DIDO: Dynamic Pipelines for In-Memory Key-Value Stores on Coupled CPU-GPU Architectures

Kai Zhang*, Jiayu Hu†, Bingsheng He*, Bei Hua†
*School of Computing, National University of Singapore
†School of Computer Science and Technology, University of Science and Technology of China

***Abstract*—As an emerging hardware, the coupled CPU-GPU architecture integrates a CPU and a GPU into a single chip, where the two processors share the same memory space. This special property opens up new opportunities for building in-memory key-value store systems, as it eliminates the data transfer costs on PCI-e bus, and enables fine-grained cooperation between the CPU and the GPU. In this paper, we propose DIDO, an in-memory key-value store system with dynamic pipeline executions on the coupled CPU-GPU architecture, to address the limitations and drawbacks of state-of-the-art system designs. DIDO is capable of adapting to different workloads through dynamically adjusting the pipeline with fine-grained task assignment to the CPU and the GPU at runtime. By exploiting the hardware features of coupled CPU-GPU architectures, DIDO achieves this goal with a set of techniques, including *dynamic pipeline partitioning*, *flexible index operation assignment*, and *work stealing*. We develop a cost model guided adaption mechanism to determine the optimal pipeline configuration. Our experiments have shown the effectiveness of DIDO in significantly enhancing the system throughput for diverse workloads.**

I. Introduction

The coupled CPU-GPU architecture is an emerging hybrid architecture that integrates a CPU and a GPU in the same chip, e.g., AMD Kaveri APU architecture. In this architecture, the CPU and the GPU share the same physical memory and have a unified memory address space. The two processors become capable of accessing the same data set in the host memory simultaneously. This eliminates the expensive PCIe data transfer, and dramatically reduces the cost of CPU-GPU communication. These features make fine-grained CPU-GPU cooperation feasible, thus new opportunities are opened up for the design and implementation of data processing systems. Recently, hybrid CPU-GPU architectures have been used to improve the throughput and energy efficiency of in-memory key-value stores (IMKV), such as Mega-KV [1] and MemcachedGPU [2]. Because of the effective memory access latency hiding capability and the massive number of cores in GPUs, CPU-GPU co-processing becomes an effective way for building efficient IMKV systems.

The current IMKV systems on CPU-GPU platforms, e.g., Mega-KV and MemcachedGPU, are designed for architectures with discrete GPUs, where the CPU and the GPU are connected via a PCI-e bus. We find that these IMKV designs are inefficient on the coupled CPU-GPU architecture. Due to the costly PCI-e data transfer in the discrete CPU-GPU architecture, the CPU and the GPU in current IMKV systems form a fixed and static pipeline to co-process key-value queries. For instance, Mega-KV has three major stages in its pipeline. By evaluating these systems on the coupled CPU-GPU architecture, we find that existing static pipeline designs have two major inherent limitations, which makes them incapable of fully utilizing the coupled CPU-GPU architecture.

First, static pipeline designs can result in severe pipeline imbalance and resource underutilization. IMKVs store a variety of key-value objects, such as user-account status information and object metadata of applications [3]. Consequently, the workload characteristics of IMKVs tend to have very high variance. For instance, the GET ratio in Facebook ranges from 18% to 99%, and the value size ranges from one byte to thousands of bytes [3]. Such highly variable workloads may result in a significant difference in the execution time of each pipeline stage. As a result, an IMKV system with a fixed pipeline partitioning scheme can suffer severe pipeline imbalance, resulting in suboptimal performance for certain workloads. On the other hand, in an imbalanced pipeline, the processor that is in charge of the pipeline stage with a lighter workload becomes idle when waiting for other stages with heavier workloads. Both the compute and memory resources would be underutilized during the idle time.

Second, the static pipeline designs fail to efficiently allocate workloads to appropriate processors. As throughput-oriented processors, GPUs are good at processing a large batch of requests to improve resource utilization. In the static pipeline design, we find that a large portion of GPU's execution time is spent on processing a small amount of index update operations (i.e., *Insert* and *Delete*) with read-intensive workloads, which makes up almost half of the GPU execution time in Mega-KV.

To address those inefficiencies and limitations, we propose DIDO, an IMKV system with dynamic pipeline executions on coupled CPU-GPU architectures. Unlike the systems on discrete CPU-GPU architectures, both the CPU and the GPU in the coupled architecture are able to access all the data structures and key-value objects, and the cost of data communication between processors becomes extremely low. DIDO takes advantage of these features to dynamically adjust the pipeline partitioning at runtime. With a cost model guided adaption mechanism, the system delivers high CPU/GPU resource utilization and more balanced pipelines by dynamically adopting the optimal pipeline partitioning scheme for the current workload, flexibly assigning index operations to appropriate processors, and work stealing between the CPU and the GPU.

This paper makes the following main contributions: (1) By evaluating the existing GPU-based IMKV systems on coupled CPU-GPU architectures, we have identified the inefficiency of static pipeline designs in handling diverse workloads and in taking advantage of the coupled CPU-GPU architecture. (2) We have proposed DIDO, an in memory key-value store system with Dynamic pIpeline executions for Diverse wOrkloads. With a cost model guided adaption mechanism, DIDO is able to dynamically adapt to different workloads by balancing its pipeline at runtime. (3) Based on DIDO, we build an IMKV system on an AMD Kaveri APU for evaluation. Our experi-

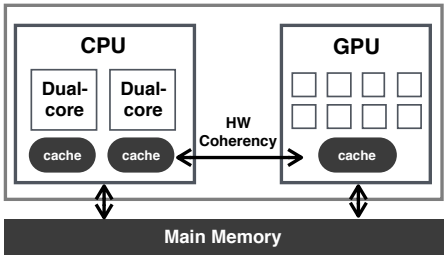

Fig. 1. A10-7850K APU Architecture

ments have shown the effectiveness of the dynamic pipeline execution in achieving significant performance improvement over the state-of-the-art system designs on the coupled CPU-GPU architecture.

The road map of this paper is as follows. Section II introduces the background and motivation of this research. Section III lists the major techniques of DIDO and shows its framework, and the cost model is introduced in Section IV. The system performance is evaluated in Section V. Section VI reviews related work, and Section VII concludes the paper.

II. Background and Motivations

A. Coupled CPU-GPU Architectures

Heterogeneous CPU-GPU architectures fit for accelerating applications with different computation needs. In *discrete* CPU-GPU architectures, data for GPU processing should be transferred to the GPU memory via PCIe bus, which is considered as one of the largest overhead for GPU execution [4]. The *coupled* CPU-GPU architecture integrates a CPU and a GPU into the same chip, where the two processors are able to share the same physical memory. As an example, Figure 1 sketches the architecture of the AMD A10-7850K Kaveri APU. The APU integrates four CPU cores and eight GPU compute units on a chip. As a significant advancement over previous products, Kaveri APUs first support heterogeneous Uniform Memory Access (hUMA). hUMA provides three new features, including unified memory address space, GPU-supported paged virtual memory and cache coherency. Cache coherency guarantees that the CPU and the GPU can always have an up-to-date view of data [5]. Therefore, the two processors are now able to work on the same data at the same time with negligible costs. This offers a totally different way of building systems, as the architecture not only removes the need of explicitly transferring data between the two processors, but also makes the processors capable of performing fine-grained cooperation.

Although the host memory shared by the integrated GPU has lower memory bandwidth than that of discrete GPUs, memory intensive workloads such as in-memory key-value stores are still able to benefit from the GPU for its massive number of cores and the capability of hiding memory access latency. With the advanced memory sharing capability and high performance-price ratio, we believe the coupled CPU-GPU architecture is a promising direction for building efficient IMKV systems.

B. State-of-the-Art GPU-Accelerated IMKVs

The workflow of query processing on an IMKV node is as follows. Firstly, packets are processed in the TCP/IP

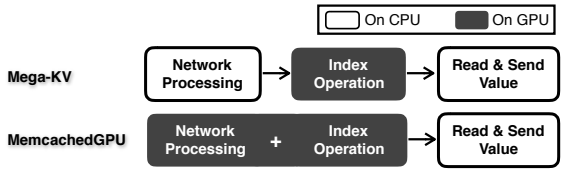

Fig. 2. Pipelines of Mega-KV and MemcachedGPU for GET Queries

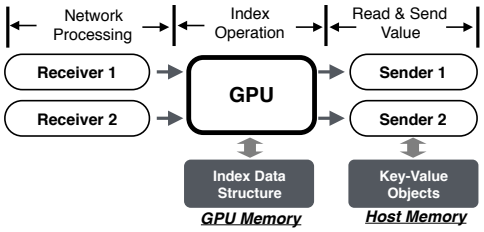

Fig. 3. The Architecture of Mega-KV

stack, then queries in the packet are parsed to extract the semantic information. Three types of queries, i.e., GET, SET, and DELETE, serve as the interface between IMKV and clients. If a GET query is received, the key is looked up in the index data structure to locate its value, then the value is sent to the requesting client. In current implementations such as [1], [6], [7], key comparison is needed, as small and fixed-length key signatures are stored in the index data structure to improve performance. If a SET query is received, memory is allocated for the new key-value object, or an existing key-value object is evicted to store the new one if the system does not have enough memory. For a DELETE query, the key-value object is removed from both the main memory and the index data structure. For processing queries, three types of index operations are performed on the index data structure, which are *Search*, *Insert*, and *Delete*. *Search* operations are performed for all GET queries to locate the values, and the index of a new object is added with an *Insert* operation. For evicted or deleted objects, their indexes are removed with *Delete* operations.

State-of-the-art systems Mega-KV [1] and MemcachedGPU [2] utilize CPU-GPU heterogeneous architectures to improve the efficiency of IMKVs. Because the architecture characteristics of CPUs and GPUs are different, they are only efficient for performing specific tasks [8]. Consequently, Mega-KV and MemcachedGPU both adopt a pipelined model for query co-processing, where CPUs and GPUs are in charge of different pipeline stages. Figure 2 shows their pipelines, where the query processing is partitioned into three main parts: *Network Processing*, *Index Operation*, and *Read & Send Value*. Both systems adopt static pipeline designs. Different with Mega-KV, MemcachedGPU utilizes GPUDirect to directly DMA packets to the GPU memory, and it has only two pipeline stages. In the following of this paper, we focus on Mega-KV as the state-of-the-art system as it achieves the highest throughput.

The main idea of Mega-KV is to utilize GPUs to break the bottleneck caused by the random memory accesses in index operations. In Mega-KV, key-value objects are stored in the host memory, while a cuckoo hash table in the GPU memory serves as its index data structure. Due to the high PCI-e data transfer cost, index operations are processed by GPUs, and tasks such as reading key-value objects are performed on

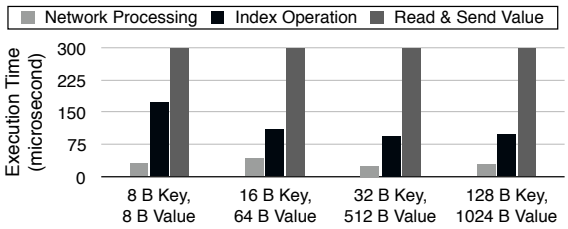

Fig. 4. Execution Time of Mega-KV Pipeline Stages on Coupled Architectures. (95% GET 5% SET, the key popularity follows a Zipf distribution of skewness 0.99)

CPUs. Figure 3 shows the architecture of Mega-KV. Mega-KV implements multiple pipelines to take advantage of the multicore architecture. In each pipeline, three types of threads are in charge of three pipeline stages, respectively. *Receiver* and *Sender* threads handle *Network Processing* and *Read & Send Value*, respectively. A *Scheduler* thread launches GPU kernels periodically to perform index operations.

C. Evaluations of GPU-based IMKVs on APU

To assess if current IMKV system designs are able to well utilize coupled CPU-GPU architectures, we evaluate Mega-KV on an AMD A10-7850K APU. Mega-KV is originally implemented in CUDA, and we port it to OpenCL 2.0 to run on the APU. More detailed experimental setup can be found in Section V. In the following, we present our major findings.

1) Inefficiency in Handling Diverse Workloads: In production systems, different applications of IMKVs have huge variations in terms of GET/SET ratio, request sizes and rates, and usage patterns. Facebook lists five different Memcached workloads, where the GET ratio ranges from 18% to 99%; the value size ranges from one byte to tens of thousands of bytes; key popularity varies significantly. In the workload that stores user-account status information (USR), it has very small value size (2 bytes) [3]. On the other hand, in the workload which represents general cache usage of multiple applications (ETC), the value size distributes very widely where the number of values with sizes under 1,000 bytes is almost the same with that between 1,000 bytes and 10,000 bytes. Furthermore, there are traffic spikes in production systems. The spikes are typically caused by a swift surge in user interest on one topic, such as major news or media events, or it can be caused by operational or programmatic causes [3]. For instance, when machines go down, keys will be redistributed with consistent hashing [9], which may change the workload characteristics of other IMKV nodes.

We firstly evaluate the balance of Mega-KV pipeline with different workloads. Figure 4 shows the measured execution time for all the pipeline stages with four data sets. In the experiments, the maximum execution time of all pipeline stages is controlled within 300 microseconds with the periodical scheduling technique in Mega-KV. Because the random memory accesses involved in accessing key-value objects is the bottleneck of Mega-KV, the execution time of the third stage (*Read & Send Value*) equals to 300 microseconds. Ideally, a balanced pipeline has all the stages with the same execution time so that all processors can be fully utilized. The execution time of *Network Processing*, however, only ranges around 25-42 microseconds. For the 8-byte-key workload, the execution

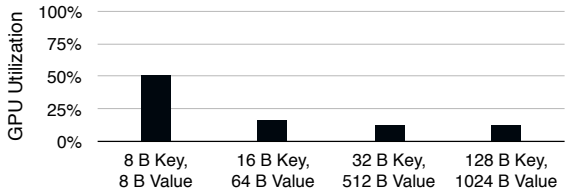

Fig. 5. GPU utilization of Mega-KV on Coupled Architectures

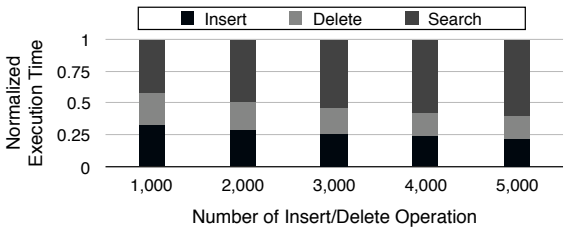

Fig. 6. Ratio of the Execution Time of Index Operations. (95% GET 5% SET, the key popularity follows a Zipf distribution of skewness 0.99)

time of *Index Operation* is 174 microseconds. When the key-value size grows larger, larger number of memory accesses are demanded in *Read & Send Value*. Consequently, fewer queries can be processed within 300 microseconds. As the size of key-value objects has no impact on the execution time of *Index Operation*, it dramatically drops to 97 microseconds due to the smaller batch size. In these experiments, the pipeline of Mega-KV is extremely imbalance for all the workloads.

To understand the impact from the imbalanced pipeline, we measure the GPU utilization of Mega-KV in Figure 5. For all the four workloads, the GPU is severely underutilized due to the imbalanced pipeline. For small key-value sizes, the GPU utilization reaches up to 51%, but it drops to only 12% when the key-value size grows larger. There are two main reasons that result in such low GPU utilization. First, as the load of other pipeline stage is much heavier than that of *Index Operation*, GPUs become idle when waiting for other pipeline stages. Second, GPUs are of low efficiency when the number of queries in a batch is small.

2) Inefficiency of GPUs in Processing SET Queries: Figure 6 depicts the normalized GPU execution time of *Search*, *Insert*, and *Delete* operations of Mega-KV with a read-dominant workload. The horizontal axis indicates the batch size of *Insert* operations, where there will be the same number of *Delete* operations and 19 times *Search* operations (95:5) in the batch. This is because, when the system does not have enough memory, a SET query needs to evict an existing key-value object to store the new object. Consequently, an *Insert* operation and a *Delete* operation are generated for the new object and the evicted object, respectively.

As shown in Figure 6, although both the *Insert* and *Delete* operations take less than 5% of the total number of index operations, they take 26.8% and 20.4% of the overall execution time on average, respectively. The reason is that GPUs are extremely inefficient at handling small batch of jobs, as the large number of cores would become idle. As a result, the GPU spends as high as 35%-56% of its time in processing these two types of index operations. However, due to the static pipeline design of Mega-KV, all index operations have to be performed by GPUs. Consequently, such static pipeline designs

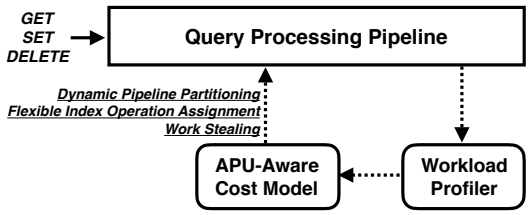

Fig. 7. The Framework of DIDO

cause inevitable performance slowdown on index operations with different computational requirements.

3) Summary: The main reason for the inefficiency of existing GPU-based IMKV system designs on coupled CPU-GPU architectures is that a fixed pipeline design cannot be optimal for all workloads. Although Mega-KV is able to immediately utilize the sharing of the memory space and avoids PCI-e data transfer in the coupled CPU-GPU architecture, its static pipeline design still causes pipeline imbalance and severe resource underutilization for diverse workloads.

As both the CPU and the GPU can directly access the index data structure and the key-value objects simultaneously, either of them can be in charge of any tasks in the workflow of key-value query processing. This can lead to a more balanced system pipeline if it is adjusted according to the workload. Still, there are a number of challenges to well utilize the architecture. First, the pipeline design should be able to adapt to diverse IMKV workloads. In production systems, the workload characteristics of every key-value store system can be of huge difference, let alone they are changing over time. Second, there lacks an effective mechanism to find the optimal pipeline configuration for a workload. The key-value store workload can be different in many terms, including key-value size, GET/SET ratio, and key popularity. Finding the best system configuration for a workload needs to consider various factors in workloads and architectures.

III. DIDO Design

We propose DIDO, an IMKV system with its execution pipeline dynamically adapted to workloads to achieve high throughput. In this section, we introduce the framework and the major techniques adopted by DIDO.

A. DIDO: An Overview

The major design of DIDO is to utilize fine-grained cooperation between the CPU and the GPU to dynamically adapt the pipeline to the workload. Figure 7 sketches the framework of DIDO. DIDO consists of three major components: *Query Processing Pipeline*, *Workload Profiler* and *APU-Aware Cost Model*. *Workload Profiler* profiles workload characteristics, and utilizes a *Cost Model* to guide the selection of the optimal pipeline partitioning scheme. According to the scheme, DIDO remaps tasks to the processors to rebuild the pipeline.

The *Cost Model* and *Workload Profiler* are designed to be lightweight in order to reduce runtime overhead. The *Cost Model* only requires the *Workload Profiler* to profile a few workload characteristics of each batch, including GET/SET ratio and average key-value size. They can be implemented

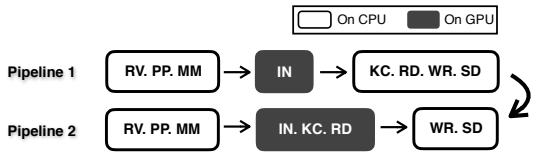

Fig. 8. An Example of Pipeline Changing in DIDO

with only a few counters. With the profiled information of the executed batch, the *Cost Model* is able to get the optimal configuration for the coming batch of workload (Section IV). According to the configuration, DIDO balances its pipeline with a set of techniques, including dynamic pipeline partitioning, flexible index operation assignment, and work stealing. *Workload Profiler* profiles the workload characteristics for each batch of queries. Pipeline adaptation occurs only when the workload characteristics of the current batch have a substantial change compared with those of the previous batch. In our implementation, the upper limit for the alteration of workload counters is set to 10%. If the change exceeds the upper limit, DIDO uses the cost model to calculate the new pipeline configuration for the coming queries.

To allow more flexible and more balanced pipelines, we partition the workflow of a key-value store system into eight fine-grained tasks, which are the granularity for pipeline mapping. (1) *RV*: receive packets from network; (2) *PP*: packet processing, including TCP/IP processing and query parsing; (3) *MM*: memory management, including memory allocation and eviction; (4) *IN*: index operations, including *Search*, *Insert*, and *Delete*; (5) *KC*: key comparison; (6) *RD*: read key-value object in the memory; (7) *WR*: write response packet; and (8) *SD*: send responses to clients. Corresponding to Mega-KV's design, our fine-grained tasks (1)*RV*–(3)*MM* correspond to *Network Processing* of Mega-KV; (5)*KC*–(8)*SD* correspond to *Read & Send Value* of Mega-KV. It is worth noting that when *RD* and *WR* are assigned to different pipeline stages, the task *RD* reads the key-value objects in the memory (random read) and writes them into a buffer sequentially. As a result, the task *WR* on the other stage needs to read the key-value objects in the buffer to construct responses. This transfers random memory read of key-value objects into sequential read in the task *WR*.

DIDO changes its pipeline by assigning the tasks to different pipeline stages. As an example, Figure 8 shows a possible pipeline change in DIDO. The *pipeline 1* adopted by DIDO only assigns index operations to the GPU. After the workload changes, such as smaller key-value size or lower GET/SET ratio, the GPU can be underutilized with the current pipeline. To address the pipeline imbalance, DIDO is capable of moving tasks (e.g., *KC* and *RD*) to the GPU to form a new pipeline, which becomes *pipeline 2* in the figure.

B. Major Optimization Techniques

DIDO balances its pipeline with three major techniques: (I) dynamic pipeline partitioning, (II) flexibly assigning index operations between the CPU and the GPU, and (III) work stealing. These techniques aim at making a balanced system pipeline, where dynamic pipeline partitioning adjusts the load of each pipeline stage; and flexible assigning index operations to the CPU/GPU can dramatically enhance its utilization. After the two techniques are applied, the CPU and the GPU are still

able to perform work stealing to further improve the utilization of the CPU and the GPU.

1) Dynamic Pipeline Partitioning: Static pipeline execution causes severe resource underutilization and pipeline imbalance. To adapt to a workload, DIDO dynamically balances its pipeline by assigning the eight tasks to the CPU or the GPU. We implement the tasks as independent functions. Tasks can be placed in the same stage by performing function calls. When applying a new pipeline, as queries being processed may have gone through some of the pipeline stages, their following processing flows should not be changed. Therefore, due to the batch processing of GPUs, the pipeline configuration is applied to each batch of queries. In DIDO, we embed the pipeline information into each batch to make all pipeline stages know how to process the queries in it. This mechanism ensures that queries can be handled correctly when the pipeline is changed at runtime.

We find that the tasks in query processing are not *performance independent* of each other, where some adjacent tasks achieve higher performance when they are placed in the same pipeline stage. We call this phenomenon as *task affinity*. Consequently, the overall execution time of a query can be different after moving a task to another pipeline stage. For instance, *KC* and *RD* have an affinity. *KC* needs to access key-value objects to compare the keys, which fetches the objects into the cache. Therefore, placing *RD* in the same stage with *KC* would be much faster than performing them on different processors, where the data has to be read into the cache again with a non-trivial overhead. In DIDO, *task affinity* is a major concern in determining the optimal pipeline partitioning scheme.

In a traditional pipeline design, pipeline stages should be balanced for the highest performance, i.e., the execution time of all pipeline stages should be as close as possible. However, this assumption applies only if the execution time of a task is constant when being placed in different pipeline stages, and it does not hold for the pipeline of a key-value store system with *task affinity*. This is because, although moving a task to another stage may lead to less disparity in the processing time, the overall throughput may also drop due to the inefficient placement of tasks. Our cost model takes *task affinity* between tasks into consideration, where the overall throughput is the only criterion for choosing the optimal pipeline partitioning scheme.

2) Flexible Index Operation Assignment: As shown in Figure 6, where the 5% *Insert* and *Delete* operations take up to 58% of the overall GPU execution time, GPUs are inefficient in processing small batches of *Insert* and *Delete* operations. By utilizing the coupled CPU-GPU architecture, DIDO is capable of making the entire system more efficient by assigning different index operations to appropriate processors. As the GET/SET ratio and the load of each pipeline stage vary with different workloads, a fixed index operation assignment policy may also result in pipeline imbalance for certain workloads. We choose to adjust the assignment of index operations according to the workload at runtime. To facilitate flexible index operation assignment in DIDO, we treat *Search*, *Delete*, and *Insert* operations as three independent tasks. Their assignments in the pipeline are determined together with the remaining seven tasks, except that they can be placed in arbitrary orders. According to the cost model in Section IV, the optimal index operation assignment policy and pipeline partitioning scheme can be derived for a workload.

When the index data structure is accessed concurrently, there can be conflicts between the operations. In the current coupled architecture, although memory sharing occurs at the granularity of individual load/store into bytes within OpenCL buffer memory objects, loads and stores may be cached [10]. Built-in atomic operations can be used to provide fine-grained control over memory consistency, e.g., *atomic_store* (WRITE), *atomic_load* (READ), and *atomic_compare_exchange* (CAS). We use *atomic_compare_exchange* for *Insert* and *Delete* operations to avoid write-write conflicts, and *Search* operations employ *atomic_load* to read up-to-date data.

3) Work Stealing: Although the dynamic pipeline partitioning and index operation assignment help improve the overall system throughput, imbalance still happens between the CPU and the GPU for various reasons, such as the coarse granularity of the eight tasks as well as errors in the cost model prediction. We propose to adopt work stealing [11] between the CPU and the GPU to further improve resource utilization. Work stealing can be effectively adopted in DIDO for three main reasons. First, each query is independent of each other, thus queries can be processed in parallel. Second, the sharing of the same memory space in the coupled CPU-GPU architecture makes the overhead of communication and synchronization between the CPU and the GPU extremely low. Third, as each pipeline stage processes a batch of queries in parallel, queries are stored in a buffer and are processed in a FIFO order. The above system and architecture features facilitate work stealing between compute units.

Because the threads in a GPU wavefront are always scheduled for execution simultaneously, they may process multiple queries in parallel. With work stealing, there can be a situation that a wavefront is going to process a set of queries, where parts of them have been processed by the CPU. Threads in both the CPU and the GPU have to check if a query has already been processed or is being processed, and mark the query for processing if it is not. Instead of stealing one query at a time, we propose to steal a set of queries to amortize the synchronization overhead. In DIDO, both the CPU and the GPU grab a set of queries to process. The best granularity for the number of queries in a set should be the thread number of a wavefront, which is 64 in APUs. We implement an array of tags for the CPU-GPU cooperation, where tag i represents the state of queries from $64 \times i$ to $64 \times (i + 1) - 1$ in the batch. The tags are updated with atomic operations when a processor is going to grab the corresponding queries for processing.

IV. Cost Model

Choosing the optimal system configuration for a workload is a critical task for DIDO, especially with many options and tuning parameters. In this section, we develop a cost model to estimate the execution time of each pipeline stage on the coupled architecture, and then use the cost model to determine the best system configuration for a workload.

The evolution of the coupled CPU-GPU architecture brings new challenges for building a cost model. Different with the cost models for database operations [12], [13], our model goes

TABLE I. NOTATIONS IN THE COST MODEL

Notation	Description
XPU	CPU or GPU
N	The number of queries in a batch
F	$F \in$ {RV, PP, MM, IN, KC, RD, WR, SD}
I_F^{XPU}	The number of instructions of task F on XPU
T_F^{XPU}	The execution time of task F on XPU
N_F^M	The number of memory accesses for each query in task F
N_F^C	The number of cache accesses for each query in task F
L_M^{XPU}	The access latency between XPU and memory
L_C^{XPU}	The access latency between XPU and L2 cache
A	The task set of a pipeline stage
T_A^{XPU}	The execution time of task set A on XPU
T_A^{WS}	The execution time of task set A with work stealing enabled
IPC_{XPU}	The peak instruction per cycle on XPU
μ_{N_C,N_G}^{XPU}	Performance interference to the XPU with N_C memory accesses on the CPU and N_G memory accesses on the GPU
T_{max}	The maximum execution time of all pipeline stages
S	The system throughput

beyond existing studies in the following aspects due to the interplay between dynamic pipeline partitioning and diverse IMKV workloads. First, the CPU and the GPU on the coupled architecture can cause serious performance interference to each other [14], which makes the performance prediction for any single compute unit inaccurate. Second, two adjacent tasks in query processing may have *task affinity*, where the performance of the second task can be significantly improved as the data to be accessed has been read into cache by its previous task. As a result, depending on whether its previous task is placed in the same pipeline stage, the execution time of a task may change dramatically. Third, with work stealing, the CPU and the GPU cooperate to process the same batch of queries, where the work partitioning is determined at runtime. All these factors need to be considered in the cost model.

A. The Abstract Model

In the model, the execution time of a task is estimated to be the sum of its computation time and memory access time. The computation time is derived based on the theoretical peak Instruction Per Cycle (IPC) and the number of instructions. The estimation of memory time considers *task affinity* and key popularity. Table I lists the notations in our cost model.

For the computation time, we count the number of instructions running on devices with the same method in [12], and calculate the total computation time of instructions according to the theoretical peak instructions per cycle (IPC) of the processor. The memory access time is estimated according to the cost of memory access and cache access. We estimate the execution time of task *F* on XPU with Equation 1.

$$T_F^{XPU} = N \times (\frac{I_F^{XPU}}{IPC_{XPU}} + N_F^M \times L_M^{XPU} + N_F^C \times L_C^{XPU}) \quad (1)$$

If a task set $A = \{F_1, F_2, ..., F_n\}$ is assigned to a pipeline stage, its execution time on XPU should be the sum of the execution time of all the tasks in it. However, its performance can be interfered by other compute units due to the competition for shared resources, where GPUs can have a higher impact to the performance of CPUs [14]. We use a factor μ_{N_C,N_G}^{XPU} to estimate the performance influence from the other compute unit, and develop a microbenchmark to measure this factor for different situations. Basically, we generate N_C memory accesses on the CPU and N_G memory accesses on the GPU to measure the value of μ_{N_C,N_G}^{XPU}, where N_C and N_G can be estimated with our approach in Section IV-B. The execution time of the task set *A* on XPU is estimated with Equation 2.

$$T_A^{XPU} = \sum_{i=1}^{n} T_{F_i}^{XPU} \times \mu_{N_C,N_G}^{XPU} \qquad (F_i \in A) \qquad (2)$$

If the GPU that is assigned with task set *A* becomes the bottleneck of the system, CPU threads will perform work stealing after they complete their own tasks. Suppose the CPU thread needs to process task set *B* before work stealing, the processing time of task set *A* with work stealing can be derived with Equation 3.

$$T_A^{WS} = T_B^{CPU} + \frac{T_A^{CPU} \times (T_A^{GPU} - T_B^{CPU})}{T_A^{CPU} + T_A^{GPU}} \qquad (3)$$

From Equation 2 and Equation 3, we can estimate the execution time of each pipeline stage with a batch size of N. DIDO adopts the periodical scheduling mechanism of Mega-KV in order to achieve a predictable latency. The scheduling mechanism limits the maximum execution time of each pipeline stage to be within a pre-defined time interval I. According to the required latency, the maximum number of queries in a batch, N, can be calculated by limiting $T_{max} \leq I$. Then we calculate the system throughput S with $S = N/T_{max}$ (4). The goal of our cost model is to find the optimal system configuration that achieves the highest throughput.

B. Adopt the Cost Model in DIDO

We first apply the cost model to the eight fine-grained tasks by considering the couple CPU-GPU architecture as well as workload characteristics, followed by the algorithm for finding the optimal configuration.

Since RV and SD are fixed to run on the CPU, we use a simple profiling-based approach to estimate their execution time. Specifically, we use microbenchmarks to measure the unit cost of each task execution in RV and SD, and then estimate the total cost to be the unit cost multiplying by the batch size N.

For the rest six tasks, the estimation is more complicated, due to the dynamic pipeline partitioning and workload characteristics. In the cost model, values such as N_F^M and N_F^C depend on workload characteristics and implementation details. We estimate N_F^M and N_F^C by either theoretical calculation or microbenchmarks. In IMKV systems, two main data structures are subject to intensive memory accesses in query processing: the index data structure and key-value objects.

Index Data Structure. DIDO adopts a cuckoo hash table as its index data structure [15]. For cuckoo hashing with n hash functions, each *Search* or *Delete* operation theoretically takes an average of $(\sum_{i=1}^{n} i)/n$ random memory accesses ($N_F^M = (\sum_{i=1}^{n} i)/n$, $N_F^C = 0$). Since the amortized cost of an *Insert* operation is $O(1)$ [15], we calculate the average number of accessed buckets for an *Insert* operation at runtime to estimate its memory cost.

Key-Value Objects. For an access to a key-value object of size L, we estimate the memory cost as one memory access ($N_F^M = 1$) and $\lfloor L/C^{XPU} \rfloor$ L2 cache accesses ($N_F^C = \lceil L/C^{XPU} - 1 \rceil$), where C^{XPU} is the cache line size of XPU.

This is because current processors are able to prefetch data into the cache by recognizing the access pattern, which turns the expensive memory accesses into low-cost cache accesses. Therefore, besides the first access to an object is taken as a memory access, the cost of accessing the rest of the object, i.e., the access to the following cache lines, is estimated as cache accesses. N_F^M and N_F^C can be influenced by two critical factors from the workload characteristics and the pipeline partitioning scheme.

The first factor is *task affinity*. If a task and its affinity task are placed in the same pipeline stage, the number of memory accesses and L2 cache accesses are different. For instance, if *KC* is placed in the same pipeline stage with *RD*, the memory cost of *RD* is estimated as $\lceil L/C^{XPU} \rceil$ L2 cache accesses. This makes a huge difference for small key-value objects, because the huge memory access latency would take a large portion in the overall latency. In Equation 2, all tasks except the first one in a pipeline stage estimate memory cost by taking *task affinity* into consideration.

The second factor is *key popularity*. For a skewed key popularity, we suppose that the CPU cache is able to cache the most frequently visited key-value objects. According to the average key-value size and the cache size, the number of key-value objects that are cached can be calculated. Because skewed workloads are well modeled by the Zipf distribution [16], we are also able to calculate the access frequencies of all keys through Zipf's Law. Then we estimate the portion of memory accesses that are turned into cache accesses as $P = \sum_{i=1}^{n'} f_i / \sum_{j=1}^{n} f_j$, where f_i is the access frequency of the ith key-value object when sorted by frequency in descending order, n' is the number of cached key-value objects, and n is the total number of key-value objects. Therefore, N_F^M and N_F^C should be recalculated as $(1 - P) \times N_F^M$ and $P \times N_F^M + N_F^C$ in estimating the memory access cost, respectively.

In Zipf's Law, the skewness of a workload is needed in calculating the access frequency of a key. At runtime, we estimate the skewness with the sampling method in [17], which calculates the skewness according to the access frequencies of sampled keys and their mean frequency. As the overhead of maintaining the frequencies of all accessed keys is huge, we develop a lightweight mechanism for estimation. A counter and a timestamp are added to each key-value object, which are used to count its access frequency and denote a new sampling procedure, respectively. When accessing a key-value object, the counter is initialized to 1 if its timestamp does not match the timestamp of the current sampling procedure, or is increased by one if matches. Within a sampling time interval, the total number of accessed objects and their total frequencies are recorded to calculate the mean frequency. By reading the frequencies of the accessed objects in the next time interval, the skewness can be calculated according to [17].

Finding the optimal pipeline configuration. In DIDO, we search the entire configuration space to obtain the optimal configuration plan. Since we only have a limited number of pipeline partitioning schemes for the eight fine-grained tasks and a limited number of index operation assignment policies, the cost model estimates the system throughput for all the configurations and chooses the one with the highest throughput. The runtime overhead of this cost estimation is very small, as observed in our experiments.

V. Experiment

A. Configurations and Setups

Hardware and Software Configurations. We conduct the experiments on a PC with an AMD A10-7850K Kaveri APU. It consists of four 3.7 GHz CPU cores and 8 GPU compute units, where each GPU compute unit has 64 720MHz shaders. The processor has an integrated memory controller installed with 4×4 GB 1333 MHz DDR3 memory. The peak computing power of the CPU and the GPU in the APU are 118 GFLOPS and 737 GFLOPS, respectively. The 3DMark[1] score of the APU is 3500, and the CinebenchR15[2] score of the APU is 318.

An Intel 82599 10 GbE NIC is installed for network I/O, and IXGBE 4.3.13 is used as the NIC driver. All the experiments are performed by feeding queries generated from another machine to the in-memory key-value store via network with the UDP protocol. To avoid network I/O becoming the bottleneck in the system, queries and their responses are batched in an Ethernet frame as many as possible. DIDO is programmed with OpenCL 2.0. The operating system is 64-bit Ubuntu Server 15.04 with Linux kernel version 3.19.0-15.

Workloads. We adopt the major workloads of YCSB benchmark [18]. As YCSB is unable to alter the key and value size, our benchmark is implemented to be capable of evaluating workloads with different key-value sizes, key distributions, and GET/SET ratios.

In the benchmark, there are four data sets with different key-value sizes: *K8*) 8 bytes key and 8 bytes value; *K16*) 16 bytes key and 64 bytes value; *K32*) 32 bytes key and 256 bytes value; and *K128*) 128 bytes key and 1024 bytes value. Those data sets are commonly used in evaluating IMKV systems [1], [7], [6], [19]. The APU we used for evaluation can only allocate 1,908 MB shared memory between the CPU and the GPU (can be queried by OpenCL interface clDeviceGetInfo()). In the experiments, we store as many key-value objects as possible with an upper limit of the data set size to be 1,908 MB. Therefore, the number of key-value objects stored in the system varies according to the key-value size. We find that the shared memory size has little impact to the system performance when it is far beyond the size of the CPU and the GPU cache, because almost all data accesses become random memory accesses.

Two key distributions, *uniform* and *skewed*, are evaluated. The uniform distribution uses the same key popularity for all queries. The key popularity of the skewed workload follows a Zipf distribution of skewness 0.99, which is the same with the YCSB workload.

The queries have varied GET/SET ratio: 50% GET (50% SET), 95% GET (5% SET), and 100% GET. They correspond to YCSB workloads *A*, *B*, and *C*, respectively. With four data sets, two key distributions, and three GET/SET ratios, there are a total of 24 workloads in the benchmark.

[1] https://www.3dmark.com/

[2] https://www.maxon.net/en/products/cinebench/

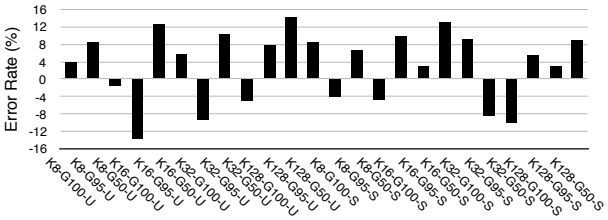

Fig. 9. Error Rate of the Cost Model

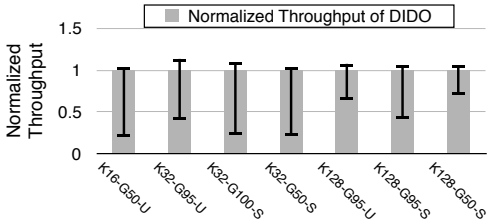

Fig. 10. Comparison between DIDO and the Optimal Configuration

We use a notation with three components to represent a workload, which indicate the key-value size, the GET ratio, and the key distribution, accordingly. For instance, *K32-G95-U* means a workload with the K32 data set (32 bytes key and 256 bytes value), 95% GET(5% SET), and a uniform key distribution (the skewed distribution is represented by *S*).

Comparison. We take Mega-KV as a state-of-the-art IMKV system for comparison. We use Mega-KV (Discrete) to represent its original implementation on discrete architectures, while Mega-KV (Coupled) denotes its OpenCL implementation on the coupled architectures. Mega-KV (Coupled) also takes advantage of memory sharing in the APU architecture to avoid copying data.

Latency. In the experiments, the average system latencies of DIDO and Mega-KV are always limited within 1,000 microseconds (μs) with the periodical scheduling policy [1]. Therefore, DIDO has the same latency as Mega-KV when comparing their throughput.

B. Evaluation of the Cost Model

The cost model plays a key role in choosing the pipeline and index operation assignment policy for DIDO. In this subsection, we first evaluate the accuracy of the cost model in estimating the performance, then we show the effectiveness of our cost model in predicting optimal system configurations for diverse workloads.

We use error rate as the metric to evaluate the accuracy of the cost model. It is calculated as $(T_{DIDO} - T_{Model})/T_{DIDO}$, where T_{DIDO} is the measured throughput of DIDO and T_{Model} is the estimated throughput. As shown in Figure 9, the cost model has a maximum error rate of 14.2% and an average error rate of 7.7%. Overall, our cost model predicts the performance well for all workloads.

To verify the decisions made by the cost model, we evaluate the performance of all possible pipeline mappings and index operation assignment policies in DIDO. We confirm that the pipelines and index operation assignment policies chosen by the cost model are optimal for 17 workloads. For the rest 7 workloads, although DIDO chooses different work partitioning

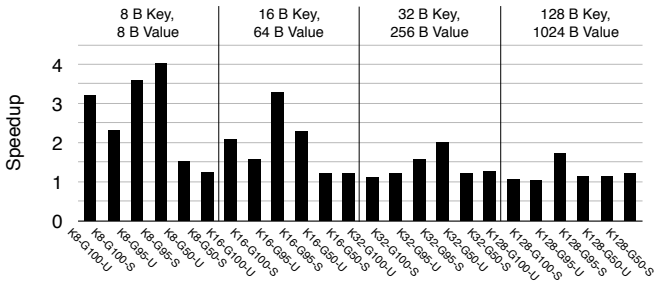

Fig. 11. Throughput Improvement of DIDO over Mega-KV (Coupled)

schemes, the throughput of DIDO is very close to the throughput of the optimal configurations. Figure 10 compares the normalized throughput of DIDO and the optimal configurations for those 7 workloads. Error bars are used to show the range of system throughput of all the possible pipeline configurations normalized to that of DIDO, where the upper end of an error bar shows the highest throughput of the optimal configuration, and the lower end shows the lowest throughput. We have two major findings. First, the average throughput of the optimal configurations is only 6.6% higher than that of DIDO for the seven workloads. Second, the system can suffer an order of magnitude lower throughput if a poor configuration is adopted. This shows the effectiveness of our cost model in choosing system configurations.

C. Overall Comparison

System Throughput. Figure 11 shows the throughput of DIDO for 24 workloads and compares it with that of Mega-KV (Coupled). As shown in the figure, the throughput of DIDO is up to 3.0 times higher than that of Mega-KV (Coupled). On average, DIDO is 81% faster than Mega-KV (Coupled) for the 24 workloads.

These experiments show that, for all workloads, DIDO outperforms Mega-KV (Coupled) by adapting its pipeline to the workload. To gain a deeper understanding of the throughput improvement of DIDO, we analyze the factors that make impacts to the performance improvement of DIDO, including key-value size, GET/SET ratio and key popularity. We compare the pipeline configuration of Mega-KV and DIDO. Note, the pipeline of Mega-KV is static: [*RV*, *PP*, *MM*]$_{CPU}$ $\rightarrow$ [*IN*]$_{GPU}$ $\rightarrow$ [*KC*, *RD*, *WR*, *SD*]$_{CPU}$, where *IN* is processed by the GPU, and the rest tasks are processed by the CPU.

Impact of Key-Value Size. As shown in the figure, the performance improvements for the small key-value data sets are much higher than those of the large key-value data sets. On average, the throughput improvements for data sets *K8* and *K16* are 166% and 95%, while the improvements for *K32* and *K128* are 40% and 23%, respectively.

For small key-value data sets, the load of the last pipeline stage in Mega-KV ([*KC*, *RD*, *WR*, *SD*]$_{CPU}$) is extremely heavy for the large number of memory accesses to key-value objects involved in *RD* and *WR*. Compared with Mega-KV, DIDO assigns tasks *KC* and *RD* to the GPU to balance the pipeline, which becomes [*RV*, *PP*, *MM*]$_{CPU}$ $\rightarrow$ [*IN*, *KC*, *RD*]$_{GPU}$ $\rightarrow$ [*WR*, *SD*]$_{CPU}$. This pipeline partitioning scheme not only significantly alleviates the load of the CPU, but also turns the random memory accesses to key-value objects in the CPU into sequential memory accesses by the separation of *RD* and *WR*.

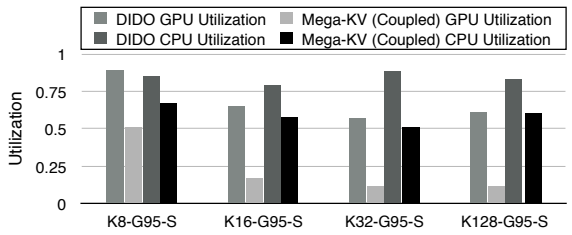

Fig. 12. CPU and GPU Utilization in DIDO

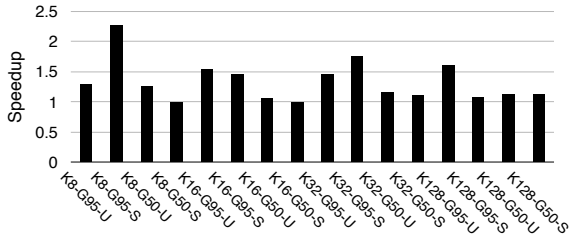

Fig. 13. Performance Improvement By Flexible Index Operation Assignment

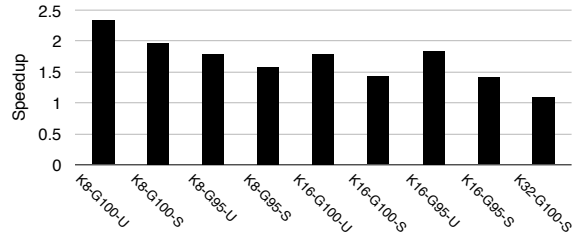

Fig. 14. Performance Improvement By Dynamic Pipeline

The performance improvement is lower for large key-value data sets, because DIDO adopts the same pipeline partitioning scheme as Mega-KV for almost all the large key-value workloads. The main reason is that the CPU is able to utilize hardware prefetching to read large objects, which results in limited benefits of separating *RD* and *WR* to turn the memory accesses into sequential ones.

Impact of GET/SET Ratio. The average throughput improvement of DIDO over Mega-KV (Coupled) for 95% GET (5% SET) workloads is 146%, while that for 100% GET and 50% GET workloads are 71% and 26%, respectively. The improvements for 95% GET workloads are much higher than other workloads, because both dynamic pipeline partitioning and flexible index operation assignment are adopted for them.

For 95% GET workloads, DIDO achieves higher efficiency by assigning all the *Insert* and *Delete* operations to CPUs for processing. After the index operation assignment policy is applied, the GPU has resources for being assigned with more tasks. For instance, DIDO uses the following pipelines for workloads with small key-value size *K8* and *K16*: [*RV*, *PP*, *MM*]$_{CPU}$ $\rightarrow$ [*IN*, *KC*, *RD*]$_{GPU}$ $\rightarrow$ [*WR*, *SD*]$_{CPU}$. These schemes balance the entire pipeline and help to achieve much higher throughput.

For 100% GET workloads, however, as all the index operations are *Search* operations, only dynamic pipeline partitioning is applied (the chosen pipeline is the same as 95% workloads). For most 50% GET workloads, only flexible index operation assignment is adopted, where DIDO assigns *Insert* and *Delete* operations to the CPU. Because there are lots of memory operations brought by the 50% SET queries, i.e., memory allocation and key-value object eviction, assigning more tasks to the GPU would significantly degrade the performance of the CPU due to the resource competition. Therefore, DIDO chooses the same pipeline partitioning scheme as Mega-KV for the 50% GET workloads. Still, the minor performance improvement over Mega-KV (Coupled) is mainly achieved by work stealing.

Impact of Key Popularity. The performance improvements for uniform workloads and skewed workloads are 90% and 71%, respectively. For skewed key distributions, as the most frequently visited key-value objects are cached by the CPU, there are a significantly smaller number of random memory accesses in tasks *RD* and *WR*. As a result, the competition for the shared memory resources is dramatically alleviated. Therefore, the uniform workloads benefit more than the skewed workloads on our techniques to balance the pipeline.

We further study the CPU and GPU utilization of DIDO in Figure 12. For comparison, the four workloads used for evaluation are the ones adopted in evaluating Mega-KV (Coupled) in Figure 5. In DIDO, the GPU utilization is significantly improved to 57% – 89%, which is 1.8 times higher than that of Mega-KV (Coupled) on average. Moreover, DIDO also improves the CPU utilization by an average of 43%, reaching up to 79%. It shows that the dynamic pipeline executions in DIDO is capable of significantly enhancing the hardware utilization.

D. Validation of Major Techniques

Different workloads may benefit from different techniques. In this subsection, we evaluate the three major techniques separately to show their effectiveness in improving the throughput.

1) Flexible Index Operation Assignment: To evaluate the effectiveness of flexible index operation assignment, we fix the pipeline partitioning to the one adopted by Mega-KV: [*RV*, *PP*, *MM*]$_{CPU}$ $\rightarrow$ [*IN*]$_{GPU}$ $\rightarrow$ [*KC*, *RD*, *WR*, *SD*]$_{CPU}$. Figure 13 shows the speedup achieved by the technique for all 95% GET and 50% GET workloads. The baseline for comparison is assigning all index operations to the GPU.

As shown in Figure 13, the throughput is consistently improved across 14 workloads by an average of 37% (at least 5%). The average performance improvement for 95% GET workloads reaches 56%, while that for 50% GET workloads is 10%. For 50% GET workloads, the load of task *MM* becomes heavier as huge amounts of memory management operations are generated by the 50% SET queries. As a result, [*RV*, *PP*, *MM*]$_{CPU}$ becomes the bottleneck of the system after it is assigned with *Insert* and *Delete* operations. Therefore, there is a disparity between the performance improvement for 95% GET and 50% GET workloads.

2) Dynamic Pipeline Partitioning: Compared with Mega-KV, DIDO chooses different pipelines for nine workloads. Their performance improvements are shown in Figure 14. For these workloads, the system performance with dynamic pipeline is an average of 69% higher than that of Mega-KV (Coupled). All these workloads are read intensive, either with

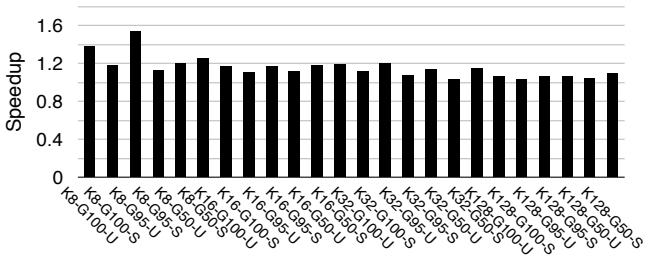

Fig. 15. Performance Improvement By Work Stealing

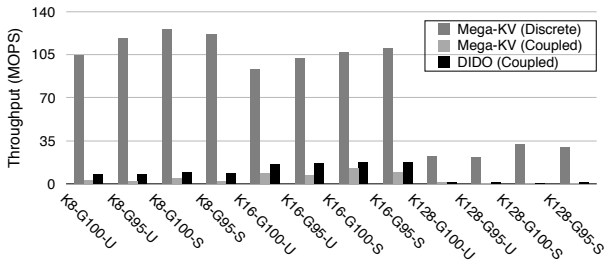

Fig. 16. Performance Comparison between Mega-KV and DIDO

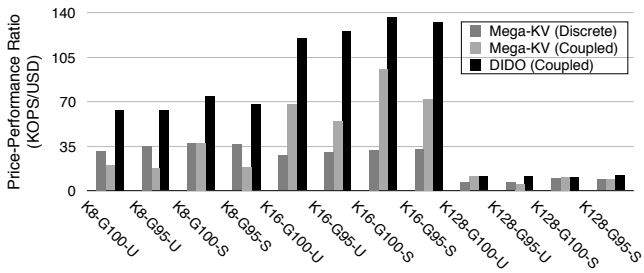

Fig. 17. Comparison of Price-Performance Ratio

100% GET or 95% GET. For 95% GET workloads, the load of the GPU has been dramatically alleviated after the small number of *Insert* and *Delete* operations are assigned to CPUs. As a result, the GPU becomes capable of handling more tasks.

For a substantial number of workloads, we find that assigning *KC* or *RD* to the GPU may degrade the overall throughput even if the GPU is not the bottleneck. Through detailed experimental analyses, we find that moving *KC* to the GPU may even degrade the performance of its previous pipeline stage on the CPU. There are two main reasons for this strange phenomenon. One is the *task affinity* between *KC* and *RD*, and the other is the performance interference between the CPU and the GPU. If *RD* and *KC* are performed on different processors, they would compete for memory bandwidth. As a result, the overall system performance can be degraded due to the resource competition between the two processors.

3) Work Stealing: After configuring the system with flexible index operation assignment and dynamic pipeline partitioning, work stealing is adopted to further improve hardware utilization and system throughput. Figure 15 shows the throughput improvement brought by work stealing for the 24 workloads. On average, work stealing improves the throughput of DIDO by 15.7%. For workloads *K8-G100-U* and *K8-G95-U*, GPU is the bottleneck of the system before work stealing is applied, and CPU cores steal the jobs from the GPU to balance the pipeline. For the rest 22 workloads, the CPU is the bottleneck in the system.

For data sets *K8* and *K16*, the average throughput improvements with work stealing are 28% and 16%, respectively, while the average improvements drop to 12% and 6% for data sets *K32* and *K128*. This is because GPU becomes low efficient for reading or writing large size data. As a result, the benefit is limited for GPUs to perform tasks such as *KC* or *RD* on the stolen jobs.

E. Comparison with Discrete CPU-GPU Architectures

We compare the performance between DIDO and Mega-KV (Discrete) in Figure 16. The performance numbers of *Mega-KV (Discrete)* are from its original paper [1], whose platform is equipped with *two* Intel E5-2650 v2 CPUs and *two* Nvidia GeForce GTX 780 GPUs. Since Mega-KV does not report 50% GET performance, and the value size of 32-byte key workloads is different, we compare the performance on all other 12 common workloads.

We make the following points when comparing DIDO and Mega-KV. DIDO always performs network I/O with the Linux kernel in its evaluation, which overhead is huge. However, for 8-byte key workloads, Mega-KV (Discrete) employs high-performance DPDK NIC driver for network I/O. For the rest workloads with other key-value sizes, Mega-KV (Discrete) is evaluated *without* network I/O. The reason that DIDO does not adopt DPDK is that the Intel-developed DPDK does not support AMD platforms for the specific Intel instructions involved. Therefore, we adjust our experiments for DIDO to match those of Mega-KV. Particularly, for 8-byte key workloads, both Mega-KV and DIDO perform network I/O. For other workloads, both Mega-KV and DIDO read packets from local memory. Thus, after omitting network I/O for those workloads, the throughput of DIDO in Figure 16 is 3.3-5.4 times higher than those in Figure 11.

Performance comparison: Due to the superior performance of the dedicated architecture, Mega-KV (Discrete) achieves 5.8-23.6 times higher throughput than DIDO. But, we argue that our major contribution is NOT on the absolute performance, but on the proposed techniques to improve the performance on coupled architectures.

Price-performance ratio comparison: DIDO shows very high price-performance ratios. Overall, the price of the processors in Mega-KV (Discrete) is 25 times higher than that of DIDO. Figure 17 compares their price-performance ratio, where DIDO outperforms Mega-KV (Discrete) for all workloads by 1.1-4.3 times.

Energy efficiency comparison: We make back-of-envelope calculation to estimate the power consumption of the processors in the systems. The Thermal Design Power (TDP) of the APU is 95W, while that of Intel E5-2650 CPU and Nvidia GTX 780 GPU are 95W and 250W, respectively. We show the energy efficiency of DIDO and Mega-KV in Figure 18. For 8-byte key and 128-byte key workloads, the energy efficiency of Mega-KV (Discrete) is 69%-225% higher than DIDO. For 16-byte key workloads, the energy efficiency of DIDO is 18%-26% higher than Mega-KV (Discrete). It is still inconclusive which system is more energy efficient.

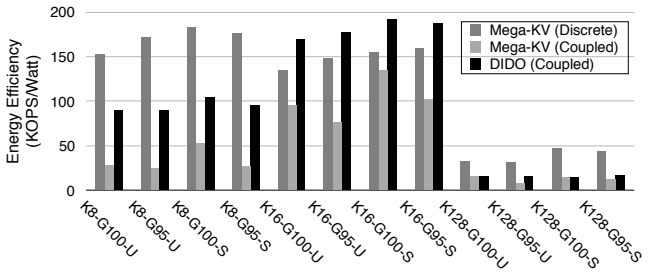

Fig. 18. Comparison of Energy Efficiency

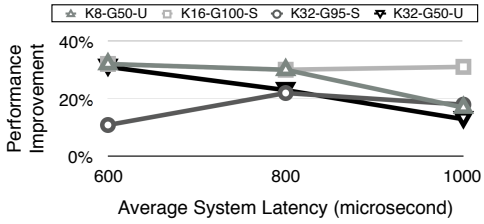

Fig. 19. Performance Improvement with Different System Latencies

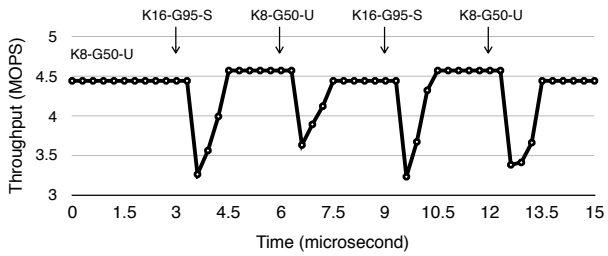

Fig. 20. Throughput of DIDO with Dynamic Workloads

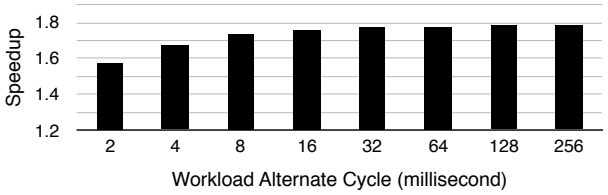

Fig. 21. Impact of Workload Fluctuation between *K8-G50-U* and *K16-G95-S*

F. Other Evaluations

Impact to System Latency. Previous experiments measure system throughput by controlling the average system latency within 1,000 μs. With different system latency requirements, the throughput of a heterogeneous key-value system may also be different. In DIDO and other GPU-based IMKVs, the system latency largely depends on the execution time of the GPU for each batch. A smaller batch size would lead to a lower processing time of each pipeline stage, therefore the overall system latency drops. Moreover, the GPU throughput would also drop for processing small batches. To show the performance improvement of DIDO with different system latencies, we limit the input speed to control the batch size.

We measure the throughput of DIDO and Mega-KV (Discrete) with four representative workloads (K8-G50-U, K16-G100-S, K32-G95-S, and K32-G50-U) by controlling the average system latency within 600 μs, 800 μs, and 1,000 μs. The throughput improvements of DIDO are shown in Figure 19. With 1,000 μs system latency, DIDO improves the system throughput by an average of 20%, and the average improvements are 26% and 27% for 800 μs and 600 μs, respectively. This denotes that DIDO is capable of achieving very good throughput improvement with different system latency configurations.

Dynamic Pipeline Adaption. We design an experiment to measure the capability of DIDO in adapting to dyncamically changing workloads. In the experiment, we choose two typical workloads, *K8-G50-U* and *K16-G95-S*, which are generated alternately for 3 milliseconds. Figure 20 shows the throughput of DIDO that is profiled every 0.3 milliseconds. The pipeline with the highest throughput for *K8-G50-U* is [*RV*, *PP*, *MM*]$_{CPU}$ $\rightarrow$ [*IN*]$_{GPU}$ $\rightarrow$ [*KC*, *RD*, *WR*, *SD*]$_{CPU}$, while the optimal pipeline for *K16-G95-S* is [*RV*, *PP*, *MM*]$_{CPU}$ $\rightarrow$ [*IN*, *KC*, *RD*]$_{GPU}$ $\rightarrow$ [*WR*, *SD*]$_{CPU}$. As is shown in the figure, the throughput won't drop immediately when the workload changes, because there are previous batches of queries in the pipeline remaining to be processed. After that, the throughput drops dramatically due to the mismatch between the pipeline and the workload. DIDO automatically adjusts its pipeline according to the new workload, and is able to recover to the highest throughput with approximately one millisecond.

We develop a stress test to evaluate DIDO with different degrees of workload fluctuation. The experiments are performed by cyclically alternating the workload between *K8-G50-U* and *K16-G95-S*. Figure 21 shows the speedup of DIDO over Mega-KV (Coupled) with different alternate cycles, from 2 ms to 256 ms. As shown in the figure, the speedup is only 1.58 for the 2 ms cycle, and it rises to 1.79 for cycles more than 64 ms. Because, as shown in Figure 20, DIDO needs around 1 ms to adjust its pipeline configurations to recover to the highest throughput. The recovering time may lead to an impact to the performance improvement, but its cost becomes negligible when the fluctuation of the workload is gentle. According to the previous study in Facebook [3], the workload of IMKVs will not suffer such frequent and radical changes as in our experiments. Therefore, the dynamic adaption mechanism in DIDO is sufficient in enhancing the system throughput of production systems.

VI. Related Work

Many research have been conducted on building high-performance IMKVs. Some studies [6], [19], [20] focus on designing efficient index data structures for concurrent access on multi-core CPUs. For instance, MemC3 [6] and CPHash [20] develop highly optimized hash tables for multicore CPUs, and Masstree [19] designs a high-performance trie-like concatenation of B+ trees for key-value stores. Work such as MICA [7] and Pilaf [21] improve key-value store throughput by optimizing network processing.

There is work that enhances IMKV performance by designing emerging hardware (e.g., [22], [23], [24]). With many cores and high memory bandwidth, GPUs tend to meet all the demands for efficient key-value query processing. Mega-KV [1] and MemcachedGPU [2] exploit heterogeneous CPU-GPU architectures to build IMKV systems. Mega-KV utilizes

the massive cores, the capability of memory access latency, and the high memory bandwidth of GPUs to accelerate index operations. Besides the index operations, MemcachedGPU also offloads packet processing to GPUs by utilizing GPUDirect to directly dump packets to the GPU memory. Both of them are developed on discrete CPU-GPU architectures and adopt static pipeline execution designs, which fail to take advantage of couple CPU-GPU architectural features.

A set of research adopts discrete CPU-GPU architectures in database systems [25], [26], [27]. Coupled CPU-GPU architectures have been studied in several data-intensive systems and applications, including relational database systems [12], [13], and irregular applications like graphs [28] and MapReduce [29]. Compared with relational database and MapReduce, IMKV systems have several more challenging aspects including diverse workloads and more flexible pipelines, which require special design and optimization techniques in this paper. On the other hand, Hetherington et. al. [30] port the Memcached implementation to OpenCL to run on an APU. The CPU and the GPU in their APU have separate memory spaces, where data has to be transferred via memory copy. Consequently, the integrated GPU still has to work in the same way as discrete GPUs.

VII. Conclusion

As an emerging architecture, coupled CPU-GPU architectures raise interesting research opportunities and challenges for building in-memory key-value stores. Our study on state-of-the-art GPU-based IMKV systems with diverse workloads show that their static pipeline execution designs cause severe resource underutilization and pipeline imbalance on coupled CPU-GPU architectures. This paper presents the design, implementation and evaluation of DIDO, an in-memory key-value store system with dynamic pipeline executions to resolve those limitations. Our experimental results show that 1) with the guidance of a cost model, DIDO is capable of adapting the optimal pipeline execution configuration to the workload at runtime, and 2) DIDO can significantly improve the overall throughput for different workloads, in comparison with the state-of-the-art IMKV designs.

Acknowledgement

This work is partially funded by a MoE AcRF Tier 1 grant (T1 251RES1610), a startup grant of NUS in Singapore and NSFC Project 61628204 in China.

References

[1] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proc. VLDB Endow.*, vol. 8, pp. 1226–1237, 2015.

[2] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *SoCC*, 2015, pp. 43–57.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012, pp. 53–64.

[4] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP*, 2008, pp. 73–82.

[5] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated cpu-gpu systems," in *MICRO-46*, 2013, pp. 457–467.

[6] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *NSDI*, 2013, pp. 371–384.

[7] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *NSDI*, 2014.

[8] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," in *ISCA*, 2010, pp. 451–460.

[9] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *STOC*, 1997, pp. 654–663.

[10] AMD, "Amd accelerated parallel processing opencl user guide," 2014.

[11] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[12] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled cpu-gpu architectures," *Proc. VLDB Endow.*, vol. 8, pp. 329–340, 2014.

[13] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled cpu-gpu architecture," *Proc. VLDB Endow.*, vol. 6, pp. 889–900, 2013.

[14] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing gpu concurrency in heterogeneous architectures," in *MICRO-47*, 2014, pp. 114–126.

[15] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, pp. 122–144, 2003.

[16] G. Cormode and S. Muthukrishnan, "Summarizing and mining skewed data streams," in *SDM*, 2005, pp. 44–55.

[17] D. N. Joanes and C. A. Gill, "Comparing measures of sample skewness and kurtosis," in *The Statistician*, vol. 47, 1998, pp. 183–189.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[19] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012, pp. 183–196.

[20] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "Cphash: A cache-partitioned hash table," in *PPoPP*, 2012, pp. 319–320.

[21] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store," in *USENIX ATC*, 2013, pp. 103–114.

[22] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ISCA*, 2015, pp. 476–488.

[23] Z. Istvn, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *FPL*, 2013, pp. 1–8.

[24] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *NSDI*, 2016, pp. 425–438.

[25] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang, "Concurrent analytical query processing with gpus," in *PVLDB*, 2014, pp. 1011–1022.

[26] J. Paul, J. He, and B. He, "Gpl: A gpu-based pipelined query processing engine," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD, 2016, pp. 1935–1950.

[27] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo - a vector algebra for portable database performance on modern hardware," *PVLDB*, vol. 9, no. 14, pp. 1707–1718, 2016.

[28] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *CGO*, 2017.

[29] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *SC*, 2012, pp. 25:1–25:11.

[30] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *ISPASS*, 2012, pp. 88–98.