ShadowVM: Accelerating Data Plane for Data Analytics with Bare Metal CPUs and GPUs

Zhifang Li, Mingcong Han, Shangwei Wu, Chuliang Weng
East China Normal University
{zhifangli,mingconghan,swwu}@stu.ecnu.edu.cn,clweng@dase.ecnu.edu.cn

Abstract

With the development of the big data ecosystem, large-scale data analytics has become more prevalent in the past few years. Apache Spark, etc., provide a flexible approach for scalable processing upon massive data. However, they are not designed for handling computing-intensive workloads due to the restrictions of JVM runtime. In contrast, GPU has been the de facto accelerator for graphics rendering and deep learning in recent years. Nevertheless, the current architecture makes it difficult to take advantage of GPUs and other accelerators in the big data world.

Now, it is time to break down this obstacle by changing the fundamental architecture. To integrate accelerators efficiently, we decouple the control plane and the data plane within big data systems via action shadowing. The control plane keeps logic information to fit well with the host systems like Spark, while the data plane holds data and performs execution upon bare metal CPUs and GPUs. Under this decoupled architecture, both the control plane and the data plane could leverage the appropriate approaches without breaking existing mechanisms. Based on this idea, we implement an accelerated data plane, namely *ShadowVM*. In our experiments on the SSB benchmark, ShadowVM lifts the JVM-based Spark with up to 14.7× speedup. Furthermore, ShadowVM could also outperform the GPU-only fashion by adopting mixed CPU-GPU execution.

CCS Concepts: • **Information systems → Data analytics**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords: big data processing, GPU, heterogeneous system

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441595

1 Introduction

In recent years, the big data ecosystem that encompasses various frameworks, including Hadoop, Spark, and Flink [18, 24, 60], has demonstrated the scalability to serve massive data analytics, such as data loading and analytical SQL query. To scale out with commercial servers, these systems have implemented task scheduling and fault tolerance on the cluster. Since network I/O and disk I/O are regarded as the primary bottlenecks in prior systems, they leverage hardware-independent languages like Java or Scala that uses *Java Virtual Machine* (JVM) as their runtime. This design simplifies the implementation of their core infrastructures.

Today, faster network technologies like 10-Gigabit Ethernet and Infiniband, along with large memory and NVMe SSD, make computations the bottleneck again. Under this situation, the JVM is hard to fully exploit hardware potentials [25]. Currently, novel hardware accelerators, such as GPUs, have shown the potentials to deal with computations. Deep learning systems like TensorFlow [13] and Caffe [32] use GPUs to shorten the training time from days to hours or even minutes. So why not bring this idea to big data analytics? How to harness accelerators to big data analytics has become a challenging research hotspot [15, 19, 37, 58].

Taking the case of "Spark+GPU" as an example, we analyze the restrictions of existing works. First, JVM languages are hard to express computations as parallelized GPU functions, often called *kernels*. In fact, the standard JVM could only involve pre-written kernels outside JVM via JNI [54] or RPC [11], while Spark may need to serve ad-hoc workloads. Second, GPU cannot directly utilize intermediate JVM objects hold by Spark. Before migrating data to GPU, Spark should serialize intermediate JVM objects into a GPU-optimized format. Finally, hardware-independent JVM also restricts hardware-aware optimizations. Official Spark 3.0 introduces accelerator-aware scheduling [1]. With this feature, RAPIDS [8] provides a plugin for Spark that rewrites its runtime to execute analytics with GPU. However, since this straightforward way is case-by-case, identical implementations are hard to reuse for other cases. Considering various host systems and accelerators, it will lead to tedious engineering work (e.g., Spark/Flink × CPU/GPU results in four permutations).

This paper presents a new architecture to bypass JVM limitations and exploit bare metal CPUs and GPUs. We introduce *ShadowVM* as an optional runtime, rather than building a new system or rewriting the current systems, such as

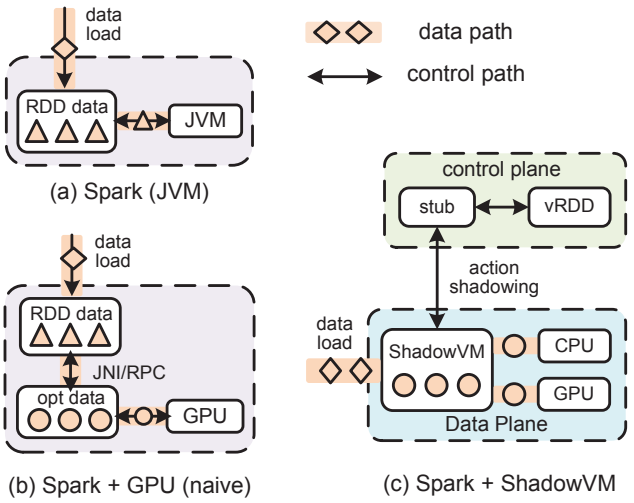


Figure 1. The Architecture of Spark+CPU/GPU.

Spark. ShadowVM can be deployed along with unmodified Spark, in which computations are offloaded to ShadowVM. For better hardware utilization and cost efficiency, ShadowVM maps computations to both CPUs and GPUs at the same time, rather than only CPUs or GPUs. The architecture of ShadowVM could be extended for other similar systems. We conclude our contributions as three points.

Decoupling control plane and data plane through action shadowing. Spark abstracts application data along with various actions as *Resilient Distributed Dataset* (RDD). The default Spark directly leverages JVM to apply analytical actions on the RDD data (see Figure 1 (a)). When employing GPU as the accelerator via JNI or RPC, the additional data path between RDD data and GPUs can become a significant bottleneck (see Figure 1 (b)). To bypass the restrictions of JVM, our initial idea is inspired by the userspace I/O in the OS kernel [16] but considers big data analytics. ShadowVM decouples the whole system as a control plane and a data plane through *action shadowing*. This technique allows the control plane and the data plane to adopt different implementations and optimizations based on their workload features. As shown in Figure 1 (c), action shadowing regards the original RDD as the control plane that does not contain any intermediate data to apply computations. Instead, the control plane employs a stub to push down computations to the decoupled data plane, which is similar to a "shadow" of control plane. This stub captures actions invoked by RDD and makes the data plane replay equivalent shadowed actions with bare metal CPUs and GPUs.

Co-utilizing CPUs and GPUs via vkernel. In the decoupled data plane, ShadowVM executes native CPU/GPU code based on shadowed actions. Prior works employ GPU as main executors [45, 47, 48] and leave CPU as the controller. This design will deteriorate the cost efficiency due to dozens of idle CPU cores. Apart from classic CPU-only and GPU-only modes, ShadowVM could map computations to CPUs and GPUs by introducing the notion of *vkernel*, which can be regarded as an extension of GPU kernel. Through two-layer partitioning, the vkernel is enabled to run on a pod of CPU cores and GPU cards in the data plane simultaneously. To address the asymmetry between CPU and GPU, we employ apriori and elastic policies to schedule vkernel execution. The apriori policy employs a pre-known feature vector to achieve proportional resource scheduling, while the elastic policy attempts to fit into an optimal feature vector at running time and introduces speculative execution to narrow the impact of suboptimal scheduling decisions.

Handling massive data with passive prefetch. Typical GPU-based systems for analytics [5, 8, 15, 47] regard data locality as the first-class consideration and keep data in the device memory as far as possible. However, when a big data system processes massive data with GPU, there still exist two issues. First, the limited PCIe bandwidth incurs a significant lag to migrate data to the GPU side. Second, the limited capacity of device memory is hard to scale with massive data. However, as input data can usually be processed by only one pass in the pipelines, we introduce an opposite way, called *passive prefetch* that keeps data in the main memory and lets GPU pull data on demand. By reducing the pressure on the PCIe bus, it allows ShadowVM to alleviate the two issues.

The rest of this paper is organized as the following. Section 2 introduces background information about Spark and GPU acceleration for analytics. Section 3 introduces the decoupled architecture based on action shadowing. We leave the details of vkernel execution and passive prefetch in Section 4 and Section 5, respectively. Then Section 6 evaluates the performance of ShadowVM. The related work is revisited in Section 7. Finally, this paper is concluded in Section 8.

2 Background

This section introduces the related background information that helps to understand our motivations of ShadowVM.

2.1 Host System of ShadowVM: Spark

Spark is one of the popular systems for large-scale data processing. Spark abstracts the partitioned data in the cluster as *Resilient Distributed Dataset* [59]. SparkSQL provides a flexible API to generate a new RDD from the previous RDDs by applying various actions. For example, the map action applies a user-defined function on every tuple of input RDD, the filter action selects the valid tuples according to the predicate, and the join action probes whether the value of each tuple exists in another reference RDD [14]. As shown in Figure 2, SparkSQL compiles the application into a series of actions on the RDDs. Then Spark executes each action with the JVM on each partition of RDD.

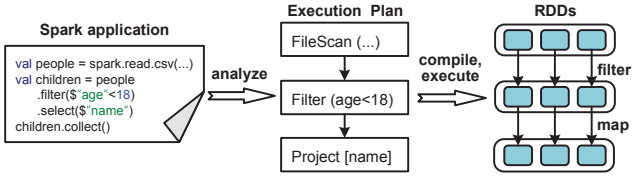


Figure 2. An example of RDDs in Spark.

In recent official Spark 3.0 [9] and thirdparty negotiators, such as Yarn [51] and Mesos [31], accelerator-aware scheduling provides an opportunity to utilize GPUs within the Spark. Deep learning libraries like Caffe [32] and TensorFlow [13] have benefited from this feature to deploy in the GPU cluster by regarding Spark as a scheduler. For data analytics, NVIDIA's RAPIDS [8] provides a plugin for Spark that allows computing RDD data with GPUs rather than the original JVM runtime.

Apache Flink [18] is another popular analytics system. Flink shares many common ideas with Spark, such as data abstraction and partitioning (similar to RDD). The difference is that Flink focuses on on-line analytics, while Spark focuses more on off-line analytics. As Flink also leverages a coupled architecture based on JVM, it still suffers from similar issues when it works with GPUs [19].

2.2 GPU for Data Analytics

CUDA [42] is the de facto platform for GPU programming that abstracts computations as a *kernel* that consists of numerous GPU threads. Unlike a pthread on the CPU that sequentially processes each tuple of input dataset, a kernel is launched to process the whole input dataset in parallel. In addition, GPU integrates device memory that provides much higher bandwidth than the main memory on the CPU side.

GPU has potentials in analytical actions such as join and aggregation. Intuitively, each action can be implemented by a kernel that transforms the input dataset into the output dataset. However, due to GPU's discrete memory space, mainstream GPU-based analytical systems rely on "active copy" [15, 47], where data should be explicitly cached to device memory before kernel execution. The cost of active copy can also slow down the time to execute analytics on the fly. Besides, the capacity of device memory restricts the potential to handle massive data.

To accelerate Spark's RDD actions by GPU, the most straightforward way is using JNI or RPC calls that invoke pre-written kernels for computing [37, 58]. This method is hard to support ad-hoc analytics and also limits the optimization space. For example, RDD actions compute intermediate data tuple by tuple, while GPU is optimized to process aligned and columnar data. Thus, RDD should convert intermediate data into GPU-optimized format before migrating data to GPU's device memory and invoking the kernel [15].

3 Decoupling Data Plane through Action Shadowing

Decoupling data plane is not a totally new idea, and the OS field has employed this notion to bypass trapping into the OS's kernel space during I/O syscalls [16]. Action shadowing is inspired by this general idea and extends it to the scenarios of big data analytics with accelerators.

In Spark, RDD deals with both scheduling and execution within JVM. By decoupling the data plane from the control plane of RDD, each part can adopt an appropriate design based on its workload. We treat the original RDD as the control plane that reuses current Spark components, including programming API, task scheduling, and fault tolerance, except for its block manager and computing engine based on JVM. The data plane is similar to a "shadow" of control plane that holds actual data in the bare metal space, where CPUs and GPUs could directly compute data by replaying the equivalent actions that come from the control plane.

Without the restrictions of coupled data plane, ShadowVM could fully make use of accelerators while also fits in with the current ecosystems. Figure 3 shows the decoupled architecture of "Spark+ShadowVM" based on action shadowing. We describe how this fashion works from the perspective of control plane and data plane, respectively.

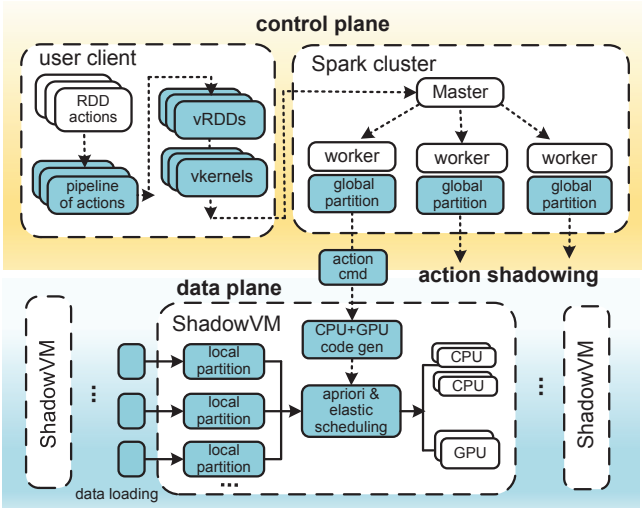


Figure 3. The Decoupled Architecture of Spark+ShadowVM.

3.1 Control Plane

To offload computations from Spark to the data plane of ShadowVM, we should implement "virtual RDD" (i.e., vRDD) as the control plane. As its name implies, vRDD does not hold any actual data for computing. Instead, a vRDD can be treated as a controller, while the original RDD employs JVM to apply actions on input data. vRDD cannot apply the action on data directly but will construct an "action command" that

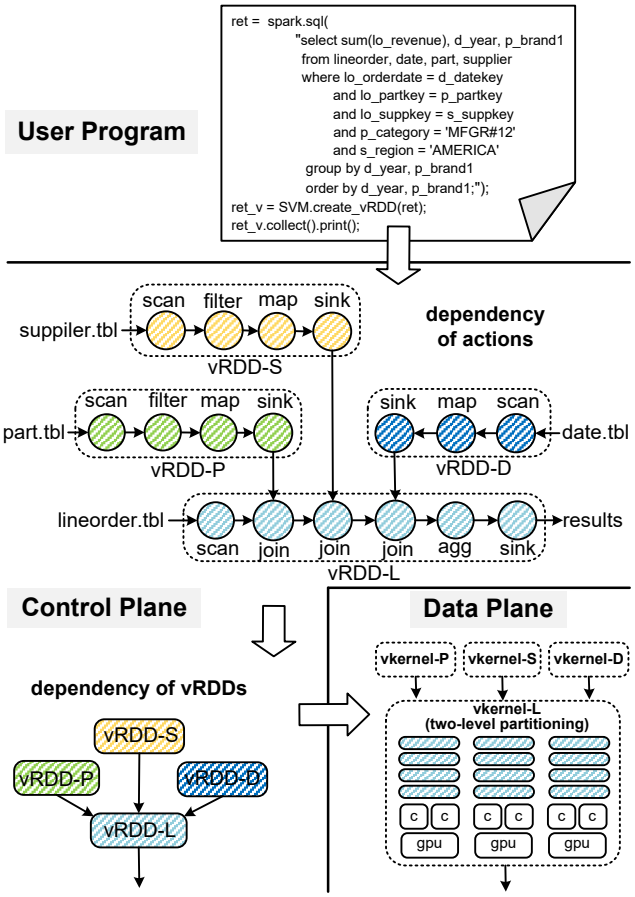


Figure 4. Constructing vRDDs for SSB's Q2.1.

presents the description of actions. Each partition of vRDD will send this action command to the data plane through a pre-connected stub. In the decoupled data plane, ShadowVM nodes will replay the equivalent actions on each partition.

To support various analytics workloads with ShadowVM, there are two categories of built-in actions: I/O and computation. Typical I/O actions, such as scan and sink, are responsible for loading data from external sources before computing and materializing intermediate results after computing. Then computation actions, such as map, filter, and join, manipulate data in the decoupled data plane that leverages CPUs and GPUs for execution.

ShadowVM provides a client library that allows the user to construct the vRDDs directly according to the data dependency among these actions. Besides, the vRDDs could also be extracted from existing Spark applications written by the structured RDD library, such as Spark SQL. By traversing the RDDs from bottom to top, the client of ShadowVM will obtain their data dependency and replace the original RDDs with the equivalent vRDDs supported by ShadowVM.

To mitigate the overhead of launching CPU pthreads and GPU kernels, ShadowVM does not create an individual vRDD after each action. Instead, we divide all actions into multiple "pipelines" to execute multiple adjacent actions by one shot. The division of pipelines is determined by the position of blocking actions that require materialized memory data. For example, a join action requires reference data produced by another pipeline. Therefore, each pipeline normally ends with a "sink" action to materialize output data as the input of the next pipeline, which will involve a synchronization barrier before reaching the subsequent pipelines.

Based on the above principle, ShadowVM creates a vRDD for each pipeline, and the dependency relationship among vRDDs is determined by their contained actions. We implement vRDDs based on the normal RDDs, where the unary dependency is handled by map, and the binary dependency is handled by union and reduce. Note that, the scheduling of vRDDs also reuses Spark's RDD implementations. In original RDDs, massive data tuples are transferred between the precursor RDD and the current RDD. But for vRDD, it only transfers a synchronization barrier that notifies the precursor vRDD has accomplished. Therefore, a vRDD will be scheduled when all its precursor synchronization barriers are released. The vRDD encapsulates the description of each pipeline of actions into an action command and sends it to the decoupled data plane. Then each partition of data plane will invoke the equivalent actions with CPUs and GPUs.

Figure 4 displays an example to construct vRDDs for Q2.1 in *Star Schema Benchmark* (SSB) [44], a typical analytics workload. Q2.1 could be executed as four vRDDs (i.e., `vRDD-P/D/S/L`). Usually, in a vRDD, scan is the first action to fetch external data. Then filter, map, join, etc., apply computations on the input data and push results to the next action. Finally, sink materializes output data in memory. In Q2.1, `vRDD-L` processes the main input data and generates final results, which also requires intermediate results from the other three precursor vRDDs.

Even though the notion of vRDD is made for Spark, the idea of action shadowing could also be extended to other host systems by adopting specialized control planes for them. Taking popular Flink as an example, we could leverage a similar implementation of control plane called "vOperator" on the top of Flink's original operator. Since Flink is optimized for on-line processing, the vOperator continues to run and invoke action commands for each time/count window rather than the whole input data in Spark.

3.2 Data Plane

After the control plane pushes down computations via action shadowing, the data plane is based on native C++/CUDA code and processes data outside JVM-based RDD. This data plane could consist of a cluster of ShadowVM nodes to work with its host system, i.e., Spark. Each ShadowVM node is assigned with local CPU cores and GPU cards, along with the given amount of main memory (for CPU) and device memory (for GPU).

ShadowVM supports three execution modes: "CPU-only", "GPU-only" and "mixed CPU-GPU". The first two modes handle all partitions with either CPU pthreads or GPU kernels, respectively. In Spark or similar systems, the dataset of RDD is partitioned as multiple symmetrical partitions. Intuitively, this approach works naturally under CPU-only (or GPU-only in RAPIDS [8]) modes. However, considering the significant differences between CPUs and GPUs, the traditional single-level partitioning seems to be infeasible in the mixed CPU-GPU mode.

To address this asymmetry, the data plane executes the invoked computation actions in the form of *vkernel*. Unlike the typical GPU kernel, a vkernel runs on a cluster of CPUs and GPUs through two-level partitioning (global+local) and enables the mixed execution mode (CPU+GPU). First of all, a vkernel is divided into multiple global partitions. Each global partition can be handled by a ShadowVM node rather than a CPU core (or a GPU card) in the single-level partitioning. After that, local partitioning further divides a global partition into fine-grained local partitions. In each ShadowVM node, CPU cores and GPU cards are assigned to handle these local partitions based on the mixed scheduling policy.

Apart from accelerating computations, the decoupled data plane also benefits I/O actions, such as data loading, since native CPU code and syscalls are more efficient than managed JVM. As raw data is directly loaded into the data plane in a columnar and aligned format optimized for both CPU and GPU, subsequent computation actions on the data plane do not involve extra conversion overhead.

3.3 Deployment

From the perspective of Spark, the vRDD is still a normal RDD. Thus, an unmodified Spark cluster could run vRDDs to play as the control plane. Once the data plane is decoupled from RDDs, each ShadowVM node could be deployed as a long-running daemon process that provides the computing service. Besides, this decoupled design also enables the data plane to reuse GPU context, while traditional Spark+GPU systems have to re-establish GPU context for each job due to the limitation of Spark's RDD.

This shared-nothing feature makes it easy to deploy ShadowVM with Spark or other host systems. Our implementation uses Protobuf [7] and gRPC library [4] to connect vRDDs with ShadowVM nodes. Note that the message to transfer in the RPC call only contains the lightweight description of action commands, instead of heavy-weight RDD data.

4 Co-utilizing CPUs and GPUs via vKernel

In many systems, such as TensorFlow [13], the GPU-only mode often has a higher priority since GPU is much more powerful than CPU in applications with compute-intensive iterations, such as model training in deep learning.

However, in data-intensive analytics, things get different. Massive data will be scanned by a series of actions by one pass (e.g., analytical query and ETL pipelines) so that the performance gap between CPUs and GPUs is relatively smaller. Thus, co-utilizing CPUs and GPUs helps to improve cost efficiency in the CPU-GPU nodes, which are now common in the clouds. This section introduces how to run vkernels on the data plane with CPUs and GPUs.

4.1 vKernel Execution

To contact with bare metal hardware directly, the data plane compiles vkernel into native code for CPU and GPU. This is inspired by the code generation technique in database systems that compiles the SQL query into customized code to reduce bothersome logic branches, which can harm execution performance, especially on GPU.

Previous works tend to generate code for either the whole SQL [17, 25] or the individual operation in the SQL [23]. The former achieves better performance by eliminating calls for each action but cannot work with heterogeneous and distributed processing, while the latter results in more expensive kernel calls. As a trade-off, we generate code in the granularity of vRDD. This design is compatible with the heterogeneous and distributed setting and enables the data plane to run multiple actions of a vRDD by one shot.

Technically, as CPU and GPU are different platforms, building a unified code generator is challenging. For example, CPU code and GPU code have different fashions for data parallelism. Besides, reusing implementations for different platforms is also a non-trivial issue, which helps to improve modularity and extensibility. The data plane of ShadowVM leverages a "blueprint-based" approach that divides this issue into two steps: 1) how to achieve data parallelism on the target platform and 2) how to implement each action in detail. Based on this idea, we design a two-step code generator that compiles the vkernel into target CPU/GPU code as follows.

Building Blueprint for Platform. The first step is to build a "blueprint", which outlines the parallel execution model on the target platform. Taking Figure 5 as an example, the blueprint for CPU contains a loop that serially reads tuples from the scan, and then performs filter and map on each tuple. In the GPU blueprint, each tuple is processed by a single CUDA thread. To locate the tuple, each CUDA thread computes its position index from `blockIdx` and `threadIdx`. This blueprint regards the input partition for scan and the output partition for sink as its arguments. Note that, this blueprint employs the push-based execution to optimize for both CPU and GPU [17], in which actions are executed by the dataflow order in the pipeline (i.e., from scan to sink). Until now, this blueprint has not taken the details of each action into account and lefts `$(action_block)` as the placeholder before the next step.

Rendering Blueprint with Implementation. Then the code generator renders the blueprint into target code by

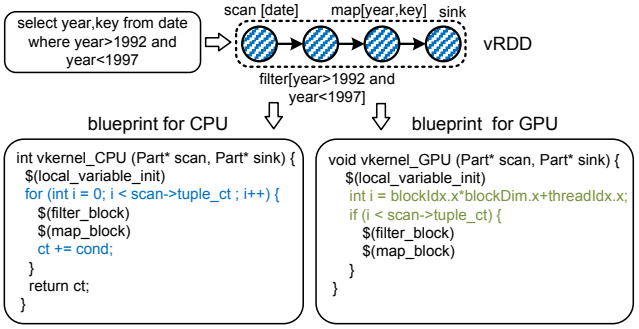


Figure 5. Blueprint Example.

replacing placeholders with their C/C++ implementations. Since the data plane of ShadowVM organizes data in a columnar format, the pointers of required columns are initialized before execution. Notice that filter has different optimizations for CPU and GPU as shown in Figure 6. The CPU version maintains a counter to track the number of valid tuples that match the condition. Considering the expensive atomic instructions to increase the counter shared by CUDA threads, the GPU version requires an extra *mask column* that indicates whether each tuple is valid. Thus, the GPU version involves additional materialization to compact invalid tuples in its output. After code generation is finished, we leverage `clang` and `nvcc` to compile target code into the executable objects (i.e., `vkernel.so` for CPU and `vkernel.ptx` for GPU). To reduce the compilation time, compiled executable objects are cached and reused for the repeated vkernel. In the blueprint-based method, *user-defined functions* (UDFs) are supported by allowing the user to register external C/C++ inline functions. As a limitation, it does not support Java-based UDFs yet, which might be solved by language virtualization [26].

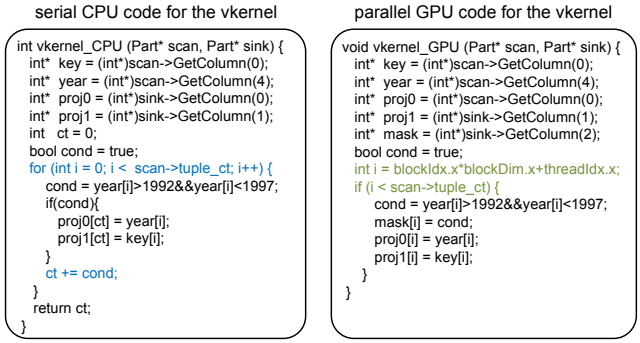


Figure 6. Target Code Example.

4.2 Mixed CPU-GPU Mode

After compiling the vkernel into executable objects, the data plane creates pthreads or kernels based on compiled objects. Ideally, with more computing resources, the mixed CPU-GPU mode will achieve better performance than the GPU-only mode (and CPU-only mode). Unfortunately, we notice that the mixed mode may show suboptimal performance due to the tail latency issue in data analytics.

As we know, CPU and GPU have asymmetric hardware features (e.g., memory/PCIe bandwidth and FLOPS), and a node owns a different number of CPUs and GPUs. Many prior GPU-aware schedulers [50] are mainly optimized for throughput, while data analytics aims to minimize the response time. As a local partition has a different time on a CPU core or a GPU card, the potential stragglers will slow down the whole vkernel that should wait for all local partitions. In this situation, the mixed mode can be worse than the GPU-only mode, even though more resources are utilized.

To exploit asymmetric CPUs and GPUs in the analytical workloads, we employ *apriori* and *elastic* policies for mixed execution, which consist of three core parts:

❶ Two "ready queues" that assign local partitions to the CPU and GPU sides, respectively.

❷ A "classifier" to divide even local partitions and dispatch them to two ready queues proportionally.

❸ An optional "balancer" for the elastic policy that enables speculative execution.

The apriori policy is an ideal way when the workload information is known in advance. It uses a resource vector $< n_c, n_g >$ that indicates the number of CPU cores and GPU cards, along with a feature vector $< \gamma_c, \gamma_g >$ that indicates the relative throughput to run the local partition with a CPU pthread or a GPU kernel. Thus, if a vkernel has been run under the CPU-only and GPU-only modes, the feature vector can be calculated from its execution statistics.

Intuitively, the execution time of a vkernel is minimized when all ready queues become empty simultaneously. With the feature vector $< \gamma_c, \gamma_g >$ and the resource vector $< n_c, n_g >$, the apriori policy leverages a scheduling vector $< \frac{\gamma_c n_c \cdot N}{\gamma_c n_c + \gamma_g n_g}, \frac{\gamma_g n_g \cdot N}{\gamma_c n_c + \gamma_g n_g} >$ that indicates the number of local partitions handled by the CPU and GPU side within N local partitions. The classifier divides all N local partitions into two ready queues by the proportion of scheduling vector. Based on the producer-consumer relationship, each ready queue runs assigned local partitions once the CPU or GPU resources become available. The vkernel is returned after all local partitions are accomplished.

4.3 Elastic Policy

The feature vector of vkernel often varies with different workloads. When prior information is unknown, the apriori policy is infeasible. Thus, we employ the elastic policy that fits the feature vector and employs it to guide scheduling on the fly. We notice that each local partition in a vkernel will execute the same actions with similar computing. It implies

that the feature vector can be learned from a part of finished local partitions. Even though the learned feature vector is more accurate with more statistics, we cannot wait for all local partitions as the vkernel will return at that time.

The elastic policy performs the above step in k folds. It is similar to the classic *mini-batch gradient descent* (MBGD) in machine learning that scans a part of data at a time to learn unknown parameters. The classifier initializes the scheduling vector as an even value $<0.5N, 0.5N>$, which implies the initial feature vector is $<0.5/n_c, 0.5/n_g>$. Unlike the apriori policy, the classifier does not handle all N local partitions instantly but carries out classification in k folds (e.g., $k = 5$). In each quantum, with the statistics of finished local partitions, the classifier updates the feature vector with a learning rate of $1/k$. It means that we only update $1/k$ value of the feature vector at each time to prevent overfitting. Based on the updated feature vector, the classifier dispatches N/k local partitions to two ready queues. The above steps are repeated until all pending local partitions are classified.

Even if the elastic policy provides an approximate solution, it is still hard to gain an accurate feature vector due to laggard statistics. This fact reduces the correctness of elastic policy inevitably. As a result, potential stragglers could still slow down the whole vkernel. The elastic policy then employs a balancer to mitigate the error of feature vector. This idea is borrowed from *speculative execution*, a classic method in distributed system [41] and architecture [35] that starts new tasks before waiting for the slow tasks.

We extend speculative execution to resolve the tail latency issue in the mixed mode. When the ready queue for GPU is empty, the balancer activates speculative execution, which replicates partitions in the ready queue for CPU and pushes them into the ready queue for GPU. For each pair of replicated local partitions, the redundant one can be removed from the ready queue once the earliest side returns the computed results. With this approach, the vkernel can be returned before waiting for the stragglers. Even if speculative execution involves redundant computations, it reduces the response time of vkernel in data analytics. Algorithm 1 concludes the elastic policies with speculative execution.

Algorithm 1: The Elastic Policy for Mixed Execution.

Input: N local partitions, $<n_c, n_g>$

```
wq ← { N local partitions } ; // waiting queue
rq_c, rq_g ← φ, φ ; // ready queue
fq ← φ ; // finished queue
speculative ← false ;
<γc, γg> ← <0.5/nc, 0.5/ng>;
while fq.size() ≠ N do
    if wq ≠ φ then
        sched_set ← pop N/k local paritions from wq ;
        update 1/k of <γc, γg> with the current statistics ;
        rq_c, rq_g ← classify sched_set according to the
          proportion of < γc·nc·N / (γc·nc+γg·ng), γg·ng·N / (γc·nc+γg·ng) > ;
    end
    push rq_g.pop_fin() ∪ rq_c.pop_fin() into fq ;
    if speculative then
        pop fp ∩ rq_c from rq_c, pop fp ∩ rq_g from rq_g ;
    end
    schedule rq_c, rq_g with nc× CPU cores and ng× GPU
      cards based on the producer-consumer relationship ;
    if rq_c ≠ φ and rq_g = φ then
        // activate speculative execution;
        rq_g ← rq_c.replicate();
        speculative ← true ;
    end
    move to the next scheduling quantum;
end
```

5 GPU Execution with Passive Prefetch

Modern GPUs are equipped with high bandwidth device memory. In many GPU-based systems [8, 15, 47], *active copy* is a widely-used strategy that keeps data in device memory as far as possible. To handle a local partition, it applies `cudaMemcpy` before and after kernel execution. This strategy mainly focuses on the medium-scale and iterative workloads, such as data visualization and deep learning, where cached data in device memory will benefit subsequent kernels.

However, to cope with data analytics with a relatively lower computing density, the hotspot will become the limited PCIe bandwidth [57] (about 12GB/s) that involves an extra lag of data transfer. Furthermore, the limited capacity of device memory also restricts the potential to process massive data that analytics systems usually face. For instance, it is difficult to handle a 100GB workload with a single V100 GPU (16/32 GB device memory).

A well-known optimization of active copy is to overlap PCIe transfer with computations. In CUDA, *HyperQ* [36] provides multiple streams to overlap concurrent operations. Technically, even with perfect overlapping, the throughput of GPU execution cannot exceed the bandwidth of PCIe bus in essence. Considering the 12GB/s bandwidth, 100GB of data requires more than 8.3 seconds of processing time.

To cope with restriction, we employ *passive prefetch* that not only overlaps transfer and computations but also reduces the pressure on the PCIe bus. This approach comes from three phenomenons we observed in data analytics. 1) First, an action in the vkernel often does not require all columns of input tuples. For example, in SSB Q2.1, only four of thirteen columns of the fact table are consumed. 2) Second, not all input tuples are consumed by every action. If an action fails to match predicates, subsequent actions will terminate early before consuming subsequent data. This is common when there are multiple selective actions, such as filter and join. Active copy is hard to optimize the above two issues as the accurate read set is unpredictable before execution. 3)

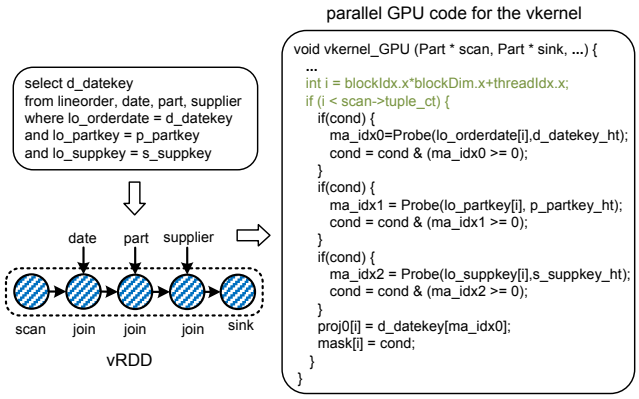


Figure 7. Multi-way Joins with Passive Prefetch.

Finally, unlike deep learning with iterative computation, in data analytics, data goes through GPU by only one pass. As a result, we gain less profit from data locality but pay the cost of PCIe transfer and the limitation of workload size.

Considering the above features, passive prefetch delays the timing to copy data until data is actually consumed by the GPU kernel. Passive prefetch is based on *unified virtual memory* (UVM), a recent method for fine-grained overlapping [33, 49] via the page fault mechanism. UVM allows the kernel to fetch the target pages via pointers, and also uses main memory as the swap space to hold oversubscribed data from device memory. Passive prefetch changes UVM's default locality policy to keep data in main memory rather than enabling automatic CPU-GPU swapping.

Passive prefetch is integrated with the code generator that produces GPU code based on the above principle. For each action of generated kernel, if the required data locates on the GPU side, the GPU thread could directly consume it. Otherwise, the GPU thread fetches input data from the main memory by accessing the column pointers linked to the required items. Instead of pre-loading all data, the step to load data from the main memory is activated only when the current action starts, which involves page faults to copy data through the PCIe bus. The GPU's scheduler will suspend GPU threads and resume them once the required data is prepared in the L1 cache. The intermediate data is cached in the register file (and shared memory) before the next action starts. In the last action, the computed results are written back to the main memory. Figure 7 presents an example of target GPU code based on passive prefetch when multiple adjacent joins lie in the vRDD.

Note that even though we employ passive prefetch as the primary approach, it still needs to work with active copy as different parts of data may have different access patterns. For example, in the map or aggregation actions, we only use passive prefetch. But in the join action, we apply passive prefetch for the fact table and active copy for the reference table that can be used by GPU multiple times.

6 Evaluation

This section experimentally evaluates ShadowVM and analyzes our observations. Our experiments leverage two types of node settings. The "CPU-only" node contains 16 CPU cores for computing (Intel Xeon Silver 4110) and 128 GB main memory. The "CPU-GPU" node is similar to the "CPU-only" node but contains an extra GPU card (Titan RTX with 24 GB device memory) for computing and 8 reserved CPU cores for GPU scheduling. Each node uses OpenJDK 8 and CUDA 10.0 as its basic environment.

To evaluate the case of analytical workloads, we adopt *Star Schema Benchmark* (SSB) [44]. SSB includes 13 SQL queries that perform filter, map, aggregation, join, and group-by actions upon the main fact table of commercial data along with at most four reference tables. The scale factor of SSB is set to SF20 (roughly 20GB) with 100 local partitions to simulate the medium-scale workloads and SF100 (roughly 100GB) with 200 local partitions to simulate the large-scale workloads. An important reason to choose SSB is that it does not contain the network-intensive shuffle action, while this paper focuses on computing performance. Besides, we also employ *Yahoo Streaming Benchmark* (YSB) [12] for a case study of on-line advertisement analysis, which executes filter, join, and map within the fine-grained windows.

6.1 Overall Performance

To evaluate the overall performance in analytical workloads, we compare ShadowVM with JVM-based Spark and typical GPU-based analytical systems, including OmniSci [47] and RAPIDS [8]. Considering the power of GPU, to make a fair comparison, we leverage three CPU-only nodes for JVM-based Spark and one CPU-GPU node for ShadowVM and GPU-based systems. To ignore the cost of disk scan, we pre-cache data in the main memory. We do not pre-cache data in the device memory to simulate the ad-hoc scenarios. As for JVM-based Spark, we enable off-heap memory to reduce JVM's GC overhead. ShadowVM is set as CPU-only, GPU-only, and CPU-GPU mixed (elastic policy) modes, and also enables passive prefetch for GPU execution.

Figure 8 reports the job execution time of 13 SQLs of SSB on the medium-scale SF20 workload. Overall, ShadowVM in CPU-only, GPU-only, and mixed CPU-GPU modes outperforms JVM-based Spark with 6.4×, 8.9×, and 10.8× speedups on average. Specifically, ShadowVM in CPU-only mode achieves up to 2.1× speedups on Q1.x (simple agg + join), 13.6× speedups on Q2.x/Q3.x (complex agg + three joins), and 9.2× speedups on Q4.x (complex agg + four joins). The speedups of ShadowVM mainly come from the decoupled data plane that reduces the overhead to exploit CPU hardware (naive C++ code is more efficient than managed JVM code). Furthermore, in the GPU-only mode, the speedups increase to 2.5×, 15.1×, and 12.5×, respectively. This trend implies that GPU could achieve better performance under

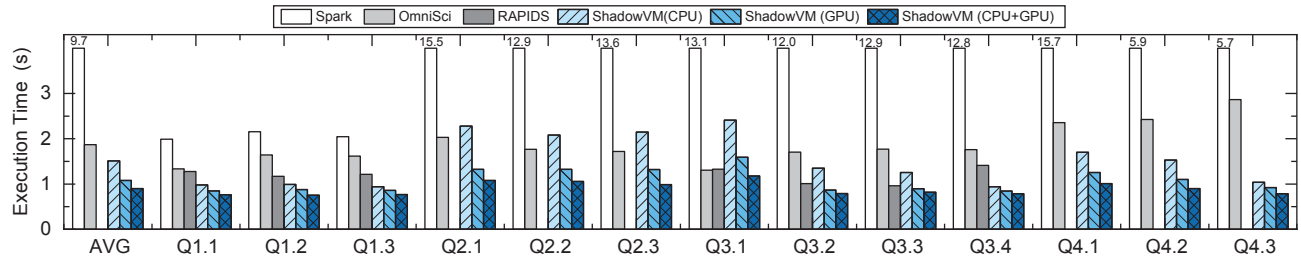


Figure 8. Medium-scale Workload (SSB SF20).

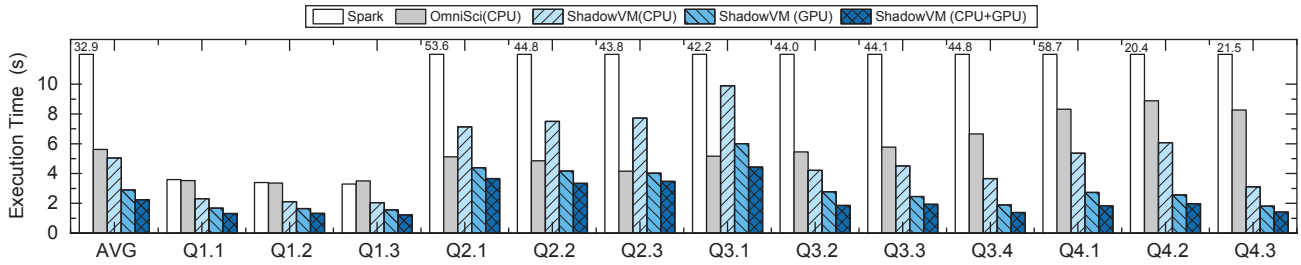


Figure 9. Large-scale Workload (SSB SF100).

the computing-intensive cases, especially for Q2.x/3.x/4.x that contain more expensive aggregation and join actions than lightweight Q1.x.

Compared with GPU-based OmniSci, ShadowVM in the GPU-only and the mixed mode shows 1.7× and 2.1× performance on average. This gap between OmniSci is caused by two reasons. First, OmniSci is based on active copy and does not overlap PCIe transfer with computing. Second, OmniSci does not fully utilize CPU resources under the GPU mode. Even so, it is important to note that OmniSci could achieve much less execution time for subsequent execution when the whole dataset has been cached in the device memory. We further evaluate RAPIDS [8] with its SQL engine BlazingSQL [3], which shows a better performance than OmniSci as it overlaps PCIe transfer latency via UVM. However, a part of SQLs run by RAPIDS results in out-of-time issues (not shown in the figures). It seems that oversubscribed intermediate data in the device memory incur the bothersome page swapping between CPU and GPU sides. In contrast, the GPU-only mode of ShadowVM employs passive prefetch that not only overlaps transfer with computing but also reduces the amount of data that goes through the PCIe bus. Besides, we also observed that RAPIDS has a significant initialization overhead as UVM needs to modify the page table of CPU and GPU sides before each execution. In ShadowVM, as a benefit of decoupled data plane, this overhead could be mitigated by a memory pool that pre-allocates UVM buffers.

We also attempt to evaluate the large-scale workload (SF100). Figure 9 shows that ShadowVM in three modes has higher 6.5×, 11.3×, and 14.7× average speedups on the JVM-based Spark. This improvement comes from the amortized constant overhead for vkernel compilation and scheduling. For this large-scale workload, OmniSci degrades to the CPU execution mode due to insufficient device memory to place this large-scale dataset, which shows a similar average performance of ShadowVM with CPU-only mode. However, it is still worse than ShadowVM when its mixed mode is enabled to exploit both CPUs and GPUs. We do not evaluate RAPIDS as it leaks the CPU mode or the mixed mode to deal with insufficient device memory.

6.2 Cost Efficiency

To understand the advantages of mixed CPU-GPU mode better, we further analyze its cost efficiency. We estimate the recent price of each solution by referring to the similar Amazon EC2 instances (e.g., `r4.4xlarge` for JVM-based Spark and `g4dn.8xlarge` for OmniSci/ShadowVM). By leveraging JVM-based Spark as the normalized baseline, we define the cost efficiency as the number of executed SQLs per hour divided by the cost of each hour.

Table 1 compares the normalized cost efficiency of different solutions under SSB's SF20 and SF100. We observed that OmniSci has lower cost efficiency than ShadowVM, which is caused by two reasons. First, a portion of execution time is spent on the PCIe data transfer rather than computing. Second, it does not fully exploit both CPUs and GPUs at the same time, while the mixed CPU-GPU mode of ShadowVM

avoids wasting hardware resources to achieve the highest cost efficiency, more than 20× of Spark.

Table 1. Normalized Cost Efficiency.

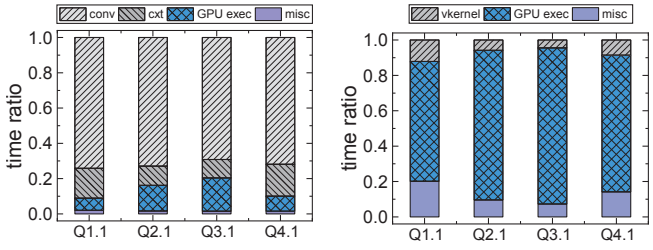
	Spark	OmniSci	ShadowVM (CPU)	ShadowVM (GPU)	ShadowVM (mixed)
SF20	1.0	7.6	9.4	13.2	**15.9**
SF100	1.0	8.6	9.5	15.9	**21.5**

6.3 The Impact of Data Plane Decoupling

To further analyze how data plane decoupling affects the performance, we carry out a breakdown experiment on ShadowVM. To avoid the interference of other factors, we reuse the identical kernel for GPU computing but compare the coupled and decoupled implementations of the data plane. We configure ShadowVM with the GPU-only mode and run four typical SQLs (i.e., Q1.1/2.1/3.1/4.1) on the SSB SF100 workload. We collect the time distribution of different parts to execute each SQL query. Figure 10 (a) and Figure 10 (b) present the breakdown results of traditional coupled and decoupled data plane, respectively.

For the case with a coupled data plane, we observed that GPU execution only occupies less than 20% of time. In contrast, most of time spends on data conversion (i.e., conv) and context initialization before GPU execution (i.e., cxt). The former stage converts tuples from RDD to columnar and aligned data, and the later stage initializes GPU context that registers the memory space of UVM to the page table of OS kernel and the GPU driver. These issues incurred by the data plane design restrict prior systems to exploit GPU potentials. During experiments, we notice an interesting fact that the GPU-only mode seems to be worse than JVM-based Spark in the lightweight Q1.1. This is because the extra overhead of coupled data plane can dominate the benefits of GPU execution under this situation.

Then when we leverage the decoupled data plane for ShadowVM, the proportion of GPU execution increases to at most 85% by solving the two above issues. Note that vkernel generation and compilation only involve minor constant overhead (about 70 milliseconds with `clang` 9.0 for CPU code and about 220 milliseconds with `nvcc` 10.0 for GPU code). Besides, miscellaneous factors (i.e., misc), such as controlling, occupy about 200 ~ 500 milliseconds. These overheads can normally be ignored under the large-scale workload. This breakdown experiment has shown the performance benefits of decoupling data plane. The following experiments will focus on optimizations under the decoupled data plane.

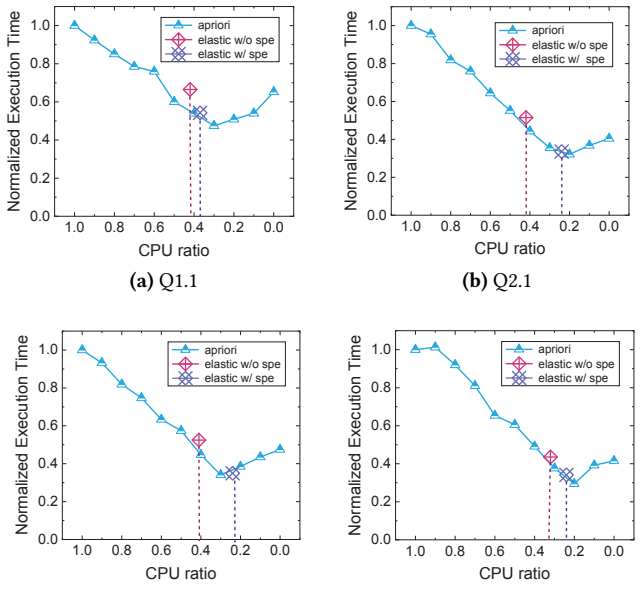


(a) Coupled Data Plane (baseline) **(b)** Decoupled Data Plane

Figure 10. The Breakdown Analysis of ShadowVM.

6.4 The Impact of Scheduling Policy

In this experiment, we analyze the impact of employed scheduling policies. We also leverage four typical SQLs from the SSB SF20 workload (i.e., Q1.1/2.1/3.1/4.1) and execute them under the mixed CPU-GPU mode with three policies, including apriori and elastic without or with speculative execution. First, we employ the apriori policy along with a varying CPU ratio (i.e., $\gamma_c n_c/(\gamma_c n_c+\gamma_g n_g)$). Figure 11 illustrates its normalized execution time that ignores the constant overhead of context initialization to focus on CPU/GPU execution.



(a) Q1.1 **(b)** Q2.1

(c) Q3.1 **(d)** Q4.1

Figure 11. Comparisons of Different Scheduling Policies.

In the mixed cases ($0.0 < \text{ratio} < 1.0$), we observed that the apriori policy with a suboptimal ratio could be worse than the GPU-only case (ratio = 0.0) due to the issue of tail latency, even if it might be better than the CPU-only case (ratio = 1.0). However, when the optimal ratio (e.g., 0.3 ~ 0.4

in Q1.1) is configured, by co-utilizing both CPUs and GPUs, the apriori policy achieves about 2.1×∼3.4× speedups over the CPU-only case and about 1.2×∼1.4× speedups over the GPU-only case. Besides, we also notice that the gap between the CPU-only case and the GPU-only case in the lightweight Q1.1 is relatively smaller than heavy-weight Q2.1/3.1/4.1, which is caused by Q1.1's lower computing density. As a result, its optimal ratio is closer to 0.5, which implies that the CPU side and GPU side have nearly equivalent performance.

After that, we further evaluate the elastic policy that does not know the optimal CPU ratio in advance. Figure 11 shows the performance of elastic policy along with its stayed CPU ratio. When speculative execution is disabled (i.e., elastic w/o spe), it could only find an inaccurate ratio that shows insignificant speedup over (or even worse than) the GPU-only case. Besides, we also observed that the elastic polity also involves 7% ∼ 15% scheduling overhead compared with the apriori policy. After speculative execution is enabled, there are about 5% ∼ 10% local partitions re-executed by the GPU side. By alleviating the tail lag incurred by bothersome stragglers, the elastic policy shows a similar performance to the apriori policy with the optimal ratio in the end.

6.5 The Impact of Passive Prefetch

In the next experiment, we analyze how to alleviate the limitation of PCIe transfer when processing massive data with GPU. We compare active copy and passive prefetch for GPU execution. We also select the four SSB SQLs in the previous experiments and execute them by the GPU-only mode of ShadowVM.

We illustrate the normalized execution time of each SQL in Figure 12. For active copy, we attempt to employ HyperQ to overlap PCIe transfer with GPU computing, in which 1 to 4 CUDA streams are involved to execute buffer copy and kernel launch in an interleaved manner. Compared with the unoverlapped version (i.e., active copy(1)), passive prefetch reduces about 33% ∼ 72% of execution time. Even though HyperQ could improve its performance by overlapping transfer with computing (i.e., active copy(2/4)), it is still worse than passive prefetch. When more than four CUDA streams are involved, we observed that the promotion becomes minor. It seems that the promotion of overlapping is restricted by the upper bound of PCIe bandwidth, while passive prefetch breaks this restriction and achieves higher throughput by saving PCIe bandwidth.

To validate our analysis, we further leverage the nvvp tool to measure the amount of data for PCIe transfer between CPU and GPU side during runtime. As shown in Figure 13, intuitively, HyperQ could overlap transfer with computing but do not change the amount of data migrated to the device memory. Thus, the promotion of overlapping is still restricted by the upper bound of PCIe bandwidth due to the barrel effect when computing spends less time than data transfer. In contrast, passive prefetch not only enables overlapping

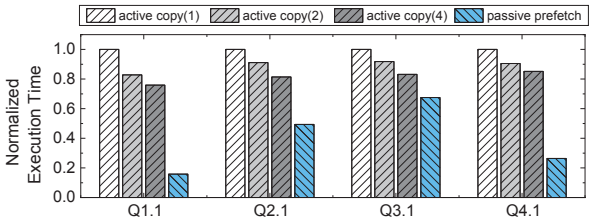


Figure 12. Active Copy vs Passive Prefetch (Execution Time).

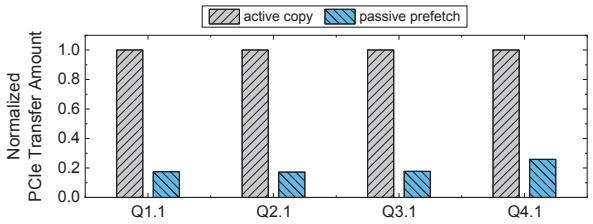


Figure 13. Active Copy vs Passive Prefetch (PCIe Transfer).

but also saves more than 70% of PCIe transfer bandwidth as it skips to copy unused data.

6.6 Case Study: On-line Advertisement Analytics

In previous experiments, we mainly evaluate off-line analytics. Considering the case of on-line analytics, we adopt YSB [12] to further evaluate the potential of ShadowVM. Unlike off-line analytics, on-line analytics needs to process continuous advertisement data within each fine-grained window rather than the whole dataset. Similar to Spark, we also regard the JVM-based Flink as the control plane, and leverage ShadowVM as the data plane for computing. Flink collects the upstream data and sends action commands to invoke computing in ShadowVM when collected data reaches the window size. Flink and ShadowVM exchange the input and output data of each window through shared memory that minimizes the communication overhead.

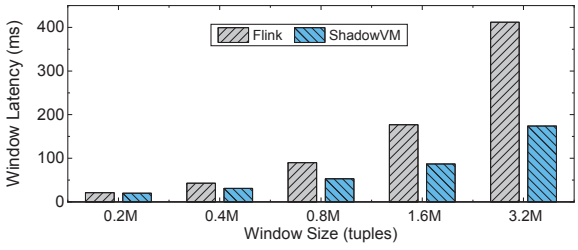


Figure 14. The Computing Latency per Window.

We leverage the CPU-only mode for ShadowVM because CPU is more appropriate for on-line analytics than GPU that is optimized for off-line analytics. Figure 14 illustrates the computing latency of each window under the varying

window size (0.2 to 3.2 millions tuples per window). The case to use ShadowVM as the data plane could reduce up to 58% of latency compared with the JVM-based Flink. This is because bare metal CPU is more efficient than the managed JVM that incurs more virtual procedure calls. Besides, we also observed that the gap is more significant under the larger widow size as the constant overhead to invoke action commands can be migrated by using a longer interval between windows.

7 Related Work

Considering the rapid development of hardware, exploring accelerators for data analytics is still an open issue. In this section, we briefly introduce and conclude the relevant works.

Today, many works attempt to exploit hardware acceleration in big data analytics. Flare [25] translates the whole spark job as a lightweight OpenMP program. HeteroSpark [37] and [43] use RPC to process RDD data with remote GPUs. To improve this, Spark-GPU [58] and SWAT [29] use JNI-based methods to process RDD data with local GPUs. G-Storm [21] and GFlink [19] enable GPU acceleration within streaming systems, where GPU processes each data bulk through JNI. GPUEnabler [15] studies how to avoid redundant intermediate data transfer when using Spark with GPU. Similarly, [20] also harnesses Spark with FPGA in a similar JNI-based method. Essentially, these works are restricted by the coupled data plane, which may incur costly initialization, serialization, and transfer overhead before data is migrated to accelerators. More importantly, these works are based on case-by-case solutions, which are hard to extend for various scenarios. By decoupling the data plane from Spark's RDD, ShadowVM bypasses JVM to provides a bare metal environment for GPUs (or other feasible accelerators) and could also leverage different frameworks as its control plane.

Enabling accelerators in JVM is a relevant field. TornadoVM [26] is inspired by OpenJDK's HotSpotVM but compiles JVM bytecodes into the OpenCL kernel at runtime, which could run on a selected type of device (e.g., CPU or GPU or FPGA). Sumatra [10], J9 [6], and Aparapi [2] are similar to TornadoVM but do not support dynamic recompilation. These works can be classified as language virtualization, while ShadowVM is more closed to system-level virtualization, which employs action shadowing to run computations on a pod of CPUs and GPUs. The contributions of ShadowVM are orthogonal to these works. For instance, language virtualization could help ShadowVM to support Java UDFs, which will be converted into the inline functions before compiling vkernel for execution.

GPU has presented its potentials in analytical databases that executes SQL primitives with GPU kernels [52]. Hawk [17] studies how to translate the SQL query into CPU code or GPU code. OmniSci [47] is a typical GPU database that executes SQLs with GPU kernels. With UVA (a previous version of UVM), [34] attempts to overlap transfer in the individual join. XeFlow [38] leverages UVM-based index structures to reduce PCIe transfer in the filter. RAPIDS [8] is a recent analytical platform with GPU, which handles overlapping and memory oversubscription with UVM. Besides, [23, 30] offload a part of actions in an SQL to the GPU side. [28] achieves mixed CPU-GPU execution in fine-grained database primitives and optimizes the mixed ratio via memory bandwidth modeling. Previous works inspire many considerations of this paper. One difference between GPU databases and big data systems with GPU is that the former mainly aims at absolute performance on a GPU node, while the latter also considers other factors, such as scalability, extensibility, and cost efficiency. This is why GPU databases tend to use the GPU-only mode and active copy, while ShadowVM employs the mixed mode and passive prefetch.

GPU scheduling is another related research topic. [22, 39, 40] employ mixed CPU-GPU execution in specific applications, including high-performance computing. Recently, Spark 3.0 [1] and other GPU-aware schedulers [46, 53, 55] are mainly optimized for ML/DL workloads rather than big data analytics. In contrast, since mainstream big data systems are still CPU-centric, many schedulers focus on CPU resources [27, 56]. The former assign GPUs for long-running training, while the latter finishes each analytical job within minutes and should take the CPU-GPU architecture into account. When GPU is becoming practical in big data systems, it is urgent to upgrade current strategies. RAPIDS [8] enables overlapped scheduling on the GPU but does not co-utilize CPU resources. How to exploit CPU/GPU resources in big data workloads is still an open issue.

8 Conclusion

This paper takes a deep dive into bringing bare metal accelerators into large-scale data analytics. To bypass the restrictions incurred by the coupled architecture, we introduce action shadowing that decouples the data plane from the managed JVM runtime. To improve the cost efficiency, the data plane fully takes advantage of bare metal accelerators by exploiting the mixed CPU-GPU execution via vkernel and employs passive prefetch to break the bottleneck of PCIe bus for GPU execution. Our main ideas could be easily extended to support other host systems and accelerators that face similar issues. This work inspires a new route to build accelerator-based systems for data processing.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by National Key Research and Development Program of China (No. 2018YFB1003400), and National Natural Science Foundation of China (No. 61772204 and 61732014). Chuliang Weng (clweng@dase.ecnu.edu.cn) is the corresponding author.

References

[1] 2020. Accelerator-aware task scheduling for Spark. https://issues.apache.org/jira/browse/SPARK-24615/.

[2] 2020. Aparapi. http://aparapi.github.io/.

[3] 2020. BlazingSQL. https://blazingsql.com/.

[4] 2020. GRPC: A high-performance, open-source universal RPC framework. https://grpc.io/.

[5] 2020. Introducing AresDB: Uber's GPU-Powered Open Source, Real-time Analytics Engine. https://eng.uber.com/aresdb/.

[6] 2020. J9 Virtual Machine (JVM). https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/user/java_jvm.html.

[7] 2020. Protobuf: A language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers/.

[8] 2020. RAPIDS. https://rapids.ai.

[9] 2020. Spark 3.0. http://spark.apache.org/news/spark-3.0.0-preview.html.

[10] 2020. Sumatra. http://openjdk.java.net/projects/sumatra/.

[11] 2020. TensorFlow Serving. https://github.com/tensorflow/serving.

[12] 2020. Yahoo Streaming Benchmarks. https://github.com/yahoo/streaming-benchmarks.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2016*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[14] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of International Conference on Management of Data (SIGMOD) 2015*. 1383–1394. https://doi.org/10.1145/2723372.2742797

[15] Ryo Asai, Masao Okita, Fumihiko Ino, and Kenichi Hagihara. 2018. Transparent Avoidance of Redundant Data Transfer on GPU-enabled Apache Spark. In *Proceedings of 11th Workshop on General Purpose Processing using GPUs 2018*. 22–30. https://doi.org/10.1145/3180270.3180276

[16] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2014*. 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[17] Sebastian Breβ, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal* 27, 6 (2018), 797–822. https://doi.org/10.1007/s00778-018-0512-y

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015). http://sites.computer.org/debull/A15dec/p28.pdf

[19] Cen Chen, Kenli Li, Aijia Ouyang, Zeng Zeng, and Keqin Li. 2018. GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data. *IEEE Transactions on Parallel and Distributed Systems* 29, 6 (2018), 1275–1288. https://doi.org/10.1109/TPDS.2018.2794343

[20] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration. In *Proceedings of IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2016*. 29. https://doi.org/10.1109/FCCM.2016.18

[21] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, and Charles Kamhoua. 2015. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *IEEE International Conference on Big Data (Big Data) 2015*. 307–312. https://doi.org/10.1109/BigData.2015.7363769

[22] JeeWhan Choi, Aparna Chandramowlishwaran, Kamesh Madduri, and Richard W. Vuduc. 2014. A CPU:GPU Hybrid Implementation and Model-Driven Scheduling of the Fast Multipole Method. In *Proceedings of Seventh Workshop on General Purpose Processing Using GPUs 2014*. 64. https://dl.acm.org/citation.cfm?id=2576787

[23] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. http://www.vldb.org/pvldb/vol12/p544-chrysogelos.pdf

[24] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113. http://doi.acm.org/10.1145/1327452.1327492

[25] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2018*. 799–815. https://www.usenix.org/conference/osdi18/presentation/essertel

[26] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of International Conference on Virtual Execution Environments (VEE) 2019*. 165–178. https://doi.org/10.1145/3313808.3313819

[27] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2016*. 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[28] Michael Gowanlock, Ben Karsin, Zane Fink, and Jordan Wright. 2019. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In *Proceedings of International Workshop on Data Management on New Hardware (DaMoN) 2019*. 7:1–7:11. https://doi.org/10.1145/3329785.3329926

[29] Max Grossman and Vivek Sarkar. 2016. SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform. In *Proceedings of International Symposium on High-Performance Parallel and Distributed Computin (HPDC) 2016*. 81–92. https://doi.org/10.1145/2907294.2907307

[30] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (2013), 709–720. http://www.vldb.org/pvldb/vol6/p709-heimel.pdf

[31] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2011*. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[32] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of ACM international conference on Multimedia (MM) 2014*. 675–678. https://doi.org/10.1145/2647868.2654889

[33] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. 2020. Overlapping host-to-device copy and computation using hidden unified memory. In *Proceedings of Symposium on Principles*

and Practice of Parallel Programming (PPoPP) 2020. 321–335. https://doi.org/10.1145/3332466.3374531

[34] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *Proceedings of International Workshop on Data Management on New Hardware (DaMoN) 2012*. 55–62. https://doi.org/10.1145/2236584.2236592

[35] Seon Wook Kim, Chong-liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. 2001. Reference idempotency analysis: a framework for optimizing speculative execution. In *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP) 2001*. 2–11. https://doi.org/10.1145/379539.379547

[36] Hao Li, Di Yu, Anand Kumar, and Yi-Cheng Tu. 2014. Performance modeling in CUDA streams – A means for high-throughput data processing. In *Proceedings of International Conference on Big Data 2014*. 301–310. https://doi.org/10.1109/BigData.2014.7004245

[37] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. 2015. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. In *Proceedings of IEEE International Conference on Networking, Architecture and Storage (NAS) 2015*. 347–348. https://doi.org/10.1109/NAS.2015.7255222

[38] Zhifang Li, Beicheng Peng, and Chuliang Weng. 2020. XeFlow: Streamlining Inter-Processor Pipeline Execution for the Discrete CPU-GPU Platform. *IEEE Trans. Computers* 69, 6 (2020), 819–831. https://doi.org/10.1109/TC.2020.2968302

[39] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP) 2016*. 41:1–41:2. https://doi.org/10.1145/2851141.2851174

[40] Yang Liu, Jianguo Wang, and Steven Swanson. 2018. Griffin: uniting CPU and GPU in information retrieval systems for intra-query parallelism. In *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP) 2018*. 327–337. https://doi.org/10.1145/3178487.3178512

[41] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2006. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst.* 24, 4 (2006), 361–392. https://doi.org/10.1145/1189256.1189258

[42] NVIDIA. 2020. CUDA C programming guide. (2020). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[43] Yasuhiro Ohno, Shin Morishima, and Hiroki Matsutani. 2016. Accelerating Spark RDD Operations with Local and Remote GPU Devices. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS) 2016*. 791–799. https://doi.org/10.1109/ICPADS.2016.0108

[44] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50. https://www.cs.umb.edu/~xuedchen/research/publications/StarSchemaB.PDF

[45] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015. http://www.vldb.org/pvldb/vol11/p1002-palkar.pdf

[46] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of EuroSys Conference (EuroSys) 2018*. 3:1–3:14. https://doi.org/10.1145/3190508.3190517

[47] Christopher Root and Todd Mostak. 2016. MapD: a GPU-powered big data analytics and visualization platform. In *Proceedings of Special Interest Group on Computer Graphics and Interactive Techniques Conference 2016*. 73:1–73:2. https://doi.org/10.1145/2897839.2927468

[48] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 2013*. 49–68. https://doi.org/10.1145/2517349.2522715

[49] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of International Conference on Management of Data (SIGMOD) 2020*. 1617–1632. https://doi.org/10.1145/3318464.3380595

[50] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 2019*. 322–337. https://doi.org/10.1145/3341301.3359658

[51] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of ACM Symposium on Cloud Computing (SOCC) 2013*. 5:1–5:16. https://doi.org/10.1145/2523616.2523633

[52] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of International Symposium on Microarchitecture (MICRO) 2012*. 107–118. https://doi.org/10.1109/MICRO.2012.19

[53] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2018*. 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[54] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Proceedings of European Conference on Parallel Processing*. Springer, 887–899. https://doi.org/10.1007/978-3-642-03869-3_82

[55] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. 2020. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC) 2020*. 173–184. https://doi.org/10.1145/3369583.3392679

[56] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2008*. 1–14. https://www.usenix.org/legacy/events/osdi08/tech/full_papers/yu_y/yu_y.pdf

[57] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. http://www.vldb.org/pvldb/vol6/p817-yuan.pdf

[58] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *Proceedings of IEEE International Conference on Big Data (BigData) 2016*. 273–283. https://doi.org/10.1109/BigData.2016.7840613

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2012*. 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[60] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets