scenario: all the relations and indexes of databases are stored in the main memory. As the previous studies [14, 15], queries (mainly OLAP queries) can be executed on the CPU or the GPU in part or entirely. We aim at improving the efficiency of query co-processing of a single query on the coupled architecture, as the previous studies on query co-processing. We conduct the motivating experiments with the basic operations in databases running on the two APUs. On both platforms, we have made a number of common observations, which motivate in-cache query co-processing on the coupled architecture. Next, we present the detailed design and implementation of our proposed query co-processing paradigm. The cost model of guiding the workload adaptation is presented in Section 4.

3.1 Motivations

Our design of in-cache query co-processing is motivated by the following observations.

Observation 1: The in-cache processing performance of the GPU is much higher than that of the CPU. Figure 2a demonstrates the benefits gained from the cache for the CPU and the GPU. The experiment measures the throughput of performing many simple sequential scans on the same relation (we run 100 scans, but exclude the impact of compulsory misses in the first scan). The relation is initially stored in the memory. The cases for "in-cache" and "out-of-cache" represent the tables with sizes of 1MB and 16MB, respectively. In comparison with the CPU, the GPU has a much sharper jump after the relation size exceeds the L2 cache size. Therefore, the GPU can gain more performance benefits from the cache than the CPU, if the data resides in the L2 cache.

Observation 2: The memory bandwidth of the GPU in the coupled architecture is much lower than that of the GPU in the discrete architecture. Figure 2b shows the percentage of memory stalled time obtained from the AMD CodeXL profiler when the table scan runs on the coupled GPU and the discrete GPU in Table 1. In the coupled system, the memory bandwidth is more limited and more memory accesses are stalled. The GPU in the coupled architecture is more memory-bound than that in the discrete architecture. This result inspires us to find a way to reduce the size of

data accesses from the main memory for the GPU in a more aggressive manner.

Observation 3: The SPMD execution model and in-order core design of the GPU can severely degrade the GPU query co-processing performance. All work items in a work group are grouped and executed in a wavefront in a lock-step fashion on the GPU. Even if only one work item is delayed by a memory access, the entire work group is delayed. Figure 2c shows the performance of a table scan with random accesses on the GPU when the number of work items that is delayed due to L2 cache miss increases from 0 to 64. 0 means all the input data are in the cache, and 64 means all the work items in a work group have cache misses. When no work item is delayed, the elapsed time is quite short. However, the performance degrades sharply as long as delayed work items exist.

These observations challenge existing query co-processing paradigms. The state-of-the-art query co-processing paradigm [19] as well as other similar data co-processing paradigms [7, 21] on coupled CPU-GPU architectures fail to capture those features. Particularly, all previous studies assign *homogeneous* workloads to the CPU and the GPU. From these observations, the GPU is severely degraded by the memory stalls, which are usually a major performance factor for databases [27, 19]. Homogeneous workload distribution still causes excessive memory stalls on the GPU, despite the fine-grained and collaborative improvements in the previous studies [19, 38, 21, 7]. The performance degradation caused by memory stalls on the GPU is much more severe than that on the CPU. An ideal query co-processing performance should exploit the advantages of the GPU (i.e., much higher in-cache data processing performance) as much as possible.

3.2 Design and Implementation

We design and develop in-cache query co-processing by extending our OpenCL-based query processor OmniDB [38]. The system is designed to support OLAP and focuses on read-only queries. It does not support on-line updates. Instead, it rebuilds a relation for batch updates. Figure 3 shows the architectural overview, specifically designed for coupled CPU-GPU architectures (this study focuses on AMD APUs). Queries are processed by the query plan generator using a Selinger-style optimizer [31]. The APU-aware cost model captures the features of the coupled architecture and produces the predicted workload assignment plan and execution time for the query. We abstract three common functional modules in the in-cache query processing paradigm: prefetching (P), decompression (D, optional), and the actual query execution (E). Each CU can work on any unit of P/D/E. These functional modules are scheduled by the workload scheduler to available CUs. Our in-cache query
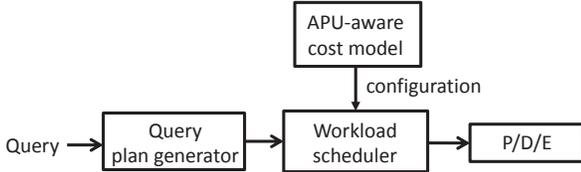
Figure 3: An overview of the system architecture.

(a) PE

(b) cPDE-c

(c) cPDE-b

(d) cPDE-g
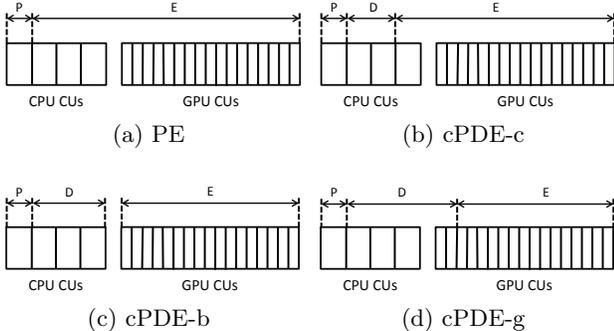
Figure 4: Four optimized execution configurations.

co-processing paradigm can be applicable to databases with or without compression [12]. If the data is compressed, decompression may be required before query executions. If not, data can be processed by query executions directly. Prefetching is exploited to hide the memory latency inside database operators, inspired by previous studies [8, 39].

Depending on how to assign the functional modules to all available CUs, we have four different execution configurations, as depicted in Figure 4.

In Figure 4a (PE), almost all CUs on the CPU and the GPU are assigned to query executions, leaving only one CPU CU to do prefetching. This is suitable for scenarios when the data is stored without compression, or query executions can be directly performed on compressed data. When decompression is required, there are three possible execution configurations, cPDE-c, cPDE-b, and cPDE-g. We define the *DE boundary* as the dividing position between the D and E functional modules. The value of DE boundary represents the number of CUs assigned to D, shifting from the CPU side to the GPU side. Figure 4b represents the case where the DE boundary is on the CPU, while Figure 4c and Figure 4d represent that the DE boundary is right on the boundary between the CPU and the GPU, and on the GPU, respectively. Additionally, we implement the state-of-the-art method [19] to achieve fine-grained workload distribution when E is put across two devices. Specifically, the query execution can be divided into *steps* (one step is an operator in a query or more fine-grained processing in an operator). Because query executions are performed on some CPU CUs and the entire GPU in PE and cPDE-c, each step can be scheduled onto two devices with different amounts of workload to achieve balanced and optimal performance on the two sides.

In this paper, we use two common and fundamental database operations as examples for illustration purposes (i.e., selection and hash join). Without indexes, the selection is implemented using the filter primitive with the predicate as the filter function [15]. Hash join is a quite complex operation. Even after various memory optimizations [19, 3], memory stalls can seriously hurt the join performance. We adopted

the state-of-the-art hash join [19]. Each phase of the hash join (partition, build, probe) is divided into a number of steps. In the following, we present more implementation details of each component.

3.2.1 Prefetching

Previous studies [8, 39] have demonstrated the effectiveness of memory prefetching in hiding the memory stall with useful computation in databases. In the context of in-cache query co-processing on the coupled architecture, we revisit the impact of prefetching with special consideration on our four execution configurations.

Our prefetching structure is based on prefetching technique proposed by Zhou et al. [39] and is adapted to massively parallel architecture like GPUs. The *work-ahead set* (WAS) structure is used to temporarily store the prefetched data. There are many threads working concurrently on the GPU and they can issue many memory accesses at the same time. Thus, we insert memory accesses into WAS in a batch manner. When decompression is necessary, we use two work-ahead sets to form an execution pipeline of two data producer-consumer pairs: one is used to prefetch the data from the compressed input for decompression, and the other one is to store the decompressed data as input to the query executions. Another important issue is that a proper size of WAS may contribute to the overall performance. If the size is too small, the helper thread may not have enough time to load the requested cache lines. If the size is too large, the helper thread may evict out those data that may be still useful. Ideally, it should be smaller than the cache capacity. Since the main thread may have other cache-resident data that are still being used, the threshold value should be further lower in order to avoid conflict misses. The other two functional modules (D and E) access the data only if they have been prefetched. Thus, they are less unlikely to suffer from cache misses. This mechanism depends on many factors such as the compression ratio and the assigned CUs. We present an analytical model to address this problem in Section 4.

We adapt prefetching techniques in a more fine-grained way. Operators have to be divided into *steps* to enable prefetching. As selection has only one step as the definition, the step can be defined along the dimension of data. Specifically, we assume the number of work items working on selection is $NDRange$. The operations on the data within the range from 0 to $NDRange$-1 is considered as the first step. For the work item i, the data to be fetched next is at the position $(i+NDRange)$. The definition of steps follows our previous study [19]. The next memory position to be used can be obtained from the current step.

We fix prefetching on one CPU CU in practice for two reasons. Firstly, a wavefront of the GPU adopts the SIMD execution pattern. Thus, if any work item is blocked, all other work items within that wavefront need to wait for the blocked one, making GPUs inefficient in prefetching that involves many cache misses. Secondly, since one GPU CU is a multi-processor, using one entire GPU CU on prefetching is wasteful.

3.2.2 Data Compression

Database compression is an effective approach to improving query co-processing on the GPU [12]. It can increase

the bandwidth utilization and resolve the memory stalls of the APU.

We select typical compression algorithms introduced in previous work [12], including NS, NSV, DICT, RLE, Delta, Scale, and FOR. NS, NSV, DICT and RLE are the main compression schemes that can be used independently, whereas others are auxiliary in the sense that they can only be used with the main schemes to further improve the compression ratio. We briefly describe those compression schemes. More implementation details can be found at the previous study [12]. NS and NSV delete the leading zeros at the most significant bits in the bit representation of each element. RLE represents values in each *run* by a pair (value, run length) stored in two arrays, each of which can be further compressed. For auxiliary schemes, Delta encodes each value by the difference from the value at the preceding position. The first value is stored in the catalog for decompression. Scale converts floating point values into integers in cases where the integer format is precise enough for the application. FOR encodes each value in a column to an offset from the base value. The base value is usually selected as the smallest value of that column.

We adapt the compression planner used in [12] to obtain the optimal compression plan candidates. The major issues are to integrate the APU performance profile into the planner, and to incorporate software prefetching and in-cache query co-processing into the cost estimation. The cost analysis is captured by our cost model.

Though decompression can be avoided in cases with single main compression algorithms (such as NS and NSV), the *cascaded* compression plan generated by the compression planner [12] often necessitates decompression (or at least partial decompression). To have a thorough insight on how the performance of query co-processing can be impacted by decompression, we investigate both cases where decompression is needed or not. The detailed results are introduced in Section 5.

When decompression is necessary, the decompression fetches one compressed data block and decompresses it into the required format. Combined with prefetching, two WAS buffers are used. Each work item working on decompression inserts the next data position into the first WAS buffer for prefetching. The output of decompression is stored in an intermediate buffer that serves as the input for query execution. To coordinate the progress of D and E, a shared flag is set to indicate that decompression on specific compressed data block has been finished, and E can move onto processing those decompressed data.

3.2.3 Device Fission

As shown in Figure 4, two functional modules may be executed on the same device. Each functional module consists of one or more OpenCL kernels. Device fission is required to divide a single device into multiple subdevices. Each subdevice can execute a kernel from some functional modules. Thus, the same device can be shared among functional modules. Currently, device fission is fully supported on CPUs, and is not supported on current GPUs.

We adopt a simple yet effective software-based approach to achieve device fission on the GPU. To support two OpenCL kernels running concurrently on the GPU, we have to "merge" them into one kernel and then launch it on the GPU. The original two kernels are differentiated by $if\text{-}else$ conditional

statement. Algorithm 1 depicts how to merge two kernels K1 and K2 into a single kernel K. Suppose K invokes $NDRangeSize$ work items in total. A tuning parameter $NDRangeSize1$ is used to adjust the device fission on the GPU, so that $NDRangeSize1$ work items are launched for K1, and $(NDRangeSize - NDRangeSize1)$ work items are launched for K2. All indices in K2 need to be updated according to the dimension information. We ensure that $NDRangeSize1$ is an integral multiple of the work group size (i.e., the number of work items within that work group). Thus, no additional branch divergence is imposed on the merged kernel. This takes advantage of the OpenCL feature that the workload scheduling unit is one work group in the OpenCL runtime.

Algorithm 1 Software-based device fission between two OpenCL kernels K1 and K2 on the GPU.

$K(NDRangeSize, NDRangeSize1)$
{
/* $index$ represents work item ID in K */
if $index < NDRangeSize1$; **then**
    $Execute\ K1$;
else
    if $index < NDRangeSize$ **then**
        /* Update the index to make it start from 0 for K2*/
        $index \leftarrow index - NDRangeSize1$;
        $Execute\ K2$;
}

---

4. COST MODEL

Choosing the optimal configuration for various tuning parameters is an important task, especially in OpenCL that targets heterogeneous computing devices. In this section, we develop a cost model to estimate the execution time of query co-processing of the four execution configurations on the coupled architecture, and then use the cost model to determine the suitable values for the tuning parameters to achieve the lowest estimated execution time.

Though there has been plenty of existing work on building a cost model for applications either on the CPU or on the GPU, the architectural evolution of the APU brings new challenges. Firstly, the co-processing paradigm requires that our model should consider different characteristics of two processors within heterogeneous architectures. Secondly, functional modules run concurrently, and D and E can be deployed onto two devices simultaneously, which makes it more difficult to accurately predict the performance. Thirdly, prefetching can change the number of cache misses, and decompression can change the size of data accessed by each functional module. All these factors need to be considered in the estimation.

Because OpenCL has provided an abstraction for all OpenCL-compatible devices, we treat the CPU or the GPU as a processor with identical architecture, differentiated by computational capability and memory bandwidth. We profile each task on the CPU and the GPU independently and derive the cost on a single CU. Besides, we divide the total cost of a database operation into two major components: computation cost and memory cost. The computation cost is derived based on the theoretical peak Instruction Per Cycle (IPC) and the number of instructions. The memory cost considers prefetching and decompression. In the remainder of this section, we first present the abstract model, and next

use two operators (selection and hash join) as examples of instantiating the abstract model.

4.1 The Abstract Model

We have presented four paradigms to achieve co-processing, PE, cPDE-c, cPDE-b, and cPDE-g as illustrated in Figure 4. We focus on how to build the cost model for cases where prefetching and decompression are integrated.

We have the following four key designs to make the model accurately find the configuration. Firstly, as the query executions (E) is optimized in the fine-grained method, operators need to be staged into *steps*. It is difficult to determine the optimal ratios among all steps to get the optimal total time. Therefore, we have adopted the cost model by He et al. [19] to address this problem. Secondly, the cache effect must be included. Ideally, all working data of D and E can be accessed directly from the cache, and only P suffers from cache misses, which can be hidden by the computation of D and E. If cache misses appear in D or E, the penalty needs to be considered in the cost model. Thirdly, the device fission divides one device into multiple subdevices with different tasks running on them. In that scenario, the cost model can estimate the execution time accurately when different DE boundaries are applied. Fourthly, functional modules work in a pipelined manner, which can cause delay (i.e., inappropriate CU assignment to P/D/E). For execution, the delay occurs when the output of D cannot satisfy the input for E in time. In cases where data is not correctly preloaded into the L2 cache, both D and E are delayed.

Table 2 lists the notations in our cost model.

Table 2: Notations in the cost model.

Notation	Description
XPU	CPU or GPU
N	The input size (of D) in bytes
$|C|$	The number of CPU CUs
$|G|$	The number of GPU CUs
B_{XPU}	The peak bandwidth (GB/s) of XPU to the shared main memory area
S	The preset data size to be prefetched
F	$F \in \{P,D,E\}$
T_F	The actual execution time of F
$comp_F^{XPU}$	The computation time of F on XPU
M_F^{XPU}	Memory access time of F
C_F, G_F	The number of CPU and GPU CUs assigned to F
$\#I_{XPU}^F$	The number of instructions of F on XPU
cr	Compression ratio
L_M^{XPU}	The access latency between XPU and main memory
L_C^{XPU}	The access latency between XPU and L2 cache
R_F	The output throughput of F in bytes
IPC_{XPU}	The theoretical peak instruction per cycle on XPU

Prefetching is performed on a CPU CU. The major workload of that CPU CU is on memory fetch instructions. We denote the data size to be prefetched in getting prefetching throughput as S. To guarantee the real execution of prefetching, light-weight computation is involved. Thus, the throughput of prefetching can be calculated by the bandwidth and the computational capability of the CPU CU(s).

$$R_P = \frac{|S|}{\frac{|S|}{B_{CPU}} + \frac{I_{CPU}^P}{IPC_{CPU}}} \quad (1)$$

The execution time of D can be calculated in two components: the computation time and the memory access time.

For computation time, we count the number of instructions running on device(s) with OpenCL profiler such as CodeXL or AMD APP Profiler, and calculate the total computation time of instructions according to the theoretical peak instructions per cycle (IPC) of the processor for each decompression algorithm. To achieve optimal compression ratio on each column, different plans are applied. Hence, the number of instructions for decompression is not constant among columns. We choose the compression plan based on the model proposed by Fang et al. [12]. We perform benchmarking before the query execution to obtain the instruction number of decompression of each column on any processor, namely, I_{XPU}^D. According to the numbers of CPU CUs and GPU CUs assigned to decompression, we can obtain the amount of instruction execution time on each device. The total execution time of decompression depends on the longer execution time of two devices, as shown in Eq. 2.

$$comp_D^{XPU} = max(comp_D^{CPU}, comp_D^{GPU})$$
$$= max(\frac{\#I_{CPU}^D \times \frac{C_D}{|C|}}{IPC_{CPU}}, \frac{\#I_{GPU}^D \times \frac{G_D}{|G|}}{IPC_{GPU}}) \quad (2)$$

The memory access time consists of time of accesses from the main memory and L2 data cache. If $\frac{N}{comp_D^{XPU}} > R_P$, an amount of $(N - comp_D^{XPU} \times R_P)$ data in bytes has to be accessed from main memory. Thus, we have the following memory access time.

$$M_D^{XPU} = (N - comp_D^{XPU} \times R_P) \times L_M^{XPU}$$
$$+ (comp_D^{XPU} \times R_P) \times L_C^{XPU} \quad (3)$$

Otherwise, the input data have been perfectly prefetched into cache before use, and we have Eq. 4.

$$M_D^{XPU} = N \times L_C^{XPU}, \quad (4)$$

We can derive the actual execution time of D to be,

$$T_D = comp_D^{XPU} + M_D^{XPU} \quad (5)$$

Similarly, we perform the estimation for E: the computation time ($comp_E^{XPU}$) as Eq. 2 and memory access time (M_E^{XPU}) as Eq. 3 ~ Eq. 4. We define the sum of $comp_E^{XPU}$ and M_E^{XPU} as the assumptive execution time of E (T_E') based on the assumption that the output of D can satisfy the input of E in time. However, as D and E form a producer-consumer chain in the real execution, E may have to wait for D if inappropriate CU assignment is adopted. The delay is the difference between T_D and T_E'. If $T_D \leq T_E'$, D can feed E in time. Thus, we have the following estimation.

$$T_E = T_E' \quad (6)$$

Otherwise, the processing time of E is limited by D. Thus, we have the following estimation.

$$T_E = T_D \quad (7)$$

The number of CUs assigned to all functional modules should be within the limit of available CUs. Therefore, the following conditions should also be satisfied.

$$0 \leq C_P + C_D + C_E \leq |C| \quad (8)$$

$$0 \leq G_D + G_E \leq |G| \quad (9)$$

The goal of our cost model is to find the optimal plan that can minimize the total execution time as Eq. 10. Because

the number of CUs in the CPU and the GPU are relatively small and P is fixed on one CPU CU, we simply iterate all the possible combinations to find the optimal plan.

$$Minimize(max(T_D, T_E)) \qquad (10)$$

4.2 Model Instantiation

We use hash joins and selections to illustrate how we use the cost model. Models for other operators can be developed in the same way.

Hash join consists of three stages: partition, build, and probe. Each stage can be divided into several steps. To demonstrate how a hash join can be mapped to the cost model, we use the build stage for illustration. The build stage can be divided into four steps (b_1, b_2, b_3, b_4) as defined in the previous paper [19]. Before step b_1 starts, each work item i needs to write the memory address that will be accessed immediately in step b_1 into the WAS buffer that exclusively serves E (assume prefetching distance is 1). In the next move, if work item i has finished operations in the current step, the next memory address is calculated and written into the WAS buffer before it proceeds. Ideally, the operations in the current step can hide the memory latency before the next data can be prefetched into cache.

Assume the number of CPU CUs and GPU CUs assigned to hash join is C_E and G_E, respectively. The computation time can be derived in the same way as Eq. 2. We compare this computation time with that of prefetching to obtain the memory access time including the cache misses impacts as Eq. 3. One issue to be handled is that the size of memory consumed by hash join is $\frac{1}{cr}$ times as large as that of D, if the data is decompressed into the original format. In this way, we obtain the memory access time and computation time of hash join with specified preset configuration. In decompression, steps are defined in a different way from hash join. As introduced in Subsection 3.2.2, operations on each small compressed data block are defined as one step. With prefetching distance of 1, the same position in the next data block is written into the WAS buffer for prefetching. Next, we compare the estimated time of D with E to obtain the real execution time according to Eq. 6 and Eq. 7 as they form a producer-consumer chain.

In this way, we can obtain the estimated execution time of hash join with one specified configuration plan. To obtain the optimal configuration plan, we search the whole configuration space within the resource constraints (e.g., $|C| = 4$ and $|G| = 5$ on A8) to get the minimal estimated execution time (Eq. 10). Thus, we can get the minimal total cost of hash join as well as the optimal configuration plan.

The estimation on selection is similar to that of hash join, except that the step defined in the selection is a batch of operations that all work items perform scan over input tuples one by one.

5. EXPERIMENTAL EVALUATION

The evaluations are categorized in two groups, whether query processing is without decompression (Section 5.2) or with decompression (from Section 5.3 to 5.4).

5.1 Experimental Setup

Hardware configuration. Our experiments are conducted on two workstations. One is equipped with AMD A8-3870K (A8) and 8 GB DRAM. The other is equipped with A10-7850K (A10) and 32GB DRAM. The configurations of A8 and A10 have been presented in Table 1. We follow the experimental methodology in the previous study [19] to study the comparison with discrete CPU-GPU architectures. We observe consistent results as the previous study [19]. In general, removing the PCI-e data transfer overhead, the APU architecture outperforms the discrete architecture with the same configuration. Since the focus of this study is to further improve the performance of query co-processing on coupled CPU-GPU architectures, we have omitted the detailed results on discrete architectures.

Data sets. We select four queries (i.e., Q3, Q6, Q9, and Q14) from TPC-H with different complexities. Q6 is a relatively simple selection query, whereas Q9 involves a rather complicated subquery. Q3 has two join operations and Q14 has a single join operation. Those queries also have other common operators such as group-by and order-by. TPC-H query experiments are conducted based on the data generated from the TPC-H data generator. The maximum scale factors are set as 5 and 10 on A8 and A10 machines, generating approximately 5GB and 10 GB databases, respectively.

We use column stores for query processing as OmniDB [38]. Furthermore, to study in-cache query co-processing in more details, we evaluate our executions for two core operators in databases (i.e., selection and hash join). In the evaluations of individual operators, the original data set contains 16M pairs <key, record-id> in each relation. Both keys and payloads are 8 bytes long as in existing work [4, 24]. To investigate the situation when the query process directly on the compressed data, we *intentionally* study the performance of individual operators on the data with specific compression algorithm, which is not optimal according to our evaluation.

We choose the CPU-only counterpart of each execution configuration as the baseline and demonstrate the performance impact of our proposal by showing the *normalized speedup* ($\frac{t}{t'}$, where t and t' are the execution times of the CPU-only execution and the studied execution configuration, respectively). Specifically, without decompression, the baseline implementation processes queries only on the CPU (i.e., E-CPU). While decompression is necessary, the baseline implementation is to perform the D/E functional modules on the CPU only (namely cDE-CPU). Additionally, we have two variants by adopting our cost model to the CPU only: cPDE-CPU and PE-CPU for the CPU-only executions with and without decompression, respectively.

5.2 Operators and Queries Evaluation without Decompression

We first investigate the performance of schemes without decompression (i.e., E and PE) on uncompressed and compressed data, respectively. Note, E and PE involve both the CPU and the GPU. We vary the WAS size from small (128KB) to large (16MB). Figure 5 shows the results for operators on uncompressed data. We find that prefetching performance depends on the WAS size, and 1MB is chosen as the optimal setting. The performance improvement of PE over the latest co-processing method E [19] is 24% and 22% on the selection and the hash join, respectively.

To study prefetching for query processing on compressed data without decompression, we apply single compression algorithm (i.e., NS) so that the operators can process the data without decompression, as shown in Figure 6. The compres-

(a) Selection   (b) Hash Join
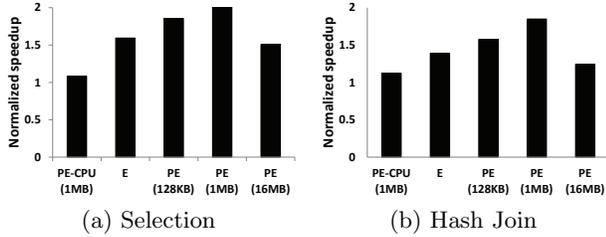
Figure 5: The normalized speedup to E-CPU of prefetching on selection and hash join on uncompressed data on A10.

(a) Selection   (b) Hash Join
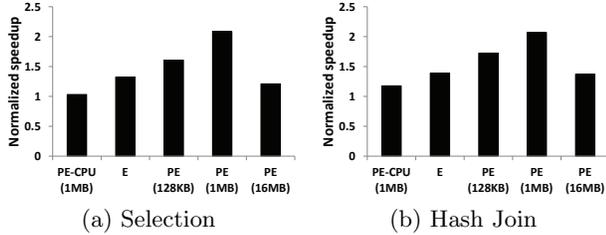
Figure 6: The normalized speedup to E-CPU of prefetching on selection and hash join on compressed data without decompression on A10.

sion ratio (defined as the data size after compression to the data size without compression) is 25% for the input relations. As expected, the execution time of each operator is reduced compared to the one on uncompressed data. We observed similar performance improvement by prefetching on direct query executions without decompression. We also studied other compression schemes. Generally, PE outperforms E with similar performance improvement.

Next, we evaluate the TPC-H query performance on the uncompressed data. Figure 7 depicts the performance comparisons on A8 and A10. Compared with the CPU-only approach, both E and PE achieve over 1.5 times speedup. Prefetching can reduce the query processing time of all queries by up to 19% on A8, and up to 20% on A10, respectively. The significant improvement in the query processing performance shows the effectiveness of in-cache query processing in reducing the effects of memory latency.

5.3 Operators with Decompression

Impact of prefetching: When the data is compressed with cascaded compression, D may be required for query executions. The results on A10 are presented in Figure 8.

(a) A8   (b) A10
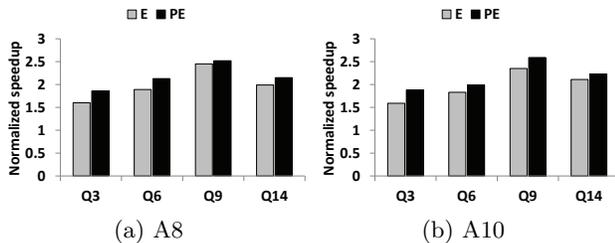
Figure 7: The normalized speedup to E-CPU of prefetching on TPC-H queries on uncompressed data.

(a) Selection   (b) Hash Join
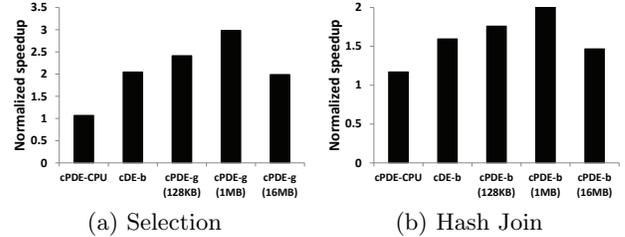
Figure 8: The normalized speedup to cDE-CPU of prefetching on selection and hash join on compressed data with decompression on A10.

Table 3: Compression results on A10 for different compression plans on Selection.

Compression plan	Compression ratio	Decompression	Time (ms)
(A):RLE	3.34%	No	11.3
(B):NS	100%	No	104.7
(C):RLE,[ε | NS]	2.76%	No	10.1
(D):RLE,[[Delta, NS] | NS]	2.55%	Partial	37.7
(E):Delta, NS	25%	Full	145.4

The speedup of the scheme with prefetching achieves 31% and 25% over the original scheme cDE-b. By measuring the L2 cache misses distribution, we find that the cache misses are now limited to the CPU CU working on prefetching. Another observation is that when the WAS size is not appropriately set, prefetching performance degrades dramatically. When the size is too large, the performance is even worse than the original scheme without prefetching because of cache pollution caused by excessive prefetching.

Impact of compression: To further reduce the data footprint, cascaded compression is applied as in the previous study [12]. Table 3 lists the results of the selection operator with five compression plans. We use the symbols adopted in previous work [12]. Plans (A) and (B) contain single compression algorithm and the compressed data can be directly processed by the selection operator. Plans (C), (D) and (E) apply two consecutive compressions onto the input data in order to achieve better compression ratio.

The processing time without compression on the GPU is 101 ms. As the table shows, when the data compression ratio is better and no decompression is needed, then the selection time can be reduced by 90%. In Plan (C), when cascaded compression is applied, the compression ratio can be improved. However, the processing time is close to the one with Plan (A), which indicates that better compression ratio does not guarantee better query processing. In Plan (D), the partial decompression policy can significantly reduce the overhead introduced by decompression compared to full decompression in Plan (E). In Plan (E), the compression ratio is improved from 100% to 25% compared to Plan (B), but the processing time increases from 11.3 ms to 145.4 ms. This is mainly because the decompression overhead offsets the benefit gained from the smaller data size to be transferred. We add the feature of APU into the cost model [12]. Our evaluations find that the cost model is able to find the optimal plan with the best performance accurately.

Impact of device fission: We statically assign the CPU and the GPU CUs to three functional modules and study the

(a) Selection

(b) Hash Join
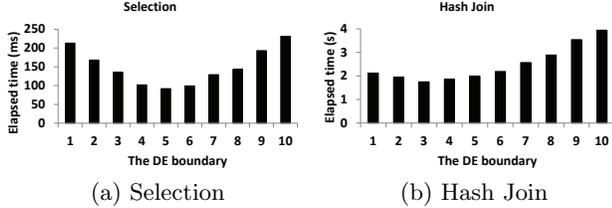
Figure 9: The results of manually configured settings on different operations on A10.

(a) Q9 on A8

(b) Q14 on A8

(c) Q9 on A10

(d) Q14 on A10

Figure 10: Model validation on the elapsed time on A8 and A10 platforms.

performance trend. Specifically, prefetching is fixed on the first CPU CU, and the DE boundary is varied from 1 to 7 on A8 and from 1 to 10 on A10. As the result on A8 is similar to that on A10, we only show the result on A10 in Figure 9.

Figure 9a shows that the performance of selection can vary significantly as the DE boundary increases. This is because decompression is the most time-consuming component for the selection operation. The fewer CPU CUs are assigned to decompression, the more time the query execution has to wait for the output from the decompression. In contrast, Figure 9b shows that the hash join is relatively less sensitive to the DE boundary. Compared with decompression, hash join is a more time-consuming operation.

5.4 Queries with Decompression

Cost model validation: To validate the effectiveness of our cost model, we compare the elapsed time and CU assignment plans from measurements with those from the cost model.

We use the notation for a CU assignment plan, (x_1Cy_1G, x_2Cy_2G), to denote that x_1 CPU CUs and y_1 GPU CUs are assigned to D, and x_2 CPU CUs and y_2 GPU CUs are assigned to E. Our model correctly predicts the optimal plan. For example, the predicted optimal plan for Q9 and Q14 are $(2C0G, 1C5G)$ and $(3C1G, 0C4G)$ respectively on A8, and are $(2C0G, 1C8G)$ and $(3C2G, 0C6G)$ respectively on A10. Thus, it matches the measured optimal configuration.

Figure 10 shows the measured and the estimated execution time for TPC-H queries on A8 and A10 as we vary the DE boundary. Overall, our estimation approximates the measurement well for TPC-H queries. It is able to produce the optimal assignment plan and predict the trend of the relative performance with the differences less than 9%.

Evaluation of TPC-H queries: In the previous work [12], a compression planner is adopted to choose the optimal compression plan delivering the best performance. For completeness, we revisit the effectiveness of the planner for coupled CPU-GPU architectures.

To obtain a comprehensive understanding on how our co-processing paradigm can benefit from two generations of APUs, we demonstrate the performance of query processing on both A8 and A10 for Q9 and Q14 in Figures 11 to 12. The results for Q3 and Q6 on A10 is presented in Figure 13. On A8, similar results are observed on Q3 and Q6 compared with those on Q9 and Q14. From the results, we can make the following three observations.

Firstly, by exploiting the power of both processors, cPDE-b can outperform the baseline by more than 2.5 times. Query co-processing with both the CPU and the GPU significantly outperforms the CPU-only approach. Moreover, our in-

(a) Q9 results on A8

(b) Q14 results on A8
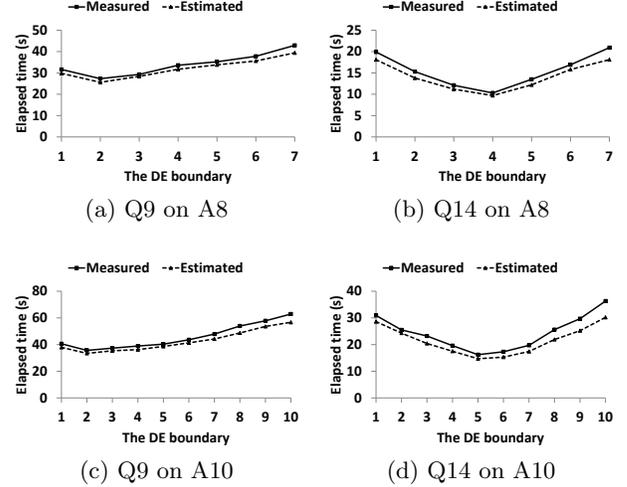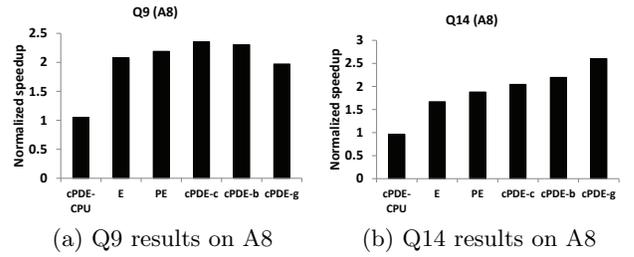
Figure 11: The normalized speedup to cDE-CPU of different schemes on TPC-H data set on A8.

(a) Q9 results on A10

(b) Q14 results on A10
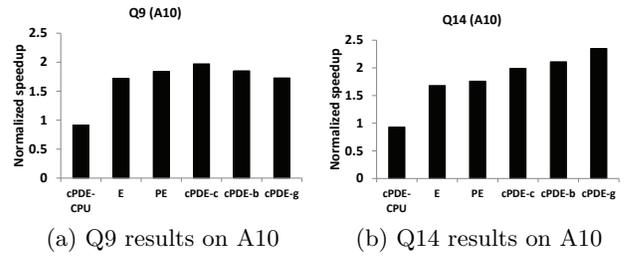
Figure 12: The normalized speedup to cDE-CPU of different schemes on TPC-H data set on A10.

(a) Q3 results on A10

(b) Q6 results on A10
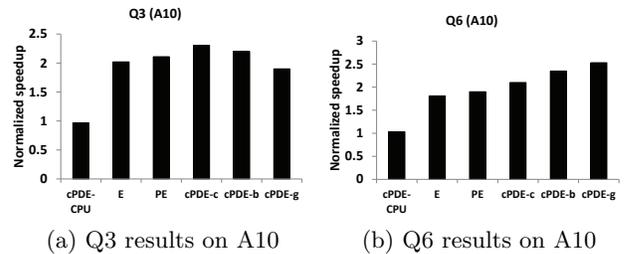
Figure 13: The normalized speedup to cDE-CPU of different schemes on TPC-H data set on A10.

cache design has significantly reduced the memory stalls of the GPU, and dynamically schedules the P/D/E functional modules to their suitable numbers of CUs on both A8 and 10. In contrast, the performance of the CPU-only approach of running P/D/E (cPDE-CPU) is similar to that of the baseline, because all the CUs of the CPU-only approach are homogeneous, and heterogeneous workload scheduling has little impact. It might even slightly slow down the performance due to runtime overhead of the advanced scheduling.

Secondly, our CPU-GPU co-processing paradigm is much faster than the state-of-the-art approach on the APU [19]. For query processing on A8, the optimal scheme can outperform the fine-grained approach (i.e., E) by up to 36%. On A10, the improvement can even achieve at 40%. Though the fine-grained method has captured the workload preference of different processors in the previous study [19], excessive memory stalls make the GPU underutilized in the coupled CPU-GPU architecture. In contrast, our heterogeneous workload scheduling is able to exploit the advantages of both the CPU and the GPU.

Thirdly, for schemes with prefetching and decompression, the DE boundary can significantly affect the optimal performance. On A8 and A10, the best scheme can outperform the worst one by 21% and 17%. With suboptimal configurations, the workload assigned to two devices is unbalanced. Furthermore, if the producer-consumer chain is stalled due to inappropriate configuration, more memory stalls are incurred which can further deteriorate the device efficiency (especially for the GPU).

We further study the profiling results on the memory stalls incurred in query processing. We only present the results of Q9 and Q14, since we observe the same behavior on Q3 and Q6. Figure 14 and 15 demonstrate the detailed time breakdown of memory units on both the CPU and the GPU for Q9 and Q14, respectively. For simplicity, we show the results of E, PE, and the optimal one of the remaining three schemes (denoted as *opt*). With prefetching enabled, the CPU CU assigned with prefetching suffers much more L2 cache misses, resulting in high percentage of stalled memory instructions. As Figure 14a shows, when P and D are not used, the average percentage of stalled memory instructions of each CPU CU is around 25% (C0 to C3). It reaches nearly 38% on the GPU that can seriously degrade the GPU efficiency. With prefetching enabled, the memory stalls suffered by the CPU CU working on prefetching (i.e. C0) are 73% of the memory unit cycles. However, it drops to around 10% on other CPU CUs and the GPU. This is because most cache misses have been shifted to prefetching CU. The benefit is even more distinct in the opt scheme as the smaller footprint and higher utilization of bandwidth can contribute more to the reduction of memory stalls. That can release the powerful computing strengths of the GPUs and enhance the overall performance.

To study the impact of data sizes, we increase scale factor (SF) from 1 to 10 on A10. The experiments are conducted in the optimal scheme produced from the cost model (denoted as *opt*). For comparison, results with fine-grained approach [19] are also presented in Figure 16 (denoted as *old*). Both *opt* and *old* schemes have good scalability with the increasing scale factor.

5.5 Insights and Implications

Through the evaluations on operators and queries, we

(a) Q9   (b) Q14
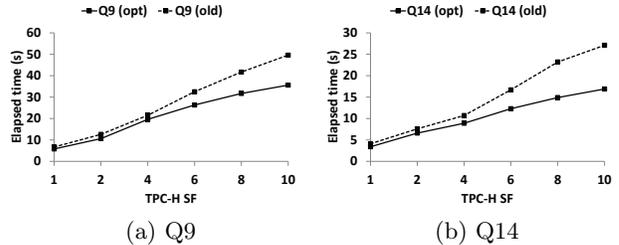
Figure 16: TPC-H query performance on A10 with variable scale factors.

have demonstrated that in-cache query co-processing can further improve the state-of-the-art query co-processing on coupled CPU-GPU architectures. Specifically, it can significantly reduce the memory stalls so that the efficiency of both the CPU and the GPU can be highly improved. From the experimental results, we can obtain some implications that can guide the future architectural design and database management systems.

Firstly, as Figures 14 and 15 show, though prefetching has significantly mitigated the cache misses suffered by the GPU, higher bandwidth and larger cache size on future architectures can further increase the efficiency of our query co-processing paradigm. That also means, memory optimizations continue to be a key performance issue for databases in future architectures.

Secondly, we argue that GPUs in coupled architectures are more sensitive to cache misses than those in discrete systems. Looking forward, more efficient cache design with multi-level architecture can be potentially beneficial to the GPU performance. Besides, replacing DDR3 with GDDR5 as main memory [33] can be an interesting approach to increasing database performance. Also, a hardware prefetching engine for resolving the memory stalls on the GPU can be very helpful.

Thirdly, modern databases require not only high computational capability, but also the capability to handle heterogeneous workload. This requirement necessitates a more sophisticated software system design to integrate various processors. "One size does not fit all" still holds. The evolving coupled CPU-GPU architectures will significantly impact the database research, which may potentially impact the architectural design of next-generation database systems.

6. RELATED WORK

We review the related work in the following categories:

Cache-optimized query processing on CPUs: Cache optimizations to improve query processing performance have been widely studied in database community. There are two major categories along this study: cache-conscious [32] and cache-oblivious [16]. Cache-conscious techniques utilize cache parameters (such as cache capacities and cache line sizes) to reduce the memory access latency to improve the database performance. In contrast, cache-oblivious techniques optimize cache performance without taking cache parameters as input. There has been much more existing work on cache-conscious optimizations. Manegold et al. [5] proposed a cost model that can estimate the execution time of query processing. However, their model does not cover prefetching or GPU co-processing. Recently, Pirk et al. [27]
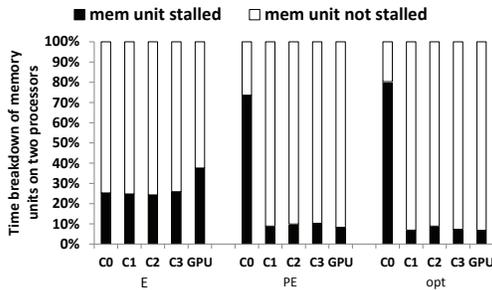
(a) Stall distribution of Q9 on A8   (b) Stall distribution of Q14 on A8
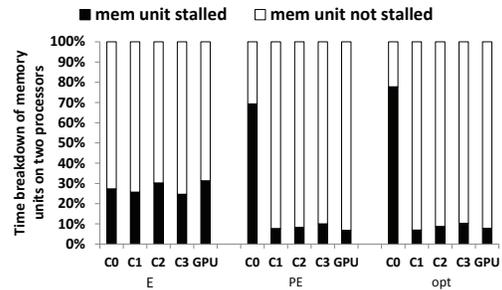
Figure 14: Memory stall distribution of different schemes between each CPU CU and the GPU on A8.
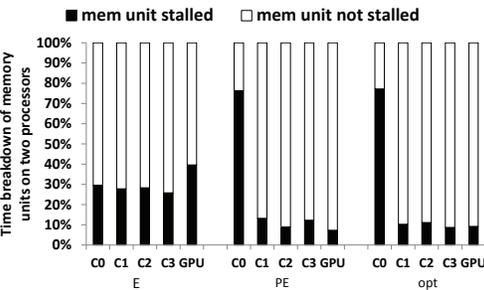
(a) Stall distribution of Q9 on A10   (b) Stall distribution of Q14 on A10
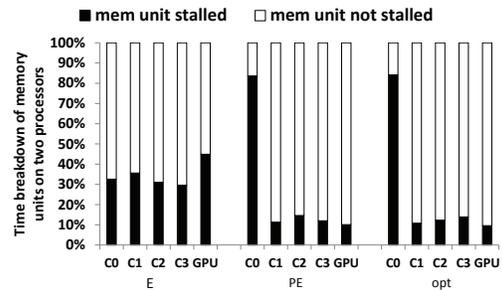
Figure 15: Memory stall distribution of different schemes between each CPU CU and the GPU on A10.

studied partial decomposition to save bandwidth without sacrificing CPU cycles. Ross et al. [10] explored the architectural features that can affect the overall performance of aggregations and hash joins. Balkesen et al. [3] advocated that additional performance can be obtained through carefully tailoring algorithms to more efficiently utilize architectural parameters such as cache sizes, TLB, and memory bandwidth. Another effective means to reduce the cache misses is database compression (e.g., [9, 1]).

In comparison, our in-cache processing design exploits the advantages of the shared cache design of the coupled CPU-GPU architecture.

Query co-processing on discrete GPUs: Because discrete GPUs have much higher bandwidth and massive thread parallelism from CPUs, they are ideal choice for query processing. For a long time, the query co-processing was achieved on the discrete CPU-GPU architecture [15, 17, 22, 6, 37, 28, 18]. Wu et al. [35] introduced a compiler and infrastructure named Red Fox for OLAP queries. Zhang et al. [38] and Heimel et al. [20] presented their findings in designing a portable query co-processing engine across different devices by using OpenCL. As compression can benefit the cache and save the bandwidth on the CPU, Fang et al. [12] explored cascaded compression on discrete GPUs.

Though workload distributions on co-processing paradigms targeting the discrete CPU-GPU architecture are also heterogeneous, they are not designed to exploit the architectural features of coupled CPU-GPU designs to achieve fine-grained workload scheduling. Our work focuses on both devices but schedules the hardware-favored workload to each of them in a fine-grained and heterogeneous manner.

Studies on coupled CPU-GPU architectures: There have also been some studies (like MapReduce [7], key-value stores [21] and hash joins [19]) on this architecture. Most studies have demonstrated the performance advantage of the coupled architecture over the CPU-only or the GPU-only algorithm. Yang et al. [36] showed the effectiveness that the CPU can assist the GPU through prefetching to highly improve the performance of the GPU. Our study focuses on database operations, and goes beyond the existing studies in two major aspects. Firstly, different from the homogeneous workload distribution among processors, our framework processes a single query in databases including many operators with varying and heterogeneous runtime features. Secondly, we generalize the roles of various device resources for query processing and highlight the efficiency of in-cache design for GPU query co-processing.

7. CONCLUSIONS

In this paper, we have proposed an in-cache query co-processing paradigm on coupled CPU-GPU architectures. We have adapted CPU-assisted prefetching to minimize the cache misses of GPU query co-processing, and CPU-assisted decompression schemes to improve query execution performance. Additionally, we have proposed a cost model to predict the execution time and choose the optimal core assignment plan. As the experimental results show, the in-cache co-processing paradigm can effectively reduce the impact of memory stalls, thus improving the overall performance of TPC-H queries by up to 36% and 40% over the state-of-the-art fine-grained method on AMD A8 and A10, respectively. Such improvements show that in-cache query co-processing is promising on coupled CPU-GPU architectures. As for future work, we are interested in extending our system to row stores (e.g., by revisiting prefetching and data compres-

sion in the context of row stores) and in exploring the issues discussed in Section 5.5.

8. ACKNOWLEDGEMENT

9. REFERENCES

[1] D. Abadi and et al. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] ARM. big.little processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[3] C. Balkesen and et al. Main-memory hash joins on multi-core CPUs: tunning to the underlying hardware. In *ICDE*, 2013.

[4] S. Blanas and et al. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, 2011.

[5] P. A. Boncz and et al. Database architecture optimized for the new bottleneck: memory access. In *VLDB*, 1999.

[6] S. Breßand G. Saake. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *PVLDB*, 2013.

[7] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a coupled CPU-GPU architecture. In *SC*, 2012.

[8] S. Chen and et al. Improving hash join performance through prefetching. *TODS*, 2007.

[9] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, 2001.

[10] J. Cieslewicz, W. Mee, and K. A. Ross. Cache-conscious buffering for database operators with state. In *DaMoN*, 2009.

[11] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *ICPP*, 2011.

[12] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 2010.

[13] N. Govindaraju and et al. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[14] N. K. Govindaraju and D. Manocha. Efficient relational database management using graphics processors. In *DaMoN*, 2005.

[15] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM TODS*, 2009.

[16] B. He and Q. Luo. Cache-oblivious databases: limitations and opportunities. *ACM Trans. Database Syst.*, 2008.

[17] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[18] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *PVLDB*, 2011.

[19] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB*, 2013.

[20] M. Heimel and et al. Hardware-oblivious parallelism for in-memory column-stores. In *PVLDB*, 2013.

[21] T. H. Hetherington and et al. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *ISPASS*, 2012.

[22] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *DaMoN*, 2012.

[23] Khronos. The OpenCL specification. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[24] C. Kim and et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2009.

[25] C. Kim and et al. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, 2010.

[26] K. Lee, H. Lin, and W.-C. Feng. Performance characterization of data-intensive kernels on AMD fusion architectures. *Comput. Sci.*, 2013.

[27] H. Pirk and et al. CPU and cache efficient management of memory-resident databases. In *ICDE*, 2013.

[28] H. Pirk, S. Mnegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*, 2014.

[29] N. Satish and et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, 2010.

[30] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*, 2009.

[31] P. G. Selinger and et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[32] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, 1994.

[33] Sony Computer Entertainment, Inc. Playstation 4 specifications. http://us.playstation.com/ps4/features/techspecs/.

[34] K. Thouti and S.R.Sathe. Comparison of OpenMP and OpenCL parallel processing technologies. *IJACSA*, 2012.

[35] H. Wu and et al. Red fox: an execution environment for relational query processing on GPUs. In *CGO*, 2014.

[36] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. CPU-assisted GPGPU on fused CPU-GPU architectures. In *HPCA*, 2012.

[37] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 2013.

[38] S. Zhang, J. He, B. He, and M. Lu. OmniDB: towards portable and efficient query processing on parallel CPU/GPU architectures. In *VLDB (demo)*, 2013.

[39] J. Zhou and et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.