Fine-Grained Multi-Query Stream Processing on Integrated Architectures

Feng Zhang, Chenyang Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du

Abstract—Exploring the sharing opportunities among multiple stream queries is crucial for high-performance stream processing. Modern stream processing necessitates accelerating multiple queries by utilizing heterogeneous coprocessors, such as GPUs, and this has shown to be an effective method. Emerging CPU-GPU integrated architectures integrate CPU and GPU on the same chip and eliminate PCI-e bandwidth bottleneck. Such a novel architecture provides new opportunities for improving multi-query performance in stream processing but has not been fully explored by existing systems. We introduce a stream processing engine, called FineStream, for efficient multi-query window-based stream processing on CPU-GPU integrated architectures. FineStream's key contribution is a novel fine-grained workload scheduling mechanism between CPU and GPU to take advantage of both architectures. Particularly, FineStream is able to efficiently handle multiple queries in both static and dynamic streams. 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 coarse-grained strategy of applying different devices for multiple queries, FineStream achieves 32% throughput improvement; 3) compared to the stream processing engine on the discrete architecture, FineStream on the integrated architecture achieves 10.4× price-throughput ratio, 1.8× energy efficiency, and can enjoy lower latency benefits.

Index Terms—Fine-grained, stream processing, CPU, GPU, integrated architectures.

1 Introduction

Multi-query optimization is essential to modern stream processing systems. Driven by the rise of the internet of things (IoT) and widely deployed 5G sensor networks, many real-world applications, such as cluster monitoring, anomaly detection in smart grids, and road tolling systems, have been proposed recently [1], [2], [3]. They often have common interests in the same data streams, leading to tremendous sharing opportunities among queries. Due to the rigid requirement and complexities of handling multiple queries in streams, accelerating stream processing engines has recently become a hot research topic. GPUs consist of a large number of lightweight computing cores, which are naturally suitable for data-intensive stream processing with multi-query supported. Prior studies [4], [5], [6] have shown that GPUs can be successfully utilized to improve the multi-query stream processing performance.

GPUs are often used as coprocessors that are connected to CPUs through PCI-e [7]. Under such discrete architectures, the input stream needs to be moved from the main memory to the GPU memory via PCI-e before GPU processing. As a result, the low bandwidth of PCI-e severely limits the performance gain of stream processing on GPUs. Subsequently, multi-query 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 [4], [8].

Hardware vendors have released integrated architectures in recent years, which completely remove PCI-e overhead. We have seen CPU-GPU integrated architectures such as NVIDIA Denver [9], AMD Kaveri [10], and Intel Skylake [11]. 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 overhead. Such integration poses new opportunities for multi-query window-based stream processing from both hardware and software perspectives.

First, in contrast to discrete CPU-GPU architectures with a separate memory hierarchy, the integrated architectures provide unified physical memory. The input queries can be processed in the shared memory of both CPUs and GPUs, which avoids the data transmission between two memory hierarchies, thus eliminating the data copy overhead via PCI-e.

Second, via fine-grained cooperations between CPUs and GPUs, the integrated architecture makes it possible for processing dynamic workloads. Streaming queries can consist of multiple operators with varying performance features on different processors. Furthermore, stream processing often involves a dynamic input workload, which affects operator performance behaviors as well. We can place operators on different devices with proper workloads in a fine-grained manner, without worrying about transmission overhead between CPUs and GPUs.

Based on the above analysis, we argue that multi-query stream processing on integrated architectures can have

- *F. Zhang, C. Zhang, L. Yang, W. Lu, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and with the School of Information, Renmin University of China, Beijing 100872, China.*
- *S. Zhang is with Information Systems Technology and Design pillar, Singapore University of Technology and Design, 487372, Singapore.*
- *B. He is with the School of Computing, National University of Singapore, 119077, Singapore.*
  (Corresponding author: Xiaoyong Du)

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. The fine-grained mechanism refers to that the operators in one query can be allocated to different devices, which is different from the mechanism that all the operators in a query need to be mapped to the same device, so-called coarse-grained mechanism. In this paper, a query represents a statement from a user to view data in streams. Currently, we support queries with five basic kinds of operators: *projection*, *selection*, *aggregation*, *group-by*, and *join*. Note that more operators can be supported in the future. More details are presented in Section 6. Our preliminary work has been presented in [12], which supports only single query. In practice, users may submit multiple queries for the same or different input streams. In this work, we extend FineStream to support multiple queries. Specifically, we propose the following key techniques. First, a performance model is proposed considering both operator topologies and different device characteristics of integrated architectures. Second, a light-weight scheduler is developed to efficiently assign the operators of a query to different processors. Third, online profiling with computing resource and topology adjustment is involved for dynamic workloads. Fourth, novel designs of operator sharing and ingestion rate detection among different queries have been proposed to handle multi-query workloads efficiently.

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-tart CPU-GPU stream processing engine on the integrated architecture. Compared to the best single processor throughput, it achieves 88% performance improvement. For multi-query processing, FineStream achieves 32% throughput improvement compared to the *bulk-synchronous strategy* [13] of allocating different queries to different devices without further operator-level partitioning.

We also compare stream processing on integrated architectures with that on discrete CPU-GPU architectures. Our evaluation shows that FineStream on integrated architectures achieves 10.4× price-throughput ratio, and 1.8× energy efficiency. When we compare FineStream on integrated architectures with stream processing on discrete architectures, although a discrete GPU could have 10× higher computation capacity and 22× higher bandwidth than the integrated architecture, the PCIe transmission could become its bottleneck. In contrast, the integrated architecture avoids PCIe transmission. Under the circumstances that the data transmission time is long and the transmission overhead exceeds the performance benefits brought by discrete GPUs of the discrete architectures, FineStream is able to achieve lower processing latency, compared to the state-of-the-art execution on discrete architectures. More details are shown in Section 8.4. This further validates the large potential of exploring the integrated architectures for data stream processing.

Overall, this work makes the following contributions:

- It proposes the first fine-grained window-based relational stream processing framework that takes the advantages of the special features of integrated architectures.
- It presents lightweight query plan adaptations for handling dynamic workloads with the performance model that considers both operator and architecture characteristics.
- It develops operator sharing and stream ingestion rate detection to handle multiple queries in both single-stream and multi-stream situations efficiently.
- It evaluates FineStream on a set of stream queries to demonstrate the performance benefits over current approaches.

2 Background and Motivation

2.1 Multi-Query Stream Processing with SQL

Various multi-query processing systems have appeared [4], [5], [6]. In this paper, a query is a way of requesting information from a stream with continuous semantics. Accordingly, multi-query in this work refers to the usage of multiple queries to retrieve information on the same machine with no further restrictions. The closest work to ours is Saber [4], [5]. Saber adopts a *bulk-synchronous parallel model* [13] for hiding PCI-e transmission overhead, and supports structured query language (SQL) on stream data [14]. 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 is about to explore multi-query stream processing with SQL on integrated architectures.

Multiple queries can be transformed into a uniformed query for processing in FineStream. For simplicity, we use one stream in Figure 1 for illustration. According to [14], 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 [4], 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.

2.2 Integrated Architecture

An architectural overview of the CPU-GPU integrated architecture is shown in Figure 2. The integrated architecture consists of a CPU, a GPU, a shared memory management

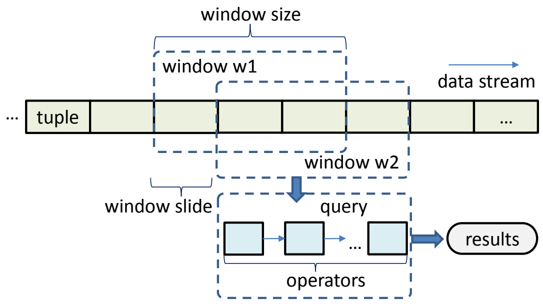


Fig. 1. Stream processing with SQL.

unit, and system DRAM. CPUs and GPUs have their own caches. Some models of integrated architectures, such as Intel Haswell i7-4770R processor [15], integrate a shared last-level cache for both CPUs and GPUs. The shared memory management unit is responsible for scheduling access 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 [16], 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 2.

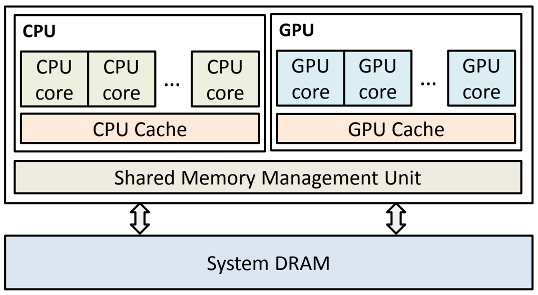


Fig. 2. A general overview of the integrated architecture.

Table 1 shows a comparison between the integrated and discrete architectures (discrete GPUs). These architectures are used in our evaluation (Section 8). 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 [10], Intel [11] and NVIDIA [9], all release their CPU-GPU integrated architectures, which couple the CPU and the GPU on the same chip. Moreover, future integrated architectures can be much more powerful, even can be a competitive building block for exascale HPC systems [17], [18], and the insights and methods in this paper still can be applied. Besides, the integrated architectures are attractive due to its efficient power consumption [19], [20] and low price [21].

2.3 Motivation

The integrated architectures bring new opportunities to stream processing. First, previous heterogeneous stream frameworks, such as Saber [4], aim to utilize discrete CPU-GPU architectures by dispatching the whole query on one

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.

device (CPU or GPU), which minimizes the PCI-e communication overhead among operators inside the same query. In contrast, the integrated architectures provide unified physical memory. The input queries can be processed in the shared memory, which avoids the data transmission between the CPU and GPU memory hierarchies. Second, streaming queries can consist of multiple operators with varying performance features on different processors. We can place operators on their preferred device without worrying about transmission overhead between CPUs and GPUs. Third, with shared memory, light-weight resource reallocation and query plan adjustment can be conducted for dynamic workloads.

3 Revisiting Stream Processing

In this section, we discuss in detail the new opportunities (Section 3.1) and challenges (Section 3.2) for multi-query stream processing on integrated architectures.

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 multiple queries, 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 synthetically generated dataset for illustration, which is explained in detail in Section 8.1. The performance results on the other datasets also exhibit similar observation: using a single type of device cannot achieve the optimal performance for all operators. For example, we use a simple query of *group-by* and *aggregation* on the integrated architecture for analysis, 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 8.1. 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.

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

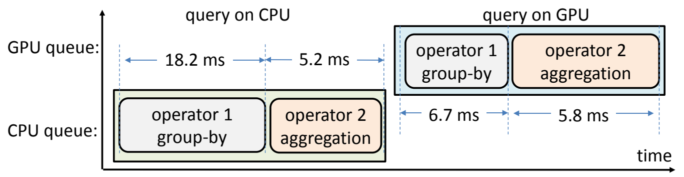


Fig. 3. An example of operator-device preference.

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 6× 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 multi-query 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 co-operation is inefficient on discrete CPU-GPU architectures due to transmission overhead. For example, Saber [4], one of the state-of-the-art stream processing engines utilizing the discrete CPU-GPU architecture, is designed aiming to hide PCI-e overhead. It adopts a micro-batch execution model which minimizes cross-operator communication but restricts operator scheduling flexibility since all operators of a query must be scheduled to one processor to process a micro-batch of data [22].

3.2 Fine-Grained Multi-Query Stream Processing

Challenges: A fine-grained multi-query 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 multi-query stream processing on integrated architectures can have more desirable properties than that on discrete CPU-GPU architectures. Particularly, this work introduces a concept of multi-query 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 multi-query 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, multiple SQL queries 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 [4], [8], [23], [24], [25], [26], [27] 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. The reason is that 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

For fine-grained multi-query stream processing on integrated architectures, we propose a framework, called FineStream. The overview of FineStream is shown in Figure 4. FineStream consists of four major components, including performance model, online profiling, dispatcher, and multi-query handling. The online profiling module is used to analyze input batches and queries for memory bandwidth and throughput (detailed in Section 7.3), 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. The multi-query handling module starts to work if multiple queries are detected.

Multi-query handling. For multiple queries in a single stream, we transform the multiple queries into the same one, because the input for different queries is the same. For multiple queries in multiple streams, we use separate threads with different computing resources to handle each stream. We elaborate the details in Section 6.

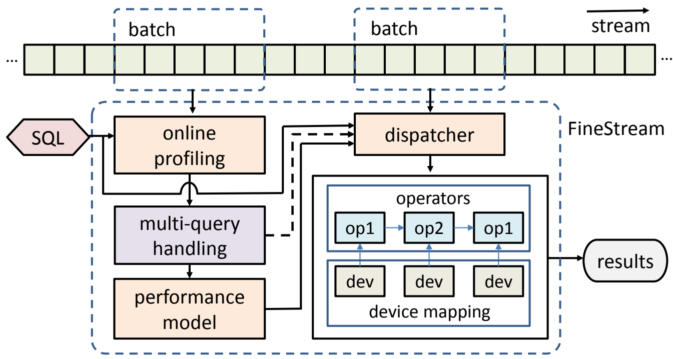


Fig. 4. FineStream overview.

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 for each query 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 [28], 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. Because 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 OPi is t_{OPi}. *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}.

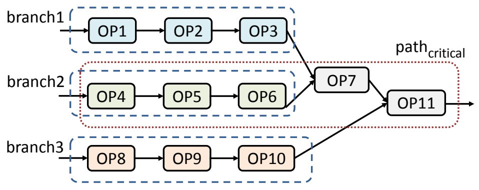


Fig. 5. An example of query operators in DAG representation, where OP_i represents an operator.

Batch vs Window. FineStream processes stream data at batch granularity. Moreover, previous studies such as Saber [4] also use batches. However, users define queries in window granularity. A common question is why we need to use *batch* instead of using *window* directly. The reason for using batches relates to parallelism: we want to use the proper size of the data unit to exploit the hardware parallelism, but the *window* size is application dependent. When the user-defined window size is small, the engine can organize a group of windows as a batch, and distributes the batch to the related operators on CPUs or GPUs for processing at a relatively larger granularity. When the window size is too large, a window can be split into several batches. Therefore, a batch can be regarded as a tuning parameter in system optimizations, while a window belongs to application semantics.

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, when FineStream processes different batches in stream, *OP2* and *OP1* can execute simultaneously: when *OP2* processes the first batch, *OP1* can start to process the second batch in parallel. However, when the platform is unable to provide enough resources for parallelizing *OP1* and *OP2* at the batch level, we can map both *OP1* and *OP2* to the same core. If so, *OP1* and *OP2* can be regarded as a whole operator that cannot be split for further fine-grained batch-level parallelism.

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. FineStream responds 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 a 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 [20]. 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*.

$$t_{stage\_i}{}' = \begin{cases} t_{stage\_i} & (Bsum \leq Bmax) \\ r_i \cdot t_{stage\_i} & (Bsum > Bmax) \end{cases} \tag{1}$$

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.

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

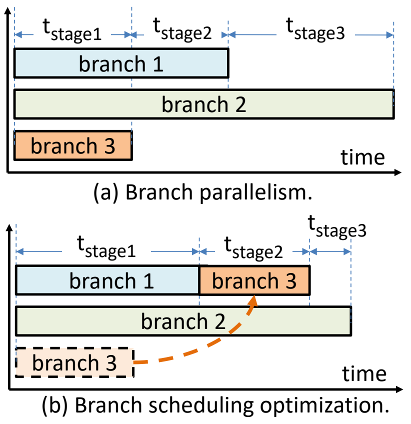


Fig. 6. An example of branch parallelism and optimization.

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 $branch2$ and $branch3$ 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.

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

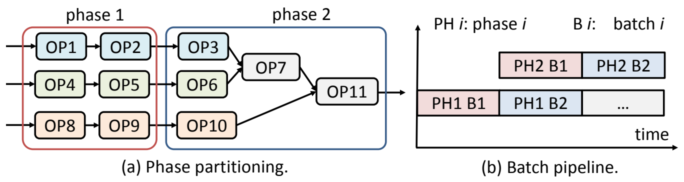


Fig. 7. An example of partitioning phases for batch pipeline.

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 7.3 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 a batch pipeline. For example, in Figure 7, the branches in *phase1* can be co-run 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 bound 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 the 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 *query plan adjustment* and *resource reallocation* are necessary for FineStream handling dynamic workload.

Query Plan Adjustment. With reference to [29], 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.

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.

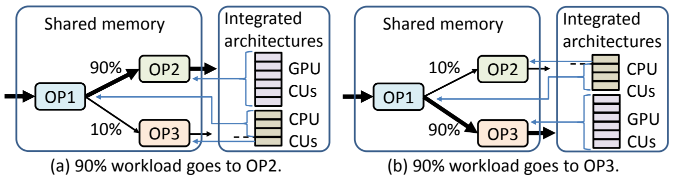


Fig. 8. An example of adjustment for dynamic workload.

In detail, for the ingestion-rate-falling branch (data arrival rate of this branch is decreasing) [29], we assume that the initial ingestion rate is r_{init}, while the current ingestion rate is r_{curr}. 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 resources in OP2 and redistribute the rest to OP3 after processing the current batch.

$$resource_{redistribute\_i} = \frac{r_{init} - r_{curr}}{r_{init}} \cdot resource_{OPi} \quad (10)$$

6 Multi-Query Support

In practice, users may need multi-queries. Additionally, if we can support different streams, the application scope of the system will be broader. In this section, we divide multi-query situations into single-stream and multi-stream scenarios, and show our design in both cases. Furthermore, we also consider the dynamic workload in multi-query support.

6.1 Single Stream

We discuss multi-queries in single stream in this part. It is common for multiple users to submit different queries for the same stream to the system. To support multi-queries in single stream, we need to handle the following three difficulties.

Difficulties. First, the intermediate results for different queries belong to separate user spaces. Second, the modeling, especially the computing resource distribution, becomes even more complicated because of different queries. Third, the complexities among queries could be different, which need to be involved in system design.

Design. Our design is as follows. First, we organize the operators from different queries into the same DAG, so that we do not need to consider separate user spaces. Second, we use the performance model in Section 5 to distribute the computing resources to operators since all operators are organized into the same DAG. Third, when a stream of tuples enters the system, multi-queries are conducted by such uniformed design.

Optimization. To further improve the performance in supporting multi-queries of a single stream, we conduct a novel optimization, called *operator-sharing*. In operator-sharing, we merge the initial paths of different queries as much as possible, so that multiple queries could share the same computing results. This design avoids extra space cost and computation caused by multiple queries. We show an example of *active fusion* in Figure 9. Figure 9 (a) shows the original example of two queries. They have the same operators *OP1* and *OP2*. To save unnecessary computation, we merge their first two operators *OP1* and *OP2* of these two queries and postpone the split process after *OP2*. FineStream detects the workload changes and periodically applies sharing-aware query optimization. When two users submit queries simultaneously, the operator sharing optimization still can be applied. In real-world cases, two users can submit different queries at the same time. For example, in stock market analysis of event stream processors, many queries can be submitted to the system on the same data stream simultaneously, even with the interests on the same stock patterns [30]. Hence, the probability that the stock market stream system processes multiple queries at the same time is very high.

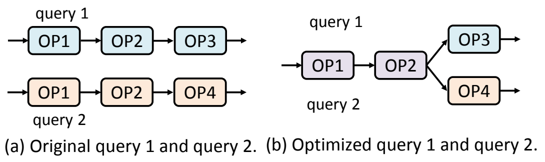

query 1
OP1
OP2
OP3
OP1
OP2
OP4
query 2
(a) Original query 1 and query 2.
query 1
OP1
OP2
OP3
OP4
query 2
(b) Optimized query 1 and query 2.

Fig. 9. Illustration for operator-sharing optimization.

6.2 Multi-Stream

Multi-queries on multiple streams are common in practice, and we enable FineStream process queries in multi-stream situations. In supporting multiple streams, a common question is that when multiple queries include the same operator, whether we can reuse the same operator for different streams to save computing resources. We find that this is hard to be optimized. The difficulties are as follows.

Difficulties. First, tuples of different streams may have different schemas. Merging streams with different schemas could incur large performance overhead. Second, even for streams with the same schemas, FineStream processes data at batch granularity. We need to separate the tuples belonging to different streams. Third, merging streams also needs to involve information indicating which stream each tuple belongs to, which causes overhead.

Design. Based on the above analysis, we adopt a separate design for multiple streams. We use a monitor thread to detect the ingestion rates of different input streams for resource allocation, as shown in Figure 10. Our design is as follows. First, the monitor thread identifies the input streams, and collect necessary performance data. Second, based on the gathered performance data, FineStream performs unified performance modeling for processing different streams. Third, FineStream distributes computing resources to the operators of queries targeting different streams.

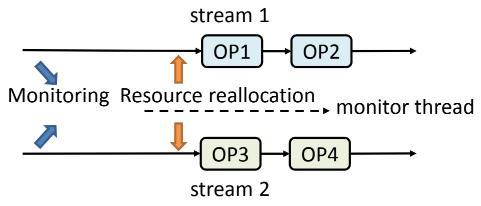


Fig. 10. Multi-stream monitoring.

6.3 Dynamic Situation

Dynamic workloads also happen in multi-query situations of both single-stream and multi-stream cases.

Difficulties. The major difficulties in a dynamic situation lie in how to detect the dynamic changes in both single and multi-streams.

Single stream. In single-stream cases, dynamic workloads could appear, as discussed in Section 5.3. However, when FineStream handles multiple queries, the query plan adjustment and light-weight resource reallocation still work, because we adopt the design of organizing the operators from different queries into the same DAG, as shown in Section 5.3.

Multi-stream. The monitor thread can be used to observe dynamic workloads among different streams. Accordingly, FineStream can adjust computing resources for different streams based on the collected performance data from the monitor thread. For example, in Figure 10, when the monitor thread detects the decrease of ingestion rate in *stream 1*, it reports such phenomenon to the system and FineStream distributes part of the *stream-1* computing resources to the operators for *stream 2*.

7 Implementation Details

7.1 How FineStream Works

We present the system workflow in Figure 11. In Figure 11, *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. With the online profiling, FineStream can obtain the operator-related data that are used to build the performance model, 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. Note that when the multi-stream or dynamic workload is detected, FineStream performs related adjustment mentioned in Section 5.3.

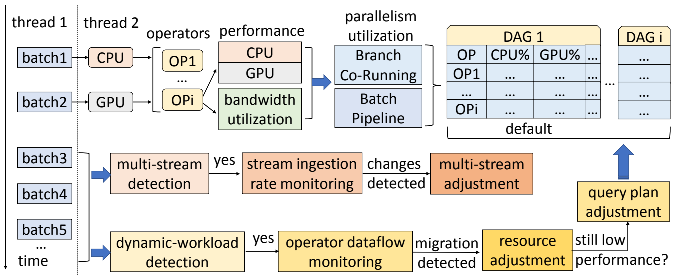


Fig. 11. FineStream workflow.

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 resources based on the parameters and the performance model.

7.2 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 (Line 2 to 5). Next, FineStream checks whether resource rescheduling is needed when multiple streams exist in the system (Line 6 to 12). When the overall performance of FineStream decreases and this is caused by the decreasing ingestion rate from one stream, FineStream reschedules the hardware resource from this stream to the others. FineStream also detects dynamic workload (Line 13 to 21). If the dynamic workload is detected, FineStream conducts the related hardware rescheduling. If such rescheduling does not help, it further conducts query plan adjustment.

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          $taskFirstRun = false$
       // Resource adjustment among
          multistreams
6      foreach stream $i$ in $FineStream.stream$ do
7          if $performance\&\&ingestionRate$ decrease
            then
8              $decreaseResource(i)$
9              $markMultiStream = true$
10     if $markMultiStream == true$ then
11         $rescheduleResource()$
12         $markMultiStream = false$
       // Handling dynamic workload and
          query plan optimization
13     if $detectDynamicWorkload()$ then
14         $resourceReallocate()$
15         if $resourceChanged == true$ then
16             if $detectDynamicWorkload()$ then
17                 $updateQueryPlan()$
18                 $queryChanged = true$
19         $resourceChanged = true$
20         if $queryChanged == true$ then
21             $resourceChanged = false$
```

7.3 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.

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 the 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. As input workload characteristics may change over time, FineStream periodically collects the aforementioned parameters at runtime and reoptimizes the query plan accordingly.

$$bandwidth_{i,dev} = \frac{s_{input\_i} + s_{output\_i}}{t_{i,dev}} \tag{11}$$

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.

7.4 Module Implementation

FineStream consists of a Java module and a C module, which is similar to Saber [4]. The Java module of FineStream is responsible for the user-defined queries and the process of analyzing and preprocessing the input stream data. The C module is implemented by OpenCL [16] to control and execute streaming queries on the CPU and the GPU of integrated architectures.

Communication. Communication between two modules is achieved by Java Native Interface (JNI). In data transmission, we use Java Disruptor to implement the resource pool for the dispatcher. When data are stored in the Disruptor buffer, threads fetch the data and execute the pseudocode in Algorithm 1. In our implementation, we integrate the operators of the same stage in one JNI method, thus reducing the JNI calling overhead.

Operators. For the operators in FineStream, we reuse the operator code from OmniDB [31]. OmniDB is a popular lightweight query processing engine. The operators are written in OpenCL, and OmniDB provides both the efficient CPU operators and the GPU operators. These operators have been widely used in many query engines, such as [32], [33], [34]. Hence, we directly reuse these operators in FineStream. 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 engines such as Voodoo [35].

8 Evaluation

Focusing on the fine-grained multi-query stream processing on integrated architectures, we evaluate FineStream in this section.

8.1 Methodology

The baseline method used in our comparison is Saber [4], 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 micro-batch processing model. 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 best single device performance for comparison, denoted as "Single" (the best performance of using only CPUs or GPUs). 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 [10], 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 third discrete platform is equipped with an Intel i7-8700K CPU and an NVIDIA GeForce GTX 1080Ti GPU, and along with 32 GB memory. The fourth discrete platform is equipped with two Intel E5-2640 v4 CPUs and an NVIDIA V100-32GB GPU, and it has 264 GB memory.

Datasets. The development of 5G thrives IoT networks and various applications, such as cluster monitoring [1], anomaly detection in smart grids [2], and road tolling systems [3], which have been proposed recently. These applications can be represented in stream processing, and thus we use these real workloads in our evaluation. We use four streaming applications (including three real-world application [4], [36], [37] and one carefully designed synthetic application) to evaluate our system. Our evaluation results have clearly demonstrated the benefits of applying stream processing in integrated CPU-GPU architectures including energy efficiency and price-throughput ratio. With NVLink and PCIe V4, our system may also be applied to discrete CPU-GPU architectures.

- The first workload is Google compute cluster monitoring [1], which comes from a cluster management scenario. Worker servers send their states to a monitoring server. The trace includes real resource-usage information from the Google production cluster, including timestamp, job ID, machine ID, event type, and so on.
- The second workload is anomaly detection in smart grids [2], which is about detection in energy consumption from different devices of a smart electricity grid. In detail, the workload is based on recordings originating from smart plugs, which are deployed in private households. The base stream includes a timestamp, value for measurement, property, plug, and so on.
- The third workload is linear road benchmark [3], which models a network of toll roads. It simulates the highway toll system for automobiles in metropolitan areas. The schema of the stream data includes a timestamp, vehicle, speed, highway, and so on.
- The fourth workload is a synthetically generated workload based on [4] for evaluating independent operators, where each tuple consists of a 64-bit timestamp and six

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

32-bit attributes drawn from a uniform distribution.

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 [4]. 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. For multi-query evaluation, we generate queries with a uniform distribution to evaluate our multi-query design. In detail, we organize *Q1* and *Q3* as multi-query group *M1*, *Q8* and *Q9* as *M2*, and *Q4* and *Q5* as *M3*. Because for *M1*, *M2*, and *M3*, the input of different queries are the same, we use these tests to measure the efficacy of handling multiple queries in a single stream. To evaluate the multi-query performance on multiple streams, we uniformly organize *Q1* and *Q8* as *M4*, *Q1* and *Q5* as *M5*, and *Q8* and *Q5* as *M6*.

Dynamic Workload Generation. We use the datasets and benchmarks to generate a dynamic workload. In the data transmitter for a stress test, we use a for-loop to continuously transmit batch size data. To control the arrival rate, we insert a *Thread.sleep()* function to control the rate of data transmission. 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 *D1*. Similar evaluations are also conducted on the second dataset of smart grid with the query *Q5*, which is denoted as *D2*, and the third dataset of linear road benchmark with the query *Q8*, which is denoted as *D3*. For the evaluation of multi-stream support, task group *D4* is a mix of query *Q1* with the first dataset and query *Q8* with the third dataset. The arrival rate of streaming data for *Q1* decreases from the original state of 322,560 to 72,533 tuples per second. Task group *D5* is a mix of *Q1* and *Q5*, and the arrival rate of the streaming data for *Q1* decreases from 331,584 to 64,573 tuples per second gradually. The task group *D6* is a mix of *Q5* and *Q8*, and the arrival rate of the streaming data for *Q5* decreases from 250,156 to 45,413 tuples per second.

We conduct model and parameter analysis in FineStream evaluation, which plays an important role in understanding the performance behavior of FineStream. Next, we present the performance results first, and then show the detailed model and parameter analysis in Section 8.4 and Section 8.5.

8.2 Handling Multiple Queries

For evaluating the performance of handling multiple queries, we compare FineStream to Saber, in which we develop the *bulk-synchronous strategy* [13] of allocating different queries to different devices without further operator-level partitioning.

Throughput. We show the throughput comparison results in Figure 12. In general, we have the following observations. First, The average performance FineStream is 744,934 tuples per second, which outperforms Saber by 32% on average. Second, the CPU of Ryzen 5 2400G has been strengthened, so the difference in computing power between CPU and GPU on this platform is relatively small. However, even under such conditions, FineStream still achieves clear performance benefits. Third, FineStream achieves performance benefits in all cases, which demonstrates the effectiveness of our method.

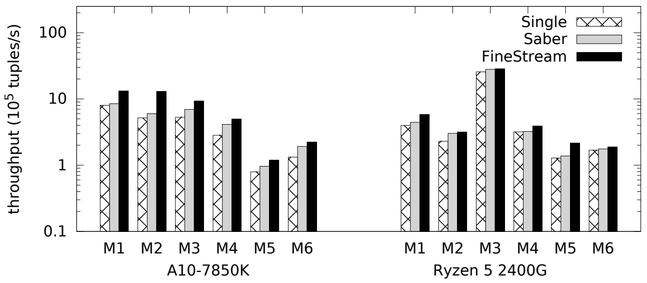


Fig. 12. Throughput of the queries on multi-queries.

Latency. We show the latency results in Figure 13. In this work, latency is defined as the end-to-end time from the time a query starts to the time it ends. FineStream achieves latency benefits in all cases. On average, FineStream achieves 28% latency lower than that of Saber. Note that the achieved latency benefit is relatively lower on Ryzen 5 2400G than that on A10-7850K. The reason is that the performance gap between Ryzen's CPU and GPU is smaller.

Dynamic Workloads. In this section, we discuss how to handle dynamic workloads. To demonstrate the capability of FineStream to handle a dynamic workload, we evaluate

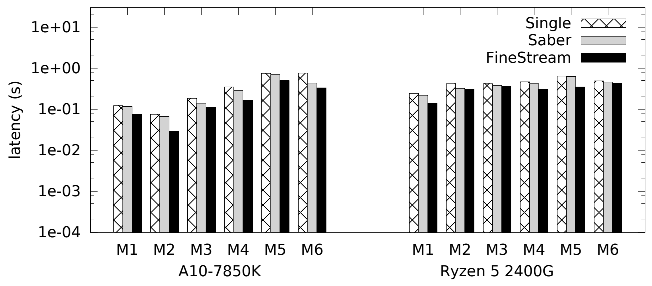
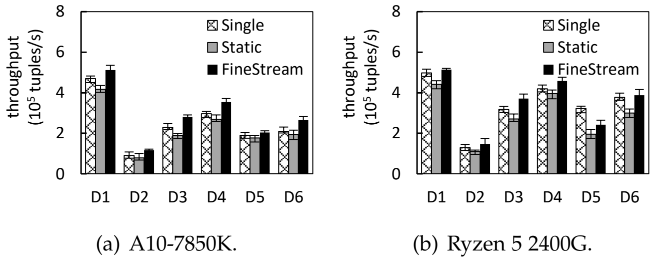


Fig. 13. Latency of the queries on multi-queries.

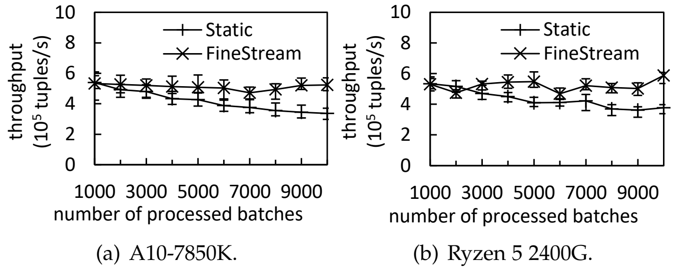
FineStream on the dynamic workloads mentioned in Section 8.1. On average, FineStream achieves a performance of 321,169 tuples per second, which outperforms the static method (we denote "static" for FineStream without adapting to dynamic workload) by 27%, as shown in Figure 14.



(a) A10-7850K. (b) Ryzen 5 2400G.

Fig. 14. Throughput of the queries on dynamic workloads.

Case Study. We use *D1* as an example, and show the detailed throughput along with the number of batches in Figure 15. In the timeline process, the static method decreases due to 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.



(a) A10-7850K. (b) Ryzen 5 2400G.

Fig. 15. Throughput of *D1* on dynamic workloads.

8.3 Single-Query Performance

Throughput. We explore the throughput of FineStream for the nine queries. Figure 16 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 over the best single device performance on average, which implies that FineStream nearly doubles the performance compared to the method of using only a single device. Compared to Saber, FineStream still achieves 52% throughput improvement, which clearly shows the advantage of fine-grained stream processing on the integrated architecture. Second, on the Ryzen 5 2400G platform, because all hardware configurations have been upgraded in comparison with A10-7850K, Saber achieves a 56% throughput improvement compared to the throughput of the best single device. However, FineStream is still 14% higher than Saber on this platform. Similar phenomena have also been observed in [20], [38].

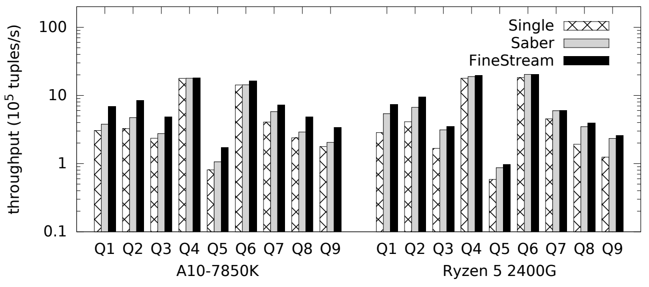


Fig. 16. Throughput of different queries.

Latency. Figure 17 reports the latency of different queries on the integrated architectures. 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.

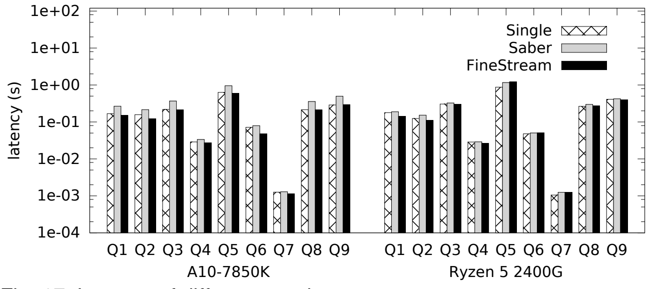


Fig. 17. Latency of different queries.

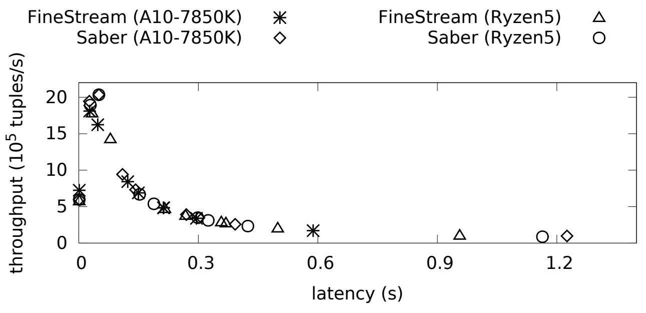


Fig. 18. Throughput vs. latency.

Profiling. We show the relationship between throughput and latency of both FineStream and Saber in Figure 18. Figure 18 shows that queries with high throughput usually have low latency, and vice versa.

We further study the CPU and GPU utilization of Saber and FineStream, and use the A10-7850K platform for illustration, as shown in Figure 19. In most cases, FineStream utilizes the GPU device better on the integrated architecture. As for $Q4$, 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.

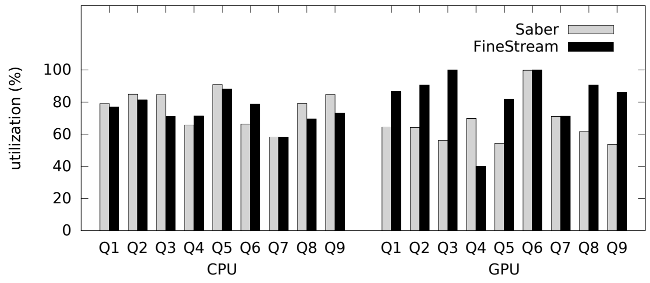


Fig. 19. Utilization of the integrated architecture.

8.4 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.

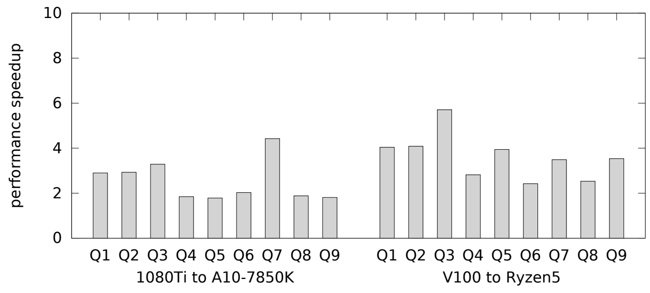


Fig. 20. Throughput comparison between Saber on discrete architectures and FineStream on integrated architectures.

Performance Comparison. The current GPU on the integrated architecture is less powerful than the discrete GPU, as mentioned in Section 2.2. The system on discrete architecture utilizes pinned memory buffer for data transfer and pipelines computation and communication to hide communication overhead. Similar ideas have also been adopted in [4], [39], [40]. Unfortunately, the communication overhead still exists since the communication time and computation time may not be the same. The discrete GPUs exhibit 1.8× to 5.7× higher throughput than the integrated architectures, as shown in Figure 20. However, as we show later, 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. Specifically, as shown in Figure 21, the operators mentioned in Table 2 are distributed in Figure 21, 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 s_{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 illustration, 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. If the t_{total} on discrete GPUs is larger than $t_{compute}$ on the integrated architectures, then FineStream on integrated architectures can achieve latency improvement.

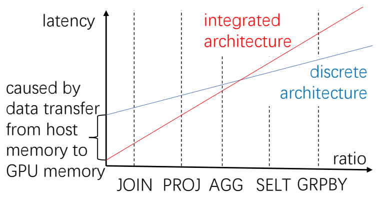


Fig. 21. Latency comparison.

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 7× higher than that of the A10-7850K integrated architecture, and the price of the V100 discrete architecture is about 64× higher than that of the Ryzen 5 2400G integrated architecture. Figure 22 shows the comparison of their price-throughput ratio. On average, FineStream on the integrated architectures outperforms Saber on the discrete architectures by 10.4×. Note that in application scenarios with high performance requirements, the acquisition cost may not be the most important factor. However, the integrated architectures can still be regarded as a compromise between performance and cost with a wide range of application scenarios.

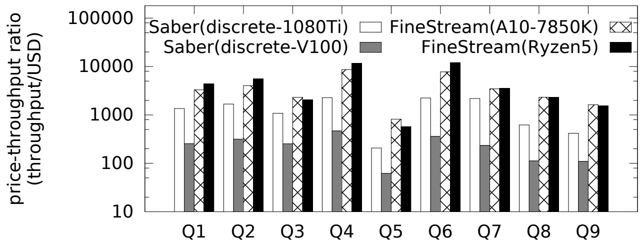


Fig. 22. Comparison of price-throughput ratio.

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. The energy efficiency is defined as throughput divided by TDP, which reflects the performance-per-power results. We show the energy efficiency of FineStream and Saber in Figure 23. On average, FineStream on the integrated architectures is 1.8× energy-efficient than Saber on the discrete architectures.

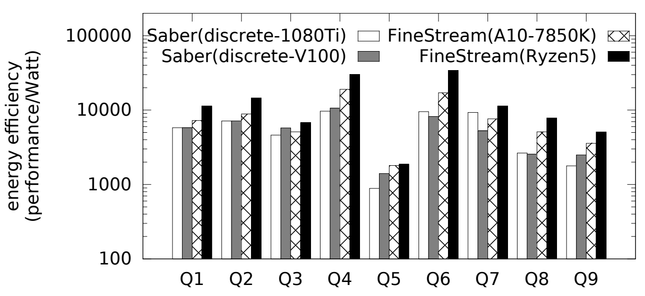
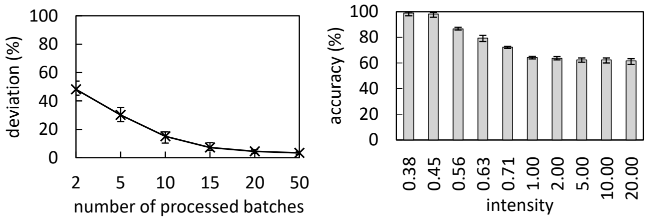


Fig. 23. Comparison of energy efficiency.

8.5 Model Analysis



(a) Deviation. (b) Intensity vs accuracy.

Fig. 24. Model analysis for *Q1* on Ryzen 5 2400G.

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 24. 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 workloads, the accuracy also depends on the intensity of workload changes. The intensity is defined as the sum of the absolute differences in the amount of data arrival per unit time. In FineStream, the minimum time interval is 0.05s. We sum the absolute values of the change in the amount of arrival data in every minimum time interval $|\Delta V|$ together, and then divide the sum by the time to get the intensity, as shown in Equation 12. In our evaluation, the greater the intensity, the lower the prediction accuracy. We use the example of *Q1* for illustration, as shown in Figure 24 (b). When the intensity is less than 0.45, the accuracy is higher than 98%. However, when the intensity is greater than 0.56, the accuracy decreases.

$$intensity = \frac{\sum_{t=0}^{range-1} |\Delta V|}{t_{range}} \tag{12}$$

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 workloads, which causes time overhead. Second, multistream detection 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.

8.6 Detailed Analysis

In this section, we perform a detailed analysis of the factors that can influence performance. These factors include the number of attributes, slide size, window size, and batch size of a query. We use *Q1* for illustration. The performance impact is shown in Figure 25, and the latency impact is shown in Figure 26. We omit the results for the other queries because they exhibit similar trends. Detailed analysis is as follows.

Number of attributes. The number of attributes has a small but observable impact on system performance. Figure 25 (a) shows the results of throughput, and Figure 26 (a) shows the results of latency. To explore the influence of the number of attributes on the throughput and latency, we vary the number of attributes in tuples of *Q1* and keep the other factors the same.

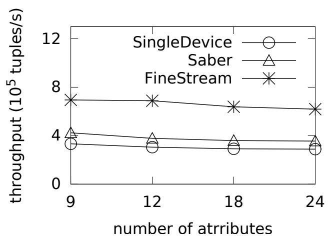
- *Observation*. From Figure 25 (a), we can see the trend of throughput declining as the number of attributes in tuple increases. Figure 26 (a) shows that the latency also increases.
- *Insight*. More attributes mean that the system needs to read more input data, which results in more data processing time, and increases the latency.

Window size. We explore the influence of window size on throughput and latency in Figure 25 (b) and Figure 26 (b). We use a block of threads to process a window in a batch.

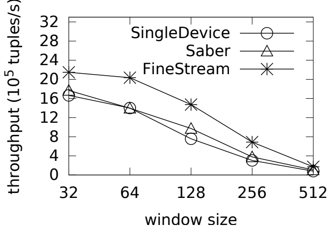
- *Observation*. As the window size becomes larger, the throughput decreases, and the latency increases. In detail, when the window size is less than 64, the performance curve changes relatively smoothly. When it is greater than 64, the performance curve changes drastically.
- *Insight*. The throughput and latency behavior mainly relates to two factors: parallelism and execution time of operators. First, in FineStream (also Saber), we use a block of threads to process a window in a batch. If the window size is too small, such as 32, the thread block is not large enough to release the GPU power. Second, in *Q1*, the execution time of *group-by* operator is long. A larger window size requires more time to group data.

Slide size. Next, we investigate the impact of slide size on performance. We vary the slide size of *Q1*, and the other factors remain the same.

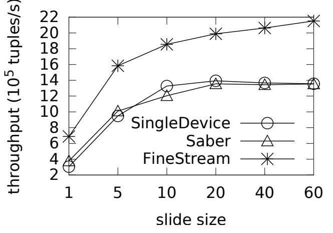
- *Observation*. Figure 25 (c) and Figure 26 (c) show that, along with the slide size, the throughput increases and the latency decreases. When the slide size is larger than ten, the performance curve does not change drastically.
- *Insight*. In our evaluation, we use the number of processed tuples from the stream as the evaluation indicator and do not repeatedly count the tuples in overlapping areas of different windows. A slide size larger than ten can make a batch be processed faster, so the latency becomes shorter.



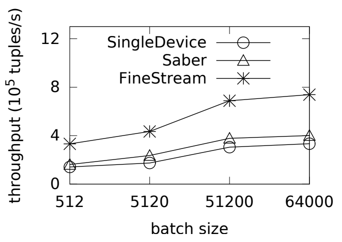
(a) Different number of attributes.



(b) Different window sizes.

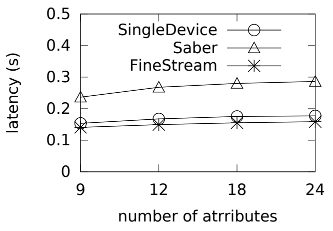


(c) Different slide sizes.

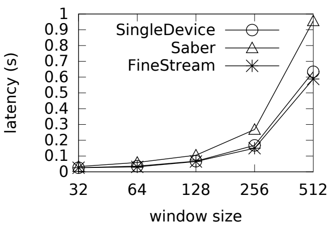


(d) Different batch sizes.

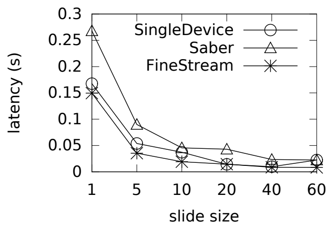
Fig. 25. Performance impact of the number of attributes, slide size, window size, and batch size for *Q1*.



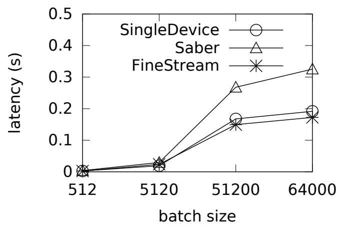
(a) Different number of attributes.



(b) Different window sizes.



(c) Different slide sizes.



(d) Different batch sizes.

Fig. 26. Latency impact of the number of attributes, slide size, window size, and batch size for *Q1*.

Batch size. After we discuss the characteristics of slides and windows, we further explore the batch influence.

- *Observation*. Figure 25 (d) reports the throughput influence of batch size, and Figure 26 (d) reports the latency influence. When the batch size is small, the throughput and latency increase with batch size. However, when the batch size reaches 51200, the performance has reached a steady state.
- *Insight*. Batches are the actual processing data units in the system. The larger batch size is good for throughput because more data can be processed in parallel. When the batch reaches a limit, the performance tends to flatten.

9 Related Work

The closest work to FineStream is Saber [4], which aims to utilize discrete CPU-GPU architectures. Saber [4] adopts the micro-batch processing model, 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 integrated architectures, FineStream adopts continuous operator (CO) model [22], 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. Although these techniques are standard, applying these techniques to integrated architectures, particularly in stream processing situations, involves several challenges to handle. For example, we need to optimize a query plan with shared memory and make adjustments for dynamic workloads. It is noteworthy that our fine-grained operator placement is different from classical placement strategies for general stream processing [36], [41], [42], [43] 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 takes device preference into consideration during placement optimization, which has not been considered before in stream processing frameworks [13].

Parallel stream processing [4], [44], [45], [46], [47], [48], [49], [50], [51], query processing [52], [53], [54], [55], [56], [57], and heterogeneous systems [35], [58], [59], [60], [61], [62], [63], [64], [65], [66] 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. [8] presented a stream processing algorithm considering various latency and throughput requirements on GPUs. Alghabi et al. [23] developed a framework for stateful stream data processing on multiple GPUs. De Matteis et al. [24] developed Gasser system for offloading operators on GPUs. Pinnecke et al. [67] studied how to efficiently process large windows on GPUs. Chen et al. [26] extended the popular stream processing system, Storm [68], to GPU platforms. FineStream differs from those previous works in two aspects: firstly on integrated architectures, and secondly for SQL streaming processing. Besides the integrated architecture, FineStream may be also well suited to discrete architectures in the future. In particular, more efficient CPU-GPU interconnects have been developed, such as NVLink and PCIe v4. FineStream's fine-grained operator placement strategy would benefit from the significantly reduced data transmission overhead in those discrete architectures.

10 Conclusion

Although stream processing has shown significant performance benefits on GPUs, 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 multi-query 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 stream processing, in comparison with discrete architectures. FineStream achives significant performance benefits over the state-of-the-art method. Experiments show that FineStream can improve the performance by 52% over Saber on the integrated architecture. For multi-query processing, FineStream achieves 32% throughput improvement compared to applying different devices for multiple queries in a coarse-grained method. Compared to the stream processing engine on the discrete architecture, FineStream on the integrated architecture achieves 10.4× price-throughput ratio, 1.8× energy efficiency, and can enjoy lower latency benefits.

Acknowledgment

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, and 61972403), and Beijing Natural Science Foundation (No. L192027). This work is also supported by a MoE AcRF Tier 1 grant (T1 251RES1824), Tier 2 grant (MOE2017-T2-1-122), and SUTD Start-up Research Grant (SRT3IS21164) in Singapore. Xiaoyong Du is the corresponding author of this paper.

References

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

[2] H. Ziekow and Z. Jerzak, "The DEBS 2014 grand challenge," in *DEBS*, 2014.

[3] A. Arasu, M. Cherniack *et al.*, "Linear road: a stream data management benchmark," in *PVLDB*, 2004.

[4] A. Koliousis and et al., "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD*, 2016.

[5] G. Theodorakis, P. R. Pietzuch, and H. Pirk, "SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries," in *EDBT*, 2020.

[6] K. Wang, K. Zhang *et al.*, "Concurrent analytical query processing with GPUs," *PVLDB*, 2014.

[7] J. Nickolls and W. J. Dally, "The GPU computing era," *Micro*, 2010.

[8] U. Verner, A. Schuster, and M. Silberstein, "Processing data streams with hard real-time constraints on heterogeneous systems," in *ICS*, 2011.

[9] D. Boggs, G. Brown *et al.*, "Denver: Nvidia's First 64-bit ARM Processor," *Micro*, 2015.

[10] D. Bouvier and B. Sander, "Applying AMD's Kaveri APU for heterogeneous computing," in *Hot Chips: A Symposium on High Performance Chips (HC26)*, 2014.

[11] J. Doweck, W. Kao *et al.*, "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," *Micro*, 2017.

[12] F. Zhang, L. Yang *et al.*, "FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures," in *USENIX ATC*, 2020.

[13] S. Zhang, F. Zhang *et al.*, "Hardware-conscious stream processing: A survey," *SIGMOD Rec.*, 2020.

[14] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, 2006.

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

[16] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, 2010.

[17] M. J. Schulte, M. Ignatowski *et al.*, "Achieving exascale capabilities through heterogeneous computing," *Micro*, 2015.

[18] T. Vijayaraghavan, Y. Eckert *et al.*, "Design and Analysis of an APU for Exascale Computing," in *HPCA*, 2017.

[19] D. Bouvier and B. Sander, "Applying AMD's Kaveri APU for heterogeneous computing," in *Hot Chips Symposium*, 2014.

[20] F. Zhang, J. Zhai *et al.*, "Understanding co-running behaviors on integrated cpu/gpu architectures," *TPDS*, 2017.

[21] Y. Go, M. A. Jamshed *et al.*, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *NSDI*, 2017.

[22] S. Venkataraman, A. Panda *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *SOSP*, 2017.

[23] F. Alghabi, U. Schipper, and A. Kolb, "A scalable software framework for stateful stream data processing on multiple gpus and applications," in *GPU Computing and Applications*, 2015.

[24] T. De Matteis, G. Mencagli *et al.*, "GASSER: An Auto-Tunable System for General Sliding-Window Streaming Operators on GPUs," *IEEE Access*, 2019.

[25] K. Zhang, J. Hu, and B. Hua, "A holistic approach to build real-time stream processing system with GPU," *JPDC*, 2015.

[26] Z. Chen, J. Xu *et al.*, "G-Storm: GPU-enabled high-throughput online data processing in Storm," in *Big Data*, 2015.

[27] C. HewaNadungodage, Y. Xia, and J. J. Lee, "GStreamMiner: a GPU-accelerated data stream mining framework," in *CIKM*, 2016.

[28] S. Touati and B. D. De Dinechin, *Advanced Backend Code Optimization*. John Wiley & Sons, 2014.

[29] B. Gedik, S. Schneider *et al.*, "Elastic scaling for data stream processing," *TPDS*, 2013.

[30] S. Zhang, H. T. Vo *et al.*, "Multi-query optimization for complex event processing in SAP ESP," in *ICDE*, 2017.

[31] S. Zhang, J. He *et al.*, "OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures," *PVLDB*, 2013.

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

[33] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *PVLDB*, 2014.

[34] J. Paul, J. He, and B. He, "GPL: A GPU-based pipelined query processing engine," in *SIGMOD*, 2016.

[35] H. Pirk, O. Moll *et al.*, "Voodoo-a vector algebra for portable database performance on modern hardware," *PVLDB*, 2016.

[36] R. Castro Fernandez, M. Migliavacca *et al.*, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013.

[37] I. S. Moreno, P. Garraghan *et al.*, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," *IEEE Transactions on Cloud Computing*, 2014.

[38] F. Zhang, B. Wu *et al.*, "Automatic Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures," *TKDE*, 2019.

[39] M. Gowanlock and B. Karsin, "A hybrid CPU/GPU approach for optimizing sorting throughput," *Parallel Computing*, vol. 85, pp. 45–55, 2019.

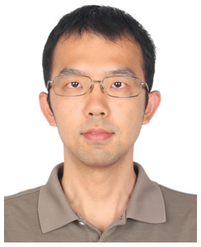
[40] Y.-M. N. Nam, D. H. Han, and M.-S. K. Kim, "SPRINTER: A Fast n-ary Join Query Processing Method for Complex OLAP Queries," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2055–2070.

[41] P. Carbone, A. Katsifodimos *et al.*, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[42] L. Neumeyer, B. Robbins *et al.*, "S4: Distributed stream computing platform," in *ICDM Workshops*, 2010.

[43] A. Toshniwal, S. Taneja *et al.*, "Storm@ twitter," in *SIGMOD*, 2014.

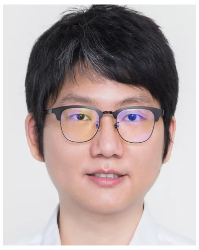
[44] P. Pietzuch, J. Ledlie *et al.*, "Network-aware operator placement for stream-processing systems," in *ICDE*, 2006.
[45] B. Chandramouli, J. Goldstein *et al.*, "Accurate latency estimation in a distributed event processing system," in *ICDE*, 2011.
[46] K. Bhardwaj, P. Agrawal *et al.*, "Appflux: Taming app delivery via streaming," *Proc. of the Usenix TRIOS*, 2015.
[47] R. Ben-Basat, G. Einziger *et al.*, "Heavy hitters in streams and sliding windows," in *INFOCOM*, 2016.
[48] N. Agrawal and A. Vulimiri, "Low-Latency Analytics on Colossal Data Streams with SummaryStore," in *SOSP*, 2017.
[49] P. Fernando, A. Gavrilovska *et al.*, "NVStream: accelerating HPC workflows with NVRAM-based transport for streaming objects," in *HPDC*, 2018.
[50] X. Fu, T. Ghaffar *et al.*, "Edgewise: a better stream processing engine for the edge," in *USENIX ATC*, 2019.
[51] S. Zhang, B. He *et al.*, "Revisiting the design of data stream processing systems on multi-core processors," in *ICDE*, 2017.
[52] T. F. Wenisch, M. Ferdman *et al.*, "Practical off-chip meta-data for temporal memory streaming," in *HPCA*, 2009.
[53] B. Chandramouli, J. Goldstein *et al.*, "Trill: A high-performance incremental query processor for diverse analytics," *PVLDB*, 2014.
[54] B. Chandramouli, R. C. Fernandez *et al.*, "Quill: efficient, transferable, and rich analytics at scale," *PVLDB*, 2016.
[55] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the Memory Wall in MonetDB," *Commun. ACM*, 2008.
[56] S. Breß and G. Saake, "Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS," *PVLDB*, 2013.
[57] P. Bakkum and K. 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.
[58] M. R. Meswani, S. Blagodurov *et al.*, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *HPCA*, 2015.
[59] Z. Tang and Y. Won, "Multithread content based file chunking system in CPU-GPGPU heterogeneous architecture," in *2011 First International Conference on Data Compression, Communications and Processing*, 2011.
[60] A. M. Merritt, V. Gupta *et al.*, "Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies," in *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, 2011.
[61] M. Silberstein, S. Kim *et al.*, "GPUnet: Networking abstractions for GPU programs," *TOCS*, 2016.
[62] S. Bergman, T. Brokhman *et al.*, "SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs," in *USENIX ATC*, 2017.
[63] Q. Cao, N. Balasubramanian, and A. 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.
[64] P. Chrysogelos, M. Karpathiotakis *et al.*, "HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines," *PVLDB*, 2019.
[65] L. Liu, S. Yang *et al.*, "Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems," *TPDS*, 2019.
[66] T. D. Doudali, S. Blagodurov *et al.*, "Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence," in *HPDC*, 2019.
[67] M. Pinnecke, D. Broneske, and G. Saake, "Toward GPU Accelerated Data Stream Processing." in *GvD*, 2015.
[68] "Apache Storm," http://storm.apache.org/.



Feng Zhang received the bachelor degree from Xidian University in 2012, and the PhD degree in computer science from Tsinghua University in 2017. He is an assistant professor with the Key Laboratory of Data Engineering and Knowledge Engineer (MOE), Renmin University of China. His major research interests include high performance computing, heterogeneous computing, and parallel and distributed systems.

Chenyang Zhang is an undergraduate in School of Information, Renmin University of China. She joined the Key Laboratory of Data Engineering and Knowledge Engineer (MOE) in 2019. Her research interests include high performance computing, heterogeneous computing, and parallel accelerating.

Lin Yang received the bachelor degree from Chengdu University of Technology in 2017. He is a graduate student in the School of information, Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineer (MOE) in 2017. His major research interests include high performance computing, heterogeneous computing, and parallel and distributed systems.



Shuhao Zhang received the bachelor degree in computer engineering from Nanyang Technological University in 2014, and the PhD degree in computer science in National University of Singapore in 2019. He is currently an Assistant Professor at the Singapore University of Technology and Design. His research interests include high performance computing, stream processing systems, and database systems.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Wei Lu received the PhD degree in computer science from the Renmin University of China, in 2011. He is currently an associate professor at the Renmin University of China. His research interests include query processing in the context of spatiotemporal, cloud database systems, and applications.

Xiaoyong Du obtained the B.S. degree from Hangzhou University, Zhengjiang, China, in 1983, the M.E. degree from Renmin University of China, Beijing, China, in 1988, and the Ph.D. degree from Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.