FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures

Feng Zhang[1], Lin Yang[1], Shuhao Zhang[2,3], Bingsheng He[3], Wei Lu[1], Xiaoyong Du[1]
[1]*Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China*
[2]*DIMA, Technische Universität Berlin*
[3]*School of Computing, National University of Singapore*
fengzhang@ruc.edu.cn, yanglin2330@ruc.edu.cn, shuhao.zhang@tu-berlin.de, hebs@comp.nus.edu.sg, lu-wei@ruc.edu.cn, duyong@ruc.edu.cn

Abstract

Accelerating SQL queries on stream processing by utilizing heterogeneous coprocessors, such as GPUs, has shown to be an effective approach. Most works show that heterogeneous coprocessors bring significant performance improvement because of their high parallelism and computation capacity. However, the discrete memory architectures with relatively low PCI-e bandwidth and high latency have dragged down the benefits of heterogeneous coprocessors. Recently, hardware vendors propose CPU-GPU integrated architectures that integrate CPU and GPU on the same chip. This integration provides new opportunities for fine-grained cooperation between CPU and GPU for optimizing SQL queries on stream processing. In this paper, we propose a data stream system, called FineStream, for efficient window-based stream processing on integrated architectures. Particularly, FineStream performs fine-grained workload scheduling between CPU and GPU to take advantage of both architectures, and it also provides efficient mechanism for handling dynamic stream queries. Our experimental results show that 1) on integrated architectures, FineStream achieves an average 52% throughput improvement and 36% lower latency over the state-of-the-art stream processing engine; 2) compared to the stream processing engine on the discrete architecture, FineStream on the integrated architecture achieves 10.4x price-throughput ratio, 1.8x energy efficiency, and can enjoy lower latency benefits.

1 Introduction

Optimizing the performance of stream processing systems has been a hot research topic due to the rigid requirement on the event processing latency and throughput. Stream processing on GPUs has been shown to be an effective method to improve its performance [23, 33, 34, 41, 54, 62, 67]. GPUs consist of a large amount of lightweight computing cores, which are naturally suitable for data-parallel stream processing. GPUs are often used as coprocessors that are connected to CPUs through PCI-e [42]. Under such discrete architectures, stream data need to be copied from the main memory to GPU memory via PCI-e before GPU processing, but the low bandwidth of PCI-e limits the performance of stream processing on GPUs. Hence, stream processing on GPUs needs to be carefully designed to hide the PCI-e overhead. For example, prior works have explored pipelining the computation and communication to hide the PCI-e transmission cost [34, 54].

Despite of various studies in previous stream processing engines on general-purpose applications [5, 23, 25, 33, 54, 62], relatively few studies focus on SQL-based relational stream processing. Supporting relational stream processing involves additional complexities, such as how to support window-based query semantics and how to utilize the parallelism with a small window or slide size efficiently. Existing engines, such as Spark Streaming [57], struggle to support small window and slide sizes, while the state-of-the-art window-based query engine, Saber [34], adopts a *bulk-synchronous parallel model* [66] for hiding PCI-e transmission overhead.

In recent years, hardware vendors have released integrated architectures, which completely remove PCI-e overhead. We have seen CPU-GPU integrated architectures such as NVIDIA Denver [13], AMD Kaveri [15], and Intel Skylake [27]. They fuse CPUs and GPUs on the same chip, and let both CPUs and GPUs share the same memory, thus avoiding the PCI-e data transmission. Such integration poses new opportunities for window-based streaming SQL queries from both hardware and software perspectives.

First, different from the separate memory hierarchy of discrete CPU-GPU architectures, the integrated architectures provide unified physical memory. The input stream data can be processed in the same memory for both CPUs and GPUs, which eliminates the data transmission between two memory hierarchies, thus eliminating the data copy via PCI-e.

Second, the integrated architecture makes it possible for processing dynamic relational workloads via fine-grained cooperations between CPUs and GPUs. A streaming query can consist of multiple operators with varying performance features on different processors. Furthermore, stream processing often involves dynamic input workload, which affects operator performance behaviors as well. We can place operators on different devices with proper workloads in a fine-grained man-

ner, without worrying about transmission overhead between CPUs and GPUs.

Based on the above analysis, we argue that stream processing on integrated architectures can have much more desirable properties than that on discrete CPU-GPU architectures. To fully exploit the benefits of integrated architectures for stream processing, we propose a fine-grained stream processing framework, called FineStream. Specifically, we propose the following key techniques. First, a performance model is proposed considering both operator topologies and different architecture characteristics of integrated architectures. Second, a light-weight scheduler is developed to efficiently assign the operators of a query plan to different processors. Third, online profiling with computing resource and topology adjustment are involved for dynamic workloads.

We evaluate FineStream on two platforms, AMD A10-7850K, and Ryzen 5 2400G. Experiments show that FineStream achieves 52% throughput improvement and 36% lower latency over the state-of-the-art CPU-GPU stream processing engine on the integrated architecture. Compared to the best single processor throughput, it achieves 88% performance improvement.

We also compare stream processing on integrated architectures with that on discrete CPU-GPU architectures. Our evaluation shows that FineStream on integrated architectures achieves 10.4x price-throughput ratio, and 1.8x energy efficiency. Under certain circumstances, it is able to achieve lower processing latency, compared to the state-of-the-art execution on discrete architectures. This further validates the large potential of exploring the integrated architectures for data stream processing.

Overall, we make the following contributions:

1) We propose the first fine-grained window-based relational stream processing framework that takes the advantages of the special features of integrated architectures.

2) We develop lightweight query plan adaptations for handling dynamic workloads with the performance model that considers both the operator and architecture characteristics.

3) We evaluate FineStream on a set of stream queries to demonstrate the performance benefits over current approaches.

2 Background

2.1 Integrated Architecture

We show an architectural overview of the CPU-GPU integrated architecture in Figure 1. The integrated architecture consists of a CPU, a GPU, a shared memory management unit, and system DRAM. CPUs and GPUs have their own caches. Some models of integrated architectures, such as Intel Haswell i7-4770R processor [3], integrate a shared last level cache for both CPUs and GPUs. The shared memory management unit is responsible for scheduling accesses to system DRAM by different devices. Compared to the discrete CPU-GPU architecture, both CPUs and GPUs are integrated on the same chip. The most attractive feature of such integration is the shared main memory which is available to both devices. With the shared main memory, CPUs and GPUs can have more opportunities for fine-grained cooperation. The most commonly used programming model for integrated architectures is OpenCL [49], which regards the CPU and the GPU as *devices*. Each device consists of several *compute units* (CUs), which are the CPU and GPU cores in Figure 1.

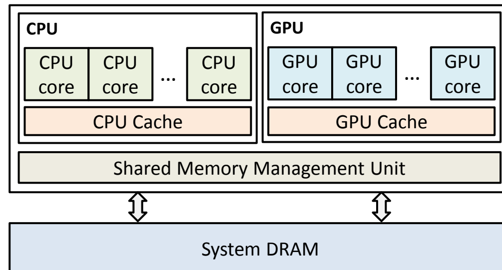

Figure 1: A general overview of the integrated architecture.

We show a comparison between the integrated and discrete architectures (discrete GPUs) in Table 1. These architectures are used in our evaluation (Section 7). Although the integrated architectures have lower computation capacity than the discrete architectures currently, the integrated architecture is a potential trend for a future generation of processors. Hardware vendors, including AMD [15], Intel [27] and NVIDIA [13], all release their integrated architectures. Moreover, future integrated architectures can be much more powerful, even can be a competitive building block for exascale HPC systems [47, 55], and the insights and methods in this paper still can be applied. Besides, the integrated architectures are attractive due to their efficient power consumption [15, 60] and low price [31].

Table 1: Integrated architectures vs. discrete architectures.

	Integrated Architectures		Discrete Architectures	
Architecture	A10-7850K	Ryzen5 2400G	GTX 1080Ti	V100
# cores	512+4	704+4	3584	5120
TFLOPS	0.9	1.7	11.3	14.1
bandwidth (GB/s)	25.6	38.4	484.4	900
price ($)	209	169	1100	8999
TDP (W)	95	65	250	300

The number of cores for each integrated architecture includes four CPU cores. For discrete architectures, we only show the GPU device.

2.2 Stream Processing with SQL

Although various heterogeneous stream processing systems have appeared [23, 33, 34, 41, 54, 62, 67], we find that most of these systems are used to process unstructured data, and only one work, Saber [34], is developed for structured stream processing on GPUs. Saber supports structured query language (SQL) on stream data [6]. The benefits of supporting SQL come from two aspects. First, with SQL, users can use familiar SQL commands to access the required records, which makes the system easy to use. Second, supporting SQL eliminates the tedious programming operations about how to reach a required record, which greatly expands the flexibility of its usage. Based on the analysis, this work explores stream processing with SQL on integrated architectures.

We consider supporting the basic SQL functions with stream processing, as shown in Figure 2. According to [6], SQL on stream processing consists of the following four major concepts: 1) **Data stream** S, which is a sequence of tuples, $< t_1, t_2, ... >$, where t_i is a tuple. A tuple is a finite ordered list of elements, and each tuple has a timestamp. 2) **Window** w, which refers to a finite sequence of tuples, which is the data unit to be processed in a query. The window in stream has two features: *window size* and *window slide*. *Window size* represents the size of the data unit to be processed, and *window slide* denotes the sliding distance between two adjacent windows. 3) **Operators**, which are the minimum processing units for the data in a window. In this work, we support common relational operators including *projection*, *selection*, *aggregation*, *group-by*, and *join*. 4) **Queries**, which are a form of data processing, each of which consists of at least one operator and is based on windows. Additionally, note that in real stream processing systems such as Saber [34], data are processed in *batch* granularity, instead of window granularity. A batch can be a group of windows when the window size is small, or a part of a window when the window size is extremely large.

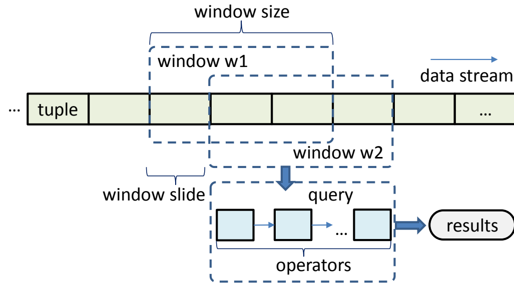

Figure 2: Stream processing with SQL.

3 Revisiting Stream Processing

We discuss the new opportunities (Section 3.1) and challenges (Section 3.2) for stream processing on integrated architectures in this section, which motivate this work.

3.1 Varying Operator-Device Preference

Opportunities: Due to the elimination of transmission cost between CPU and GPU devices on integrated architectures, we can assign operators to CPU and GPU devices in a fine-grained manner according to their device-preference.

We analyze the operators in a query, and find that different operators show various device preferences on integrated architectures. Some operators achieve higher performance on the CPU device, and others have higher performance on the GPU device. We use a simple query of *group-by* and *aggregation* on the integrated architecture for illustration, as shown in Figure 3. The GPU queue is used to sequentially execute the queries on the GPU, while the CPU queue is used to execute the related queries on the CPU. The window size is 256 tuples and the window slide is 64. Each batch contains 64,000 tuples, and each tuple is 32 bytes. The input data are synthetically generated, which is described in Section 7.1. When the query runs on the CPU, *group-by* takes about 18.2 ms and *aggregation* takes about 5.2 ms. However, when the query runs on the GPU, *group-by* takes about 6.7 ms and *aggregation* takes about 5.8 ms.

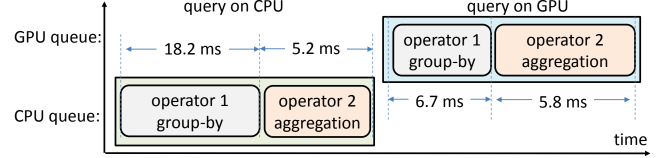

Figure 3: An example of operator-device preference.

We further evaluate the performance of operators on a single device in Table 2. Table 2 shows that using a single type of device *cannot* achieve the optimal performance for all operators. The *aggregation* includes the operators of *sum*, *count*, and *average*, and they have similar performance. We use *sum* as a representative for *aggregation*. From Table 2, we can see that *projection*, *selection*, and *group-by* achieve better performance on the GPU than on the CPU, while *aggregation* and *join* achieve better performance on the CPU than on the GPU. Additionally, *projection* shows similar performance on CPU and GPU devices. Specifically, for *join*, the CPU performance is about 6x the GPU performance. Such different device preferences inspire us to perform fine-grained stream processing on integrated architectures.

Table 2: Performance (tuples/s) of operators on the CPU and the GPU of the integrated architecture.

Operator	CPU only (10^6)	GPU only (10^6)	Device choice
projection	14.2	**14.3**	GPU
selection	13.1	**14.1**	GPU
aggregation	**14.7**	13.5	CPU
group-by	8.1	**12.4**	GPU
join	**0.7**	0.1	CPU

Integrated architectures eliminate data transmission cost between CPU and GPU devices. This provides opportunities for stream processing with operator-level fine-grained placement. The operators that can fully utilize the GPU capacity exhibit higher performance on GPUs than on CPUs, so these operators shall be executed on GPUs. In contrast, the operators with low parallelism shall be executed on CPUs. Please note that such fine-grained cooperations is inefficient on discrete CPU-GPU architectures due to transmission overhead. For example, Saber [34], one of the state-of-the-art stream processing engines utilizing the discrete CPU-GPU architectures, is designed aiming to hide PCI-e overhead. It adopts a bulk-synchronous parallel model, where all operators of a query are scheduled to one processor to process a micro-batch of data [53].

3.2 Fine-Grained Stream Processing

Challenges: A fine-grained stream processing that considers both architecture characteristics and operator preference shall have better performance, but this involves several challenges, from both application and architecture perspectives.

Based on the analysis, we argue that stream processing on integrated architectures can have much desirable properties

than that on discrete CPU-GPU architectures. Particularly, this work introduces a concept of fine-grained stream processing: co-running the operators to utilize the shared memory on integrated architectures, and dispatching the operators on devices with both architecture characteristics and operator features considered.

However, enabling fine-grained stream processing on integrated architectures is complicated by the features of SQL stream processing and integrated architectures. We summarize three major challenges as follows.

Challenge 1: Application topology combined with architectural characteristics. Application topology in stream processing refers to the organization and execution order of the operators in a SQL query. First, the relation among operators could be more complicated in practice. The operators may be represented as a directed acyclic graph (DAG), instead of a chain, which contains more parallel acceleration opportunities. Second, with architectural characteristics considered, such as the CPU and GPU architectural differences, the topology with computing resource distribution becomes very complex. In such situations, how to perform fine-grained operator placement for application topology on different devices of integrated architectures becomes a challenge. Third, to assist effective scheduling decisions, a performance model is needed to predict the benefits from various perspectives.

Challenge 2: SQL query plan optimization with shared main memory. First, a SQL query in stream processing can consist of many operators, and the execution plan of these operators may cause different bandwidth pressures and device preferences. Second, in many cases, independent operators may not consume all the memory bandwidth, but co-running them together could exceed the bandwidth limit. We need to analyze the memory bandwidth requirement of co-running. Third, CPUs and GPUs have different preferred memory access patterns. Current methods [5,23,25,33,34,54,62] do not consider these complex situations of shared main memory in integrated architectures.

Challenge 3: Adjustment for dynamic workload. During stream processing, stream data are changing dynamically in distributions and arrival speeds, which is challenging to adapt. First, workload change detection and computing resource adjustment need to be done in a lightweight manner, and they are critical to performance. Second, the query plan may also need to be updated adaptively, because the operator placement strategy based on the initial state may not be suitable when the workload changes. Third, during adaptation, online stream processing needs to be served efficiently. Resource adjustment and query plan adaptation on the fly may incur runtime overhead, because we need to adjust not only the operators in the DAG but also the hardware computing resources to each operator. Additionally, the adjustment among different streams also needs to be considered.

4 FineStream Overview

We propose a framework, called FineStream, for fine-grained stream processing on integrated architectures. The overview of FineStream is shown in Figure 4. FineStream consists of three major components, including performance model, online profiling, and dispatcher. The online profiling module is used to analyze input batches and queries for useful information, which is then fed into the performance model. The performance model module uses the collected data to build models for queries with both CPUs and GPUs to assist operator dispatching. The dispatcher module assigns stream data to operators with proper processors according to the performance model on the fly.

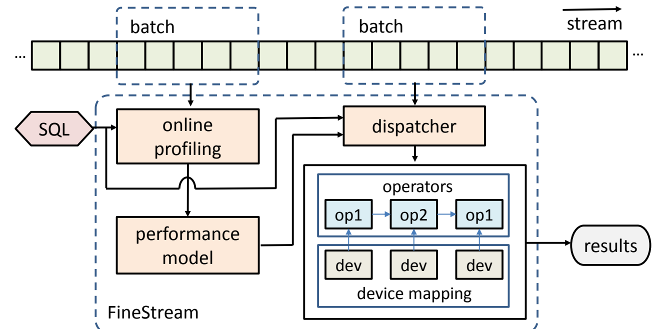

Figure 4: FineStream overview.

Next, we discuss the ideas in FineStream, including its workflow, query plan generation, processing granularity, operator mapping, and solutions to the challenges mentioned in Section 3.2.

Workflow. The workflow of FineStream is as follows. When the engine starts, it first processes several batches using only the CPUs or the GPUs to gather useful data. Second, based on these data, it builds a performance model for operators of a query on different devices. Third, after the performance model is built, the dispatcher starts to work, and the fine-grained stream processing begins. Each operator shall be assigned to the cores of the CPU or the GPU for parallel execution. Additionally, the workload could be dynamic. For dynamic workload, query plan adjustment and resource reallocation need to be conducted.

Topology. The query plan can be represented as a DAG. In this paper, we concentrate on relational queries. We show an example in Figure 5, where OP_i represents an operator. *OP7* and *OP11* can represent *joins*. We follow the terminology in compiler domain [52], and call the operators from the beginning or the operator after *join* to the operator that merges with another operator as a *branch*. Hence, the query plan is also a branch DAG. For example, the operators of *OP1*, *OP2*, and *OP3* form a branch in Figure 5. The main reason we use the branch concept is for parallelism: operators within a branch can be evaluated in a pipeline, and different branches can be executed in parallel, which shall be detailed in Section 5. The execution time in processing one batch is equal to the traversal time from the beginning to the end of the DAG. Be-

cause branches can be processed in parallel, the *branch* with the longest execution time dominates the execution time. We call the operator path that determines the total execution time as $path_{critical}$, so the branch with the longest execution time belongs to $path_{critical}$. For example, we assume that *branch2* has the longest execution time among the branches, its time is $t_{branch2}$, and the execution time for OP_i is t_{OP_i}. *OP7* and *OP11* can also be regarded as branches. Only when the outcomes of *OP3* and *OP6* are available, then *OP7* can proceed. So do to the operators of *OP7* and *OP10* to *OP11*. Assuming OP7 and OP11 are blocking join operators, the total execution time for this query is the sum of $t_{branch2}$, t_{OP7}, and t_{OP11}.

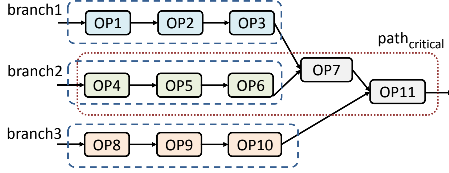

Figure 5: An example of query operators in DAG representation, where OP_i represents an operator.

Operator Mapping. The fine-grained scheduling lies in how to map the operators to the limited resources on integrated architectures. In FineStream, we allow an operator to use one or several cores of the CPU or the GPU device. When the platform cannot provide enough resources for all the operators, some operators may share the same compute units. For example, in Figure 5, *OP1* and *OP2* can share the same CPU cores. If so, the input batches sequentially goes through *OP1* and *OP2* and no pipeline exists between two batches for *OP1* and *OP2*.

Solutions to Challenges. FineStream addresses all the challenges mentioned in Section 3.2. For the first challenge, the performance model module estimates the overall performance with the help of the online profiling module by sampling on a small number of batches, and the dispatcher dynamically puts the operators on the preferred devices. For the second challenge, we have considered the bandwidth factor when building the performance model, which can be used to guide the parallelism for operators with limited bandwidth considered. For the third challenge, the online profiling checks both the stream and the operators to measure the data ingestion rate, and FineStream responses to these situations with different strategies based on the analysis for dynamic workloads. Next, we show the details of our system design.

5 Model for Parallelism Utilization

Guideline: A performance model is necessary for operator placement in FineStream, especially for the complicated operator relations in the DAG structure. The overhead of building fine-grained performance model for a query is limited because the placement strategy from the model can be reused for the continuous stream data.

We model the performance of executing a query in this section. The operators of the input query are organized as a DAG. In the performance model, we consider two kinds of parallelism. First, for intra-batch parallelism, we consider *branch co-running*, which means co-running operators in processing one batch. Second, for inter-batch parallelism, we consider *batch pipeline*, which means processing different batches in pipelines.

5.1 Branch Co-Running

Independent branches can be executed in parallel. With limited computation resources and bandwidth, we build a model for branch co-running behaviors in this part. We use *Bmax* to denote the maximum bandwidth the platform can provide. If the sum of bandwidth utilization from different parallel branches, *Bsum*, exceeds *Bmax*, we assume that the throughput degrades proportionally to the *Bmax*/*Bsum* of the throughput with enough bandwidth [60]. To measure the bandwidth utilization, generally, for n co-running tasks, we have n co-running stages, because tasks complete one by one. When multiple tasks finish at the same time, the number of stages decreases accordingly.

We use the example in Figure 5 for illustration. Assume that the time for different *branches* is shown in Figure 6 (a). If we co-run the three branches simultaneously, then the execution can be partitioned into three stages with different overlapping situations. We use t_{stage1}, t_{stage2}, and t_{stage3} to represent the related stage time when the system has enough bandwidth. Then, if the required bandwidth for $stage_i$ exceeds *Bmax*, the related real execution time t_{stage_i}' also extends accordingly. We define t_{stage_i}' in Equation 1. When the platform can provide the required bandwidth, r_i is equal to one. Otherwise, r_i is the ratio of the required bandwidth divided by *Bmax*.

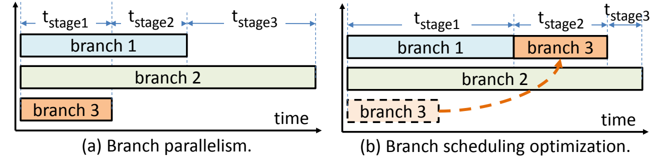

Figure 6: An example of branch parallelism and optimization.

$$t_{stage_i}{}' = r_i \cdot t_{stage_i} \tag{1}$$

To estimate the time for processing a batch in the critical path, generally for the branch DAG, we perform topology sort to organize the branches into different layers, and then we co-run branches on layer granularity. In each layer, we perform the above branch co-running. Then, the total execution time is the sum of time of all layers, as shown in Equation 2.

$$t_{total} = \sum_{j=0}^{n_{layer}} \sum_{i=0}^{n_{stage}} t_{stage_i,layer_j}{}' \tag{2}$$

The throughput is the number of tuples divided by the execution time. Assume the number of tuples in a batch is m, then, the throughput is shown in Equation 3.

$$throughput_{branchCoRun} = \frac{m}{t_{total}} \tag{3}$$

Optimization. We can perform branch scheduling for optimization, which has two major benefits. First, by moving

branches from the stage with fully occupied bandwidth utilization to the stage with surplus bandwidth, the bandwidth can be better utilized. For example, in Figure 6 (b), assume that in $stage_1$, the required bandwidth exceeds *Bmax*, but the sum of the required bandwidth of *branch*2 and *branch*3 is lower than *Bmax*, then we can move the execution of *branch3* after the execution of *branch1* for better bandwidth utilization. Second, the system may not have enough computation resources for all branches so that we can reschedule branches for better computation resource utilization. In *stage1* of Figure 6 (a), when the platform cannot provide enough computing resources for all the three branches, we can perform the scheduling in Figure 6 (b). Additionally, We can perform batch pipeline between operators in each branch, which shall be discussed next.

5.2 Batch Pipeline

We can also partition the DAG into phases, and perform co-running in pipeline between phases for processing different batches. For simplicity, in this part, we assume that the number of phases in the DAG is two. Please note that when the platform has enough resources, the pipeline for operators can be deeper. We show a simple example in Figure 7 (a). The operators in *phase1* and the operators in *phase2* need to be mapped into different compute units, so that these two phases can co-run in the pipeline. Figure 7 (b) shows the execution flow in pipeline. When FineStream completes the processing for *batch1* in *phase1* and starts to process *batch1* in *phase2*, FineStream can start to process *batch2* in *phase1* simultaneously. *Phase1* and *phase2* can co-run because they rely on different compute units.

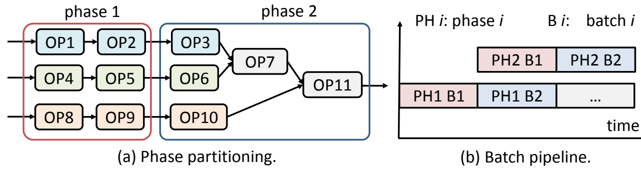

Figure 7: An example of partitioning phases for batch pipeline.

We need to estimate the bandwidth of two overlapping phases, so that we can further estimate the batch pipeline throughput. The time for a phase, t_{phase_i}, is the sum of the execution time of the operators in the phase for processing a batch. We use t_{phase1} to denote the time for *phase1* while t_{phase2} for *phase2*. When two batches are being processed in different phases in the engine, FineStream tries to maximize the overlapping of t_{phase1} and t_{phase2} of the two batches. However, the overlapping can be affected by memory bandwidth utilization. The online profiling in Section 6.3 collects the size of memory accesses $s_{i,dev}$ (including read and write) and the execution time $t_{i,dev}$ for each operator. The bandwidth of the two overlapping phases is described as Equation 4.

$$bandwidth_{overlap} = MIN(Bmax, \frac{\sum_{i=0}^{m} s_{i,dev}}{t_{phase1}} + \frac{\sum_{i=m+1}^{n} s_{i,dev}}{t_{phase2}}) \tag{4}$$

When $bandwidth_{overlap}$ does not reach *Bmax*, the execution time for processing *n* batches, $t_{nBatches}$, is shown in Equation 5.

$$t_{nBatches} = n \cdot MAX(t_{phase1}, t_{phase2}) + MIN(t_{phase1}, t_{phase2}) \tag{5}$$

When $bandwidth_{overlap}$ reaches *Bmax*, the execution time of co-running two phases in the pipeline on different batches is longer than any of their independent execution time. We assume that the independent execution time of the longer phase is t_l and the independent time for the shorter phase is t_s. Then, the overlapping ratio for the two phases r_{olp} is t_s divided by t_l. Assuming the total size of the memory accesses for the longer phase is s_l, and the total size for the shorter phase is s_s, then the execution time of the overlapping interval, t_{olp}, is shown in Equation 6.

$$t_{olp} = \frac{s_s + r_{olp} \cdot s_l}{bandwidth_{overlap}} \tag{6}$$

To estimate the time of the rest part in the longer phase, we assume that the bandwidth of the independent execution of the longer phase is $bandwidth_l$. Then, the execution time t_{rest} is shown in Equation 7.

$$t_{rest} = \frac{(1 - r_{olp}) \cdot s_l}{bandwidth_l} \tag{7}$$

Then, when bandwidth *Bmax* is reached, the execution time $t_{nBatches}$ to process *n* batches is shown in Equation 8.

$$t_{nBatches} = n \cdot (t_{olp} + t_{rest}) \tag{8}$$

We assume that a batch contains *m* tuples, and then, the throughput can be expressed by Equation 9. When bandwidth is sufficient, $t_{nBatches}$ is described as Equation 5; otherwise, Equation 8.

$$throughput_{batchPipeline} = \frac{m \cdot n}{t_{nBatches}} \tag{9}$$

Optimization. Branch co-running can also be conducted in batch pipeline. For example, in Figure 7, the branches in *phase1* can be corun when the system can provide sufficient computing resources and bandwidth. The only thing we need to do is to integrate the branch co-running technique in the potential phases.

5.3 Handling Dynamic Workload

In branch co-running, the hardware resource binded to each branch is based on the characteristics of both the operator and the workload. During workload migration, the workload pressure for each branch may be different from the original state. Hence, the static computing resource allocation may not be suitable for dynamic workload.

A possible solution is to redistribute computing resources to operators in each branch according to the performance model. However, this solution has the following two drawbacks. First, only adjusting the hardware resources on different branches may not be able to maintain the performance, because query plan topology may not fit the current streaming application. In such cases, the query plan needs to be reoptimized for system performance. Second, resource redistribution incurs overhead. Therefore, efficient *resource reallocation* and *query plan adjustment* are necessary for FineStream handling dynamic workload.

Light-Weight Resource Reallocation. In FineStream, we use a light-weight dynamic resource reallocation strategy. When the workload ingestion rate of a branch decreases, we can calculate the reduced ratio, and assume that such proportion of computing resources in that branch can be transferred to the other branches. We use an example in Figure 8 for illustration. In Figure 8 (a), 90% workload after operator *OP1* flow to *OP2*. When the workload state changes to the state in Figure 8 (b), part of the computing resource associated with *OP2* shall be assigned to *OP3* accordingly.

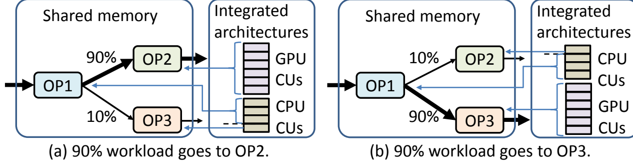

Figure 8: An example of adjustment for dynamic workload.

In detail, for the ingestion-rate-falling branch (data arrival rate of this branch is decreasing) [30], we assume that the initial ingestion rate is r_{init}, while the current ingestion rate is r_{cur}. Then, the computing resource that shall be reallocated to the other branches is shown in Equation 10. This adaptive strategy is very light-weight, because we can monitor the ingestion rate during batch loading and redistribute the proportion of reduced computing resources to the branch that has a higher ingestion rate. In the case of Figure 8 (b), we can keep limited hardware resource in OP2 and redistribute the rest to OP3 after processing the current batch.

$$resource_{redistribute_i} = \frac{r_{init} - r_{curr}}{r_{init}} \cdot resource_{OPi} \qquad (10)$$

Query Plan Adjustment. With reference to [30], FineStream generates not only the query plan that soon will be used in the stream processing, but also several possible alternatives. During stream processing, FineStream monitors the size of intermediate results. If the performance degrades and the size of intermediate results varies greatly, FineStream shall switch to another alternative query plan topology. In the implementation, FineStream generates three additional plans by default, and picks them based on the performance model.

6 Implementation Details

6.1 How FineStream Works

We present the system workflow in Figure 9. In Figure 9, *thread1* is used to cache input data, while *thread2* is used to process the cached data. The detailed workflow is as follows. First, when FineStream starts a new query, the dispatcher executes the query on the CPU for *batch1* and then on the GPU for *batch2*. Second, during these single-device executions, FineStream conducts online profiling, during which the operator-related data that are used to build the performance model are obtained, including the CPU and GPU performance, and bandwidth utilization. Third, with these data, FineStream builds the performance model considering branch co-running and batch pipeline. Fourth, after building the model, FineStream generates several query plans with detailed resource distribution. With the generated query plan, the dispatcher performs fine-grained scheduling for processing the following batches. When dynamic workload is detected, FineStream performs related adjustment mentioned in Section 5.3. For the operators in FineStream, we reuse the operator code from OmniDB [64]. Please note that the goal of this work is to provide a fine-grained stream processing method on integrated architectures. The same methodology can also be applied for using other OpenCL processing engine such as Voodoo [45].

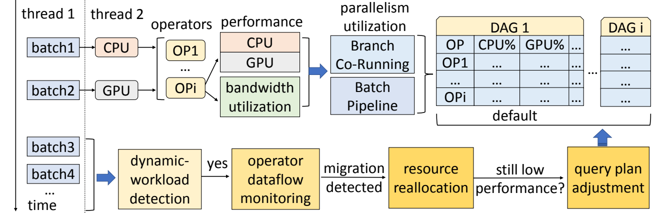

Figure 9: FineStream workflow.

Additionally, when users change the window size of a query on the fly, FineStream updates the window size parameter after the related batch processing is completed, and then continues to run with performance detection. If the performance decreases below a given value (70% of the original performance by default), FineStream re-determines the query plan with computing resource based on the parameters and the performance model.

6.2 Dispatcher

The dispatcher of FineStream is a module for assigning stream data to the related operators with hardware resources. The dispatcher has two functions. First, it splits the stream data into batches with a fixed size. Second, it sends the batches to the corresponding operators with proper hardware resources for execution. The goal of the dispatcher is to schedule operator tasks to the appropriate devices to fully utilize the capacity of the integrated architecture.

Algorithm 1 is the pseudocode of the dispatcher. When a stream is firstly presented in the engine, FineStream conducts branch co-running and batch pipeline according to the performance model mentioned in Section 5 (Lines 2 to 5). FineStream also detects dynamic workload (Lines 6 to 15). If dynamic workload is detected, FineStream conducts the related resource reallocation. If such reallocation does not help, it further conducts query plan adjustment.

6.3 Online Profiling

The purpose of online profiling is to collect useful performance data to support building the performance model.

In online profiling, we have two concerns. The first is what data to generate in this module. This is decided by the performance model. These data include the data size, execution time, bandwidth utilization, and throughput for each operator on devices. The second is, to generate the data, what information we shall collect from stream processing.

Algorithm 1: Scheduling Algorithm in FineStream

```
1  Function dispatch(batch, resource, model):
2      if taskFirstRun then
3          branchCoRun(resource, model)
4          batchPipeline(resource, model)
5          taskFirstRun = false
       // Handling dynamic workload and query plan optimization
6      if detectDynamicWorkload() then
7          resourceReallocate()
8          if resourceChanged == true then
9              if performanceDegrade() then
10                 adjustQueryPlan()
11                 queryChanged = true
12         resourceChanged = true
13         if queryChanged == true then
14             resourceChanged = false
15             queryChanged = false
```

FineStream performs online profiling for operators from memory bandwidth and computation perspectives.

Memory Bandwidth Perspective. Based on the above analysis, we use *bandwidth*, defined as the transmitted data size divided by the execution time, to depict the characteristics from data perspective of an operator. The transmitted data for an operator consists of input and output. The input relates to the batch while the output relates to both the operator and the batch. We define the bandwidth of the operator i on device dev in Equation 11. The parameters s_{input_i} and s_{output_i} denote the estimated input and the output sizes of the operator i, and $t_{i,dev}$ represents the execution time of the operator i on device dev.

$$bandwidth_{i,dev} = \frac{s_{input_i} + s_{output_i}}{t_{i,dev}} \quad (11)$$

Computation Perspective. To depict the characteristics from the computation perspective, we use $throughput_{i,dev}$, which is defined as the total number of processed tuples n_{tuples} divided by the time $t_{i,dev}$ for operator i on device dev. All these values can be obtained from online profiling.

7 Evaluation

7.1 Methodology

The baseline method used in our comparison is Saber [34], while our method is denoted as "FineStream". Saber is the state-of-the-art window-based stream processing engine for discrete architectures. It adopts a bulk-synchronous parallel model [66]. The whole query execution on a batch is distributed to a device, the GPU or the CPU, without further distributing operators of a query to different devices. The original CPU operators in Saber are written in Java, and we further rewrite the CPU operators in Saber in OpenCL for higher efficiency. Our comparisons to Saber examine whether our fine-grained method delivers better performance. To validate the co-running benefits of the two devices, we also measure the performance using only the CPU and the performance using only the GPU, denoted as "CPU-only" and "GPU-only". Further, to understand the advantage of using the integrated architecture to accelerate stream processing, we compare FineStream on integrated architectures with Saber on discrete CPU-GPU architectures.

Platforms. We perform experiments on four platforms, two integrated platforms and two discrete platforms. The first integrated platform uses the integrated architecture AMD A10-7850K [15], and it has 32 GB memory. The second integrated platform uses the integrated architecture Ryzen 5 2400G, which is the latest integrated architecture, and this platform has 32 GB memory. The first discrete platform is equipped with an Intel i7-8700K CPU and an NVIDIA GeForce GTX 1080Ti GPU, and along with 32 GB memory. The second discrete platform is equipped with two Intel E5-2640 v4 CPUs and an NVIDIA V100-32GB GPU, and has 264 GB memory.

Datasets. We use four datasets in the evaluation. The first dataset is Google compute cluster monitoring [2], which emulates a cluster management scenario. The second dataset is anomaly detection in smart grids [68], which is about detection in energy consumption from different devices of a smart electricity grid. The third dataset is linear road benchmark [7], which models a network of toll roads. These traces come from real-world applications, and are widely used in previous studies such as [19, 34, 39]. The fourth dataset is a synthetically generated dataset [34] for evaluating independent operators, where each tuple consists of a 64-bit timestamp and six 32-bit attributes drawn from a uniform distribution. To overfeed the system and test its performance capacity, we load the data from memory. This method avoids network being bottleneck. In practice, the system obtains stream data via network.

Benchmarks. We use nine queries to evaluate the overall performance of the fine-grained stream processing engine on the integrated architectures. Similar benchmarks have been used in [34]. The details of the nine queries are shown in Table 3. *Q1*, *Q2*, and *Q3* are conducted on the Google compute cluster monitoring dataset. *Q4*, *Q5*, and *Q6* are for the dataset of anomaly detection in smart grids. *Q7*, *Q8*, and *Q9* are for the dataset from the linear road benchmark.

Dynamic Workload Generation. We use the datasets and benchmarks to generate dynamic workload. For the first dataset of cluster monitoring, the seventh attribute of *category* gradually changes from type 1 to type 2 within 10,000 batches. We use the query *Q1* for illustration, and we denote it as *T1*. Similar evaluations are also conducted on the second dataset of smart grid with the query *Q5*, which is denoted as *T2*, and the third dataset of linear road benchmark with the query *Q8*, which is denoted as *T3*.

7.2 Performance Comparison

Throughput. We explore the throughput of FineStream for the nine queries. Figure 10 shows the processing throughput of the best single device, Saber, and FineStream for these queries on both the A10-7850K and Ryzen 5 2400G platforms. Please note that the y-axis of the figure is in log scale. We have the following observations. First, on the A10-7850K platform, FineStream achieves 88% throughput improvement

Table 3: The queries used in evaluation.

Query	Detail
Q1	`select timestamp, category, sum(cpu) as totalCPU from TaskEvents [range 256 slide 1] group by category`
Q2	`select timestamp, jobID, avg(cpu) as avgCPU from TaskEvents [range 256 slide 1] where eventType == 1 group by jobId`
Q3	`select timestamp, eventType, userId, max(disk) as maxDisk from TaskEvents [range 256 slide 1] group by eventType, userId`
Q4	`select timestamp, avg (value) as globalAvgLoad from SmartGridStr [range 512 slide 1]`
Q5	`select timestamp, plug, household, house, avg(value) as localAvgLoad from SmartGridStr [range 512 slide 1] group by plug, household, house`
Q6	`(select L.timestamp, L.plug, L.household, L.house from LocalLoadStr [range 1 slide 1] as L, GlobalLoadStr [range 1 slide 1] as G where L.house == G.house and L.localAvgLoad >G.globalAvgLoad) as R - select timestamp, house, count(*) from R group by house`
Q7	`(select timestamp, vehicle, speed, highway, lane, direction, (position/5280) as segment from PosSpeedStr [range unbounded]) as SegSpeedStr - select distinct L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment from SegSpeedStr [range 30 slide 1] as A, SegSpeedStr [partition by vehicle rows 1] as L where A.vehicle == L.vehicle`
Q8	`select timestamp, vehicle, count(direction) from PosSpeedStr [range 256 slide 1] group by vehicle`
Q9	`select timestamp, max(speed), highway, lane, direction from PosSpeedStr [range 256 slide 1] group by highway,lane,direction`

over the best single device performance on average; compared to Saber, FineStream achieves 52% throughput improvement. Because of the efficient CPU and GPU co-running, FineStream nearly doubles the performance compared to the method of using only a single device. Because FineStream adopts the continuous operator model where each operator could be scheduled on its preferred device, FineStream utilizes the integrated architecture better than Saber that uses the bulk-synchronous parallel model. Such result clearly shows the advantage of fine-grained stream processing on the integrated architecture. Second, on the Ryzen 5 2400G platform, all hardware configurations have been upgraded in comparison with A10-7850K, especially the CPUs; the CPU-only throughput on Ryzen 5 2400G is much higher than that on A10-7850K. Moreover, Saber achieves a 56% throughput improvement compared to the throughput of the best single device, and FineStream is still 14% higher than Saber on this platform. Similar phenomena have also been observed in [58–60].

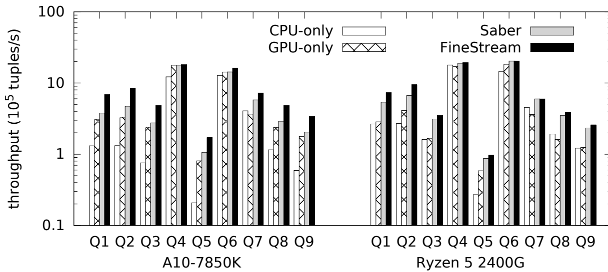

Figure 10: Throughput of different queries.

Latency. Figure 11 reports the latency of different queries on the integrated architectures. In this work, latency is defined as the end-to-end time from the time a query starts to the time it ends. FineStream has the lowest latency among these methods. First, on the A10-7850K platform, FineStream's latency is 10% lower than that of the best single device, and 36% lower than the latency of Saber. Second, on Ryzen 5 2400G platform, it is 2% lower than that of the best single device, and 9% lower than that of Saber. The reason is that FineStream considers device preference for operators and assigns the operators to their suitable devices. In this way, each batch can be processed in a more efficient manner, leading to lower latency.

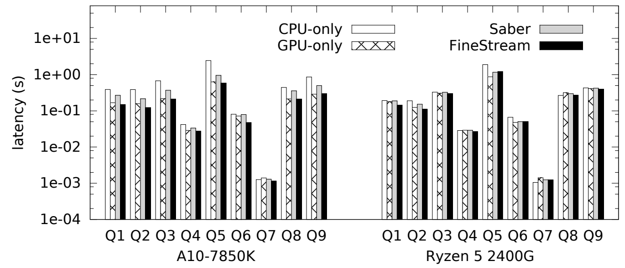

Figure 11: Latency of different queries.

Profiling. We show the relationship between throughput and latency of both FineStream and Saber in Figure 12. Figure 12 shows that queries with high throughput usually have low latency, and vice versa.

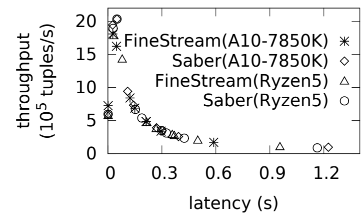

Figure 12: Throughput vs. latency.

We further study the CPU and GPU utilization of Saber and FineStream, and use the A10-7850K platform for illustration, as shown in Figure 13. In most cases, FineStream utilizes the GPU device better on the integrated architecture. As for *Q*4, the CPU processes most of the workload. On average, FineStream improves 23% GPU utilization compared to Saber, and have roughly the same CPU utilization as Saber. Since FineStream achieves better throughput and latency than Saber, such utilization results indicate that FineStream generates effective strategies in determining device preferences for individual operators.

7.3 Comparison with Discrete Architectures

In this part, we compare FineStream on the integrated architectures and Saber on the discrete architectures from three perspectives: performance, price, and energy-efficiency.

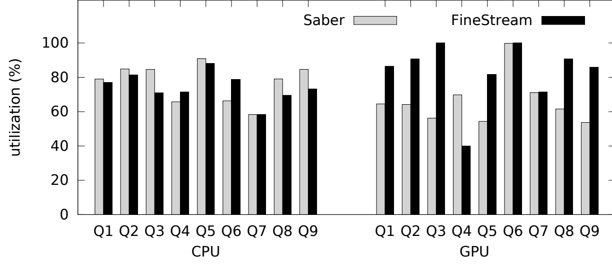

Figure 13: Utilization of the integrated architecture.

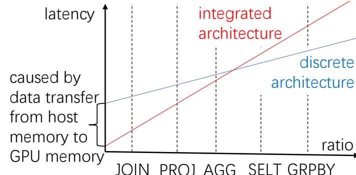

Figure 14: Latency comparison of different operators.

Performance Comparison. The current GPU on the integrated architecture is less powerful than the discrete GPU, as mentioned in Section 2.1. The discrete GPUs exhibit 1.8x to 5.7x higher throughput than the integrated architectures, due to the more computational power of discrete GPUs. However, the integrated architecture demonstrates lower processing latency compared to the discrete architecture when the data transmission cost between the host memory and GPU memory in the workload is significant. For example, the latencies for *projection*, *selection*, *aggregation*, *group-by*, and *join* are 0.6, 1.5, 1.0, 10.6, and 1924.5 ms on Ryzen 5 2400G platform, while 1.1, 1.2, 1.2, 1.6, and 7600.1 ms on GTX 1080Ti platform; these operators are distributed in Figure 14, where *join* (JOIN), *projection* (PROJ), and *aggregation* (AGG) achieve lower latency on the integrated architecture, while *selection* (SELT), and *group-by* (GRPBY) prefer the discrete architecture. The x-axis represents the ratio of $m_{compute}/(s_{write}+s_{read})$ where $m_{compute}$ denotes the kernel computation workload size, and t_{write} and s_{read} denote the data transmission sizes from the host memory to the GPU memory and from the GPU memory to the host memory via PCI-e. For further explanation, to execute a kernel on discrete GPUs, the execution time t_{total} includes 1) the time t_{write} of data transmission from the host memory to the GPU memory via PCI-e, 2) the time $t_{compute}$ for data processing kernel execution, and 3) the time t_{read} of data transmission from the GPU memory to the host memory. As for executing a kernel on the integrated architecture, although its $t_{compute}$ is longer than that on discrete GPUs, its t_{write} and t_{read} can be avoided. For the queries in Table 3, the data movement overhead on discrete architectures ranges from 31 to 62%.

Price-Throughput Ratio Comparison. FineStream on integrated architectures shows a high price-throughput ratio, compared to Saber on the discrete architectures. The price of the 1080Ti discrete architecture is about 7x higher than that of the A10-7850K integrated architecture, and the price of the V100 discrete architecture is about 64x higher than that of the Ryzen 5 2400G integrated architecture. Figure 15 shows the comparison of their price-throughput ratio. On average, FineStream on the integrated architectures outperforms Saber on the discrete architectures by 10.4x.

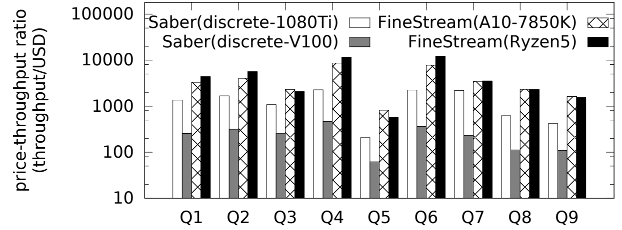

Figure 15: Comparison of price-throughput ratio.

Energy Efficiency Comparison. We also analyze the energy efficiency of FineStream and Saber. The Thermal Design Power (TDP) is 95W on A10-7850K, and 65W on Ryzen 5 2400G. For the 1080Ti platform, the TDP of the Intel i7-8700K CPU and NVIDIA GTX 1080Ti GPU are 95W and 250W, respectively. For the V100 platform, the TDP of the Intel E5-2640 v4 CPU and NVIDIA V100 GPU are 90W and 300W, respectively. Similar to [61], we use performance per Watt to define energy efficiency. On average, FineStream on the integrated architectures is 1.8x energy-efficient than Saber on the discrete architectures.

7.4 Handling Dynamic Workload

In this section, we discuss how to handle dynamic workloads. To demonstrate the capability of FineStream to handle dynamic workload, we evaluate FineStream on the dynamic workloads mentioned in Section 7.1. On average, FineStream achieves a performance of 323,727 tuples per second, which outperforms the static method (we denote "static" for FineStream without adapting to dynamic workload) by 28.6%, as shown in Table 4.

Table 4: Throughput of the queries on dynamic workloads.

Dynamic	A10-7850K (10^5 tuples/s)		Ryzen 5 2400G (10^5 tuples/s)	
Workload	Static	FineStream	Static	FineStream
T1	4.2	5.1	4.4	5.1
T2	0.8	1.2	1.1	1.5
T3	1.9	2.8	2.7	3.7

We use *T1* as an example, and show the detailed throughput along with the number of batches in Figure 16. In the timeline process, the static method decreases due to the improper hardware resource distribution. As for FineStream, the hardware computing resources can be dynamically adjusted according to the data distribution, so the performance does not decline with the changes.

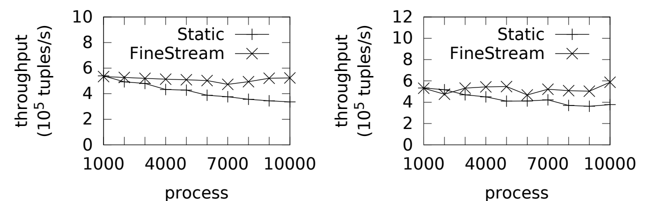

Figure 16: Throughput of *T1* on dynamic workloads.

7.5 Detailed Analysis

Performance Model Accuracy. In stream processing, after each batch is processed, we use the measured batch processing speed to correct our model. We use the example of *Q1* for illustration, as shown in Figure 17. We use the percent deviation to measure the accuracy of our performance model. The percent deviation is defined as the absolute value of the real throughput minus the estimated throughput, divided by the real throughput. The smaller the percent deviation is, the more accurate the predicted result is. The deviation decreases as the number of processed batches increases. After 20 batches are processed, we can reduce the deviation to less than 10%. Please note that in stream processing scenarios, input tuples are continuously coming, so the time for correcting performance prediction can be ignored in stream processing. For dynamic workload, the accuracy also depends on the intensity of workload changes.

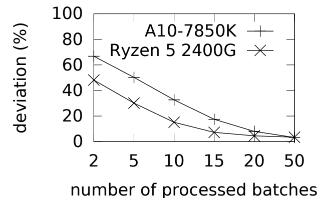

Figure 17: Deviation of *Q1*.

Runtime Overhead Analysis. FineStream incurs runtime overhead in the batch processing phase from two aspects. First, it detects whether the input stream belongs to dynamic workload, which causes time overhead. Second, the scheduling also takes time. In our evaluation, we observe that the time overhead accounts for less than 2% of the processing time, which can be ignored in stream processing.

8 Related Work

Parallel stream processing [4, 10, 12, 21, 28, 29, 34, 43, 46, 63], query processing [9, 14, 16, 20, 22, 56], and heterogeneous systems [11, 17, 24, 26, 32, 35–38, 45, 48, 50] are hot research topics in recent years. Different from these works, FineStream targets sliding window-based stream processing, which focuses on window handling with SQL and dynamic adjustment. GPUs have massive threads and high bandwidth, and have emerged to be one of the most promising heterogeneous accelerators to speedup stream processing. Verner et al. [54] presented a stream processing algorithm considering various latency and throughput requirements on GPUs. Alghabi et al. [5] developed a framework for stateful stream data processing on multiple GPUs. De Matteis et al. [25] developed Gasser system for offloading operators on GPUs. Pinnecke et al. [44] studied how to efficiently process large windows on GPUs. Chen et al. [23] extended the popular stream processing system, Storm [1], to GPU platforms. Augonnet et al. [8] explored data-aware task scheduling for multi-devices, which can be integrated into this work. FineStream differs from those previous works in two aspects: firstly on integrated architectures, and secondly for SQL streaming processing.

The closest work to FineStream is Saber [34], which aims to utilize discrete CPU-GPU architectures. Saber [34] adopts a *bulk-synchronous parallel* model [53, 66], where the whole query (with multiple operators) on each batch of input data is dispatched on one device. Such a mechanism naturally minimizes the communication overhead among operators inside the same query. It is hence suitable in discrete CPU-GPU architectures, where PCI-e overhead is significant and shall be avoided as much as possible. However, it may result in suboptimality in integrated architectures for mainly two reasons. First, it overlooks the performance difference between different devices for each operator. Second, the communication overhead between the CPU and the GPU in integrated architectures is negligible. Targeting at integrated architectures, FineStream adopts continuous operator model [53, 66], where each operator of a query can be independently placed at a device. We further build a performance model to guide operator-device placement optimization. It is noteworthy that our fine-grained operator placement is different from classical placement strategies for general stream processing [18, 19, 40, 51, 65] for their different design goals. In particular, most prior works aim at reducing communication overhead among operators, which is not an issue in FineStream. Instead, FineStream needs to take device preference into consideration during placement optimization, which has not been considered before.

9 Conclusion

Stream processing has shown significant performance benefits on GPUs. However, the data transmission via PCI-e hinders its further performance improvement. This paper revisits window-based stream processing on the promising CPU-GPU integrated architectures, and with CPUs and GPUs integrated on the same chip, the data transmission overhead is eliminated. Furthermore, such integration opens up new opportunities for fine-grained cooperation between different devices, and we develop a framework called FineStream for fine-grained stream processing on the integrated architecture. This study shows that integrated CPU-GPU architectures can be more desirable alternative architectures for low-latency and high-throughput data strream processing, in comparison with discrete architectures. Experiments show that FineStream can improve the performance by 52% over the state-of-the-art method on the integrated architecture. Compared to the stream processing engine on the discrete architecture, FineStream on the integrated architecture achieves 10.4x price-throughput ratio, 1.8x energy efficiency, and can enjoy lower latency benefits.

Acknowledgments

We sincerely thank our shepherd Sergey Blagodurov and the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), National Natural Science Foundation of China (No. 61732014, 61802412, 61972402, 61972403), and Beijing Natural Science Foundation (No. L192027), and is also supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. Xiaoyong Du is the corresponding author of this paper.

References

[1] Apache Storm. http://storm.apache.org/.

[2] More google cluster data. https://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[3] The Compute Architecture of Intel Processor Graphics Gen7.5. https://software.intel.com/.

[4] Nitin Agrawal and Ashish Vulimiri. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *SOSP*, 2017.

[5] Farhoosh Alghabi, Ulrich Schipper, and Andreas Kolb. A scalable software framework for stateful stream data processing on multiple gpus and applications. In *GPU Computing and Applications*. 2015.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 2006.

[7] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *PVLDB*, 2004.

[8] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *ICPADS*, 2010.

[9] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[10] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *INFOCOM*, 2016.

[11] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *USENIX ATC*, 2017.

[12] Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilowska, Karsten Schwan, and Adam Allred. Appflux: Taming app delivery via streaming. *Proc. of the Usenix TRIOS*, 2015.

[13] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's First 64-bit ARM Processor. *Micro*, 2015.

[14] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 2008.

[15] Dan Bouvier and Ben Sander. Applying AMD's Kaveri APU for heterogeneous computing. In *Hot Chips Symposium*, 2014.

[16] Sebastian Breß and Gunter Saake. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *PVLDB*, 2013.

[17] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. Mobirnn: Efficient recurrent neural network execution on mobile GPU. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, 2017.

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[19] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[20] Badrish Chandramouli, Raul Castro Fernandez, Jonathan Goldstein, Ahmed Eldawy, and Abdul Quamar. Quill: efficient, transferable, and rich analytics at scale. *PVLDB*, 2016.

[21] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. Accurate latency estimation in a distributed event processing system. In *ICDE*, 2011.

[22] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 2014.

[23] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, and Charles Kamhoua. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *Big Data*, 2015.

[24] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proceedings of the VLDB Endowment*, 2019.

[25] Tiziano De Matteis, Gabriele Mencagli, Daniele De Sensi, Massimo Torquati, and Marco Danelutto. GASSER: An Auto-Tunable System for General Sliding-Window Streaming Operators on GPUs. *IEEE Access*, 2019.

[26] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *HPDC*, 2019.

[27] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *Micro*, 2017.

[28] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. NVStream: accelerating HPC workflows with NVRAM-based transport for streaming objects. In *HPDC*, 2018.

[29] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *USENIX ATC*, 2019.

[30] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *TPDS*, 2013.

[31] Younghwan Go, Muhammad Asim Jamshed, Young-Gyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *NSDI*, 2017.

[32] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD*, 2020.

[33] Chandima HewaNadungodage, Yuni Xia, and John Jaehwan Lee. GStreamMiner: a GPU-accelerated data stream mining framework. In *CIKM*, 2016.

[34] Alexandros Koliousis and et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, 2016.

[35] Xinyu Li, Lei Liu, Shengjie Yang, Lu Peng, and Jiefan Qiu. Thinking about A New Mechanism for Huge Page Management. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019.

[36] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. *TPDS*, 2019.

[37] Alexander M Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. In *VTDC*, 2011.

[38] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *HPCA*, 2015.

[39] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing*, 2014.

[40] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *ICDM Workshops*, 2010.

[41] Dong Nguyen and Jongeun Lee. Communication-aware mapping of stream graphs for multi-GPU platforms. In *CGO*, 2016.

[42] John Nickolls and William J Dally. The GPU computing era. *Micro*, 2010.

[43] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[44] Marcus Pinnecke, David Broneske, and Gunter Saake. Toward GPU Accelerated Data Stream Processing. In *GvD*, 2015.

[45] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 2016.

[46] Arosha Rodrigo, Miyuru Dayarathna, and Sanath Jayasena. Latency-Aware Secure Elastic Stream Processing with Homomorphic Encryption. *Data Science and Engineering*, 2019.

[47] Michael J Schulte, Mike Ignatowski, Gabriel H Loh, Bradford M Beckmann, William C Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K Reinhardt, and Gregory Rodgers. Achieving exascale capabilities through heterogeneous computing. *Micro*, 2015.

[48] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *TOCS*, 2016.

[49] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[50] Zhi Tang and Youjip Won. Multithread content based file chunking system in CPU-GPGPU heterogeneous architecture. In *2011 First International Conference on Data Compression, Communications and Processing*, 2011.

[51] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *SIGMOD*, 2014.

[52] Sid Touati and Benoit Dupont De Dinechin. *Advanced Backend Code Optimization*. John Wiley & Sons, 2014.

[53] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[54] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *ICS*, 2011.

[55] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Bradford M Beckmann, William C Brantley, Joseph L Greathouse, Wei Huang, Arun Karunanithi, et al. Design and Analysis of an APU for Exascale Computing. In *HPCA*, 2017.

[56] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical off-chip meta-data for temporal memory streaming. In *HPCA*, 2009.

[57] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[58] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *CGO*, 2017.

[59] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, Wenguang Chen, and Xiaoyong Du. Automatic Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. *TKDE*, 2019.

[60] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. Understanding co-running behaviors on integrated cpu/gpu architectures. *TPDS*, 2017.

[61] Kai Zhang, Jiayu Hu, Bingsheng He, and Bei Hua. DIDO: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *ICDE*, 2017.

[62] Kai Zhang, Jiayu Hu, and Bei Hua. A holistic approach to build real-time stream processing system with GPU. *JPDC*, 2015.

[63] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *ICDE*, 2017.

[64] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB*, 2013.

[65] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. Briskstream: Scaling Data Stream Processing on Multicore Architectures. In *SIGMOD*, 2019.

[66] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. Hardware-conscious stream processing: A survey. *SIGMOD Rec.*, 2020.

[67] Yongpeng Zhang and Frank Mueller. GStream: A general-purpose data streaming framework on GPU clusters. In *ICPP*, 2011.

[68] Holger Ziekow and Zbigniew Jerzak. The DEBS 2014 grand challenge. In *DEBS*, 2014.