Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

dStream: An Online-based Dynamic Operator-level Query Mapping Scheme on Discrete CPU-GPU Architectures

GYEONGHWAN JUNG[1], YEONWOO JEONG[1], KYULI PARK[1], DONGJAE LEE[1], HONGSU BYUN[1], SUYEON LEE[2], SUNGYONG PARK[1](Member, IEEE)
[1]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea
[2]Department of Computer Science, Georgia Institute of Technology, GA 30332, USA

Corresponding author: Sungyong Park (e-mail: parksy@sogang.ac.kr).

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2024-00453929) (RS-2024-00416666). This work was supported by the IITP(Institute of Information Communications Technology Planning Evaluation)-ITRC(Information Technology Research Center(IITP-2024-RS-2023-00259099, 30%) grant funded by the Korean government(Ministry of Science and ICT).

ABSTRACT In streaming systems with a discrete CPU-GPU architecture, leveraging the strengths of both processing units can significantly improve query performance. Existing studies assign entire queries to either the CPU or GPU to reduce data transfer overhead and boost throughput. However, this coarse-grained approach can limit performance for two main reasons. Firstly, PCIe transfer overhead is minimal for small data sizes, and the device preference of each operator within a query may change with variations in data size. Secondly, it neglects performance fluctuations based on the placement location of consecutive operators within the device. To address these issues, we propose dSTREAM, a distributed stream processing system that dynamically maps queries at the operator level on discrete CPU-GPU architectures. dSTREAM adapts to runtime conditions by selecting the optimal device for each operator dynamically. Through dynamic operator-level query mapping without prior knowledge, dSTREAM consistently achieves high performance. Extensive evaluation has confirmed that dSTREAM enhances average throughput by up to 45% and reduces average latency by up to 42.5% across various types of stream SQL queries, regardless of traffic types.

INDEX TERMS Query planning, data stream processing, heterogeneous architectures

I. INTRODUCTION

With the explosive growth of data, large-scale stream processing applications such as traffic congestion monitoring [1], real-time IoT data processing [2], real-time alarming [3], and real-time advertising [4] have gained significant attention. To enable scalable and efficient stream processing, a variety of distributed stream processing systems (DSPS) such as Spark Streaming [5], Storm [6], Flink [7], Heron [8], and Samza [9] have emerged. These DSPS achieve high performance by distributing data across shared clusters and processing it in parallel.

In recent years, modern server hardware has evolved to become heterogeneous, incorporating multi-core CPUs and hardware accelerators such as GPUs. Furthermore, the latest generation of the peripheral component interconnect express (PCIe) interface such as PCIe 5.0 [10] has become commonplace on modern motherboards. Additionally, low-cost commodity GPU cards that provide computing performance similar to server GPU cards are accessible to many industries. Thanks to these features, discrete CPU-GPU architectures, where two types of processors have separate memory spaces, have been adopted as an attractive environment for accelerating stream query processing in DSPS. Given that query performance in DSPS heavily depends on the device to which the query is mapped, inefficient query mapping can degrade overall performance. Determining the most appropriate execution device for a query poses a critical challenge [11]–[16].

To tackle this challenge, a wide range of studies have been proposed to identify the optimal computing device for query execution. These studies can be broadly categorized into two primary approaches. The first approach is coarse-grained mapping [17]–[20], which assigns entire queries to either the CPU or GPU. This approach eliminates data transition overhead (i.e., PCIe transfer time) by mapping queries to a single device. However, it overlooks the unique device preference of each operator within a query, thus limiting the potential performance gain from placing the operator on the ideal device. The second approach is fine-grained mapping

[21], [22], which selectively assigns each operator to an appropriate device based on a simple cost model or predetermined device preference. Among them, FineStream [21] performs operator-level mapping on integrated CPU-GPU architectures, where both the CPU and GPU are integrated on the same chip. Leveraging the fact that there is no data transition overhead via PCIe bus, FineStream independently places each operator on a suitable device. However, this study is not directly applicable to dedicated architectures, which inherently involve data transition overhead. Crystal [22] is another fine-grained approach based on the assumption that the device preference of each operator does not change during runtime, which may be unrealistic in practice.

Through an experimental study, we identified two main factors that affect the device preference of each operator.

Dynamic device preference of operator by data size. In streaming environments, data size can vary due to fluctuating traffic. Consequently, the data size processed by operators also fluctuates. For operators that only split datasets and perform calculation without conditional branches, GPU affinity increases as the data volume grows. However, as the data size becomes extremely small, processing with CPUs may be faster without copying the data to GPU memory via PCIe bus. Furthermore, we also observed that data transition overhead in discrete CPU-GPU architectures is not as severe as expected up to a certain data size and has a negligible impact on the overall performance. This suggests that the preferred device for each operator can shift based on the dynamic data size at runtime. In other words, there is a potential opportunity to map each operator to the heterogeneous device that provides the best performance for the specific data size encountered.

Location of consecutive operators varies performance. DSPS transform data to specific structures and store it in memory (i.e., RDD in Spark [5], DataStream in Flink [7], Parquet [23]) during query execution. This in-memory data format allows subsequent operators to utilize data locality, thereby achieving better performance when mapped to the same device. However, this data layout is only effective when the operator is mapped to the same device as the previous operator. For example, if an operator is executed on the CPU and the subsequent operator is mapped to the GPU, it is necessary to convert the data structure to a column-friendly format [24]. This data transition degrades query performance, resulting in inferior performance compared to a case where successive operators are mapped to a single device [25], [26].

Based on the above findings, this paper introduces an online-based dynamic operator-level query mapping scheme designed for discrete CPU-GPU architectures. This scheme dynamically assigns the operators to the optimal device based on a variety of conditions, thereby enhancing query performance. Specifically, we present the following key techniques. Firstly, we estimate the execution cost of an operator by considering the dynamic data size and the execution position of the preceding operators. Secondly, we propose a lightweight device mapper to determine the optimal query execution plan for mapping each operator to an appropriate device. Thirdly, we implement a dynamic operator-level mapping scheme in an online manner without any prior knowledge to specify the device preference of an operator.

We implemented dSTREAM with Spark Streaming [5]. To validate its effectiveness, we conducted comprehensive experiments with real-world workloads under various traffic scenarios. dSTREAM outperforms alternative methods, significantly increasing the average throughput by up to 45% and reducing the average latency per query by up to 42.5%.

The contributions of this paper are summarized as follows.

- dSTREAM identified that the input data size and the location of the consecutive operators affected the dynamic device preference of each operator (Section II-D).
- dSTREAM presented a dynamic operator-level query mapping scheme for discrete CPU-GPU architectures (Section III), adaptable to any DSPS at runtime, with the goal of reducing latency and enhancing throughput.
- dSTREAM performed operator-level query mapping in an online manner without interrupting query processing (Section III-B).
- We implemented dSTREAM (Section IV) on a real system and evaluated its performance using real-world workloads (Section V).

II. BACKGROUND & MOTIVATION

DSPS support SQL semantics, enabling users to execute sophisticated data transformations, aggregations, and analytics. This section outlines the SQL semantics [27] commonly employed across a range of DSPS. Furthermore, we discuss the limitations of existing studies and report several observations that drive the fundamental concepts behind dSTREAM.

A. DISTRIBUTED STREAMING PROCESSING SYSTEM

1) SQL Semantics

Spark SQL [27] is a Spark module that allows users to manipulate data by executing SQL queries. The query execution plan in Spark SQL is generated based on user code. Below is an example of a query to compute the frequency of each word.

```
var fileInput = spark.readStream
                 .csv("file_path")
var wordCountQuery = fileInput
             .flatMap(line => line.split(" "))
             .map(word => (word,1))
             .reduceByKey(_ + _)
```

Code 1. Query example in Spark SQL (WordCount query)

In this example, a query is composed of three SQL APIs (i.e., flatMap, map, and reduceByKey). Spark SQL analyzes these APIs, matching them with suitable SQL operators such as *filter* and *aggregate*. Each operator serves as a fundamental unit for data manipulation. Consequently, users write various queries composing multiple operators to obtain the desired results.

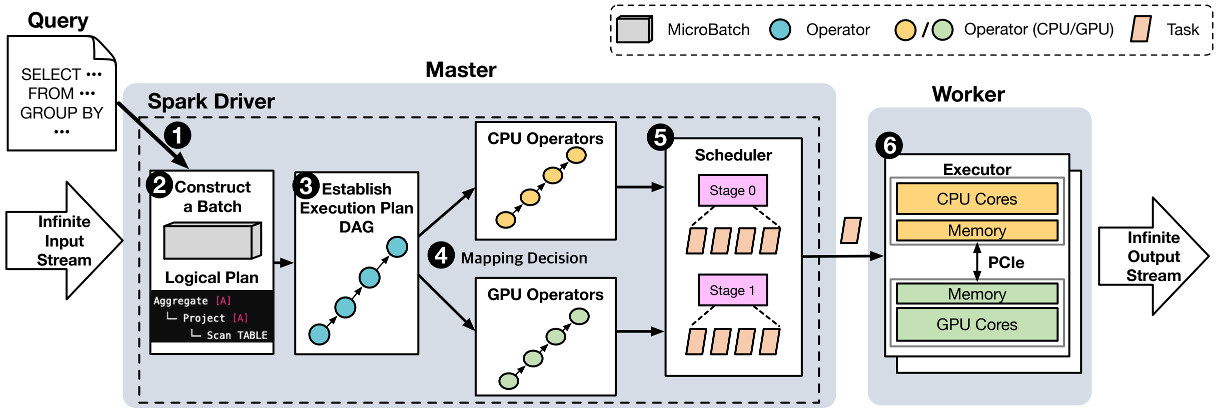


FIGURE 1. Flow of Spark Streaming on discrete CPU-GPU architectures.

2) Directed Acyclic Graph for Query Scheduling

A Directed Acyclic Graph (DAG) delineates the sequence of operations necessary for data processing by transforming each query into a structured execution plan. Each query is decomposed into multiple stages, where each stage consists of a set of tasks that can be executed in parallel. Nodes in the DAG represent a set of tasks, and edges establish dependencies between stages. This structure enables Spark to enhance query execution by minimizing data shuffling and optimizing resource utilization. Additionally, the DAG supports task lineage tracking, which is critical for fault tolerance, allowing Spark to recompute only the tasks affected by failures, rather than restarting the entire job.

3) Spark Execution Model on Heterogeneous Architectures

Spark Streaming [5] is a scalable and fault-tolerant stream processing engine capable of processing real-time data streams from various sources. Figure 1 illustrates the interaction between the master and worker node in Spark Streaming during the query execution process.

Upon query submission ❶, the master node launches a Spark driver, which subsequently launches executors on worker nodes. The Spark driver creates an execution plan and coordinates with these executors to process the given query. After initializing all necessary processes for query execution, the streaming engine starts collecting datasets from data sources. The engine buffers data over a specified period to construct a *micro-batch* ❷, which serves as a unit of execution and represents a portion of the entire dataset.

Once the micro-batch construction is completed, the driver analyzes the submitted query and generates a logical plan that defines the steps required to achieve the desired results, without detailing how each operator in the query will be executed. First, the driver parses the query to check for syntax errors and creates an initial logical plan, which may reference unvalidated columns and tables. Then, it verifies these references, producing a resolved logical plan. Finally, the driver optimizes the plan, such as by removing unnecessary columns, to create the final logical plan.

After the logical plan is finalized, the driver creates a physical plan ❸, which outlines how the query will be executed in the distributed computing environment. Based on this physical plan, each operator is assigned to run on either the CPU or the GPU, forming the final execution plan.

The current stream processing engine uses a coarse-grained mapping approach where all operators in the DAG are assigned to a single device ❹. Next, the scheduler divides the selected execution plan into a series of stages to distribute the datasets ❺. Each stage consists of multiple tasks that execute identical computations, which is the smallest unit of execution. These tasks are subsequently assigned to run on either the CPU or the GPU on the executor to perform the computations ❻. Upon completion of tasks within each stage, the results are transmitted back to the driver. This process is repeated until the application terminates.

B. DEVICE PREFERENCE OF OPERATOR

GPUs are commonly employed as co-processors alongside CPUs in various practical applications, each offering distinct advantages. CPUs optimize serial performance by utilizing multiple high-complexity cores and extensive caches, with the goal of maximizing implicit instruction-level parallelism (ILP). On the other hand, GPUs focus on maximizing data parallelism by executing massive numbers of threads simultaneously, utilizing thousands of lightweight cores. Due to these architectural features, each operator within a query exhibits a preference for a specific processor [21], [28].

For example, the *aggregate* operator, which includes functions such as sum, average, and count, can take advantage of the massively parallel architecture of GPUs. This architecture allows for the simultaneous execution of a large number of simple operations across multiple processing units. As a consequence, GPUs typically deliver superior performance compared to CPUs, and this performance advantage tends to scale with increasing data sizes. In contrast, the *filter* operator makes decisions based on conditional statements that depend on data dependencies and complex control flow. CPUs are well-suited for handling conditional branches due to their highly flexible execution pipelines and sophisticated branch prediction mechanisms.

TABLE 1. Summary of previous work on query mapping on heterogeneous architectures.

Previous work	CPU-GPU Architecture	Mapping Granularity	Device Preference	Prior Knowledge
Spark-Rapids [17]	Discrete	Coarse-grained	Static	Yes
GFlink [18]	Discrete	Coarse-grained	Static	Yes
G-Storm [19]	Discrete	Coarse-grained	Static	Yes
SABER [20]	Discrete	Coarse-grained	Dynamic	No
FineStream [21]	Integrated	Fine-grained	Static	No
Crystal [22]	Discrete	Fine-grained	Static	Yes
dStream(Ours)	Discrete	Fine-grained	Dynamic	No

C. RELATED WORKS

To improve query performance on heterogeneous architectures, several studies have proposed methods to determine the most appropriate device for query execution [18]–[22], [29], [30]. In this section, we categorize previous studies into two distinct methods: *coarse-grained* mapping and *fine-grained* mapping.

1) Coarse-grained Mapping

A straightforward approach to mapping a query is to assign all operators to a single device.

Spark-Rapids [17], an open-source library supported by NVIDIA, enables Spark [5] to leverage GPUs for maximizing query throughput. In Spark-Rapids, the entire query is mapped to GPUs, and the selected device remains constant throughout the query's execution. GFlink [18] and G-Storm [19] extend Flink [7] and Storm [6], respectively, to utilize GPUs in query processing. These systems also map the entire query to GPUs, thus avoiding data transition overhead resulting from data movement between heterogeneous devices. SABER [20] provides a hybrid stream processing engine that executes SQL queries using all available CPU and GPU cores. To achieve this, SABER views the query as a series of data-parallel query tasks that can run on either CPUs or GPUs. It schedules the query while monitoring the throughput of each query task on a specific processor.

Overall, these aforementioned approaches overlook the device preferences of individual operators, which enable them to achieve superior efficiency on specific devices [28].

2) Fine-grained Mapping

Fine-grained mapping selectively assigns each operator to heterogeneous devices based on its device preference.

FineStream [21] initially considers the static device preference for each operator and performs operator-level mapping at runtime on integrated CPU-GPU architectures. This approach schedules each operator to heterogeneous devices by considering the input data size according to varying data ingestion rates. Since FineStream is designed for integrated CPU-GPU architectures, it ignores the data transition overhead between heterogeneous devices, which is impractical for discrete CPU-GPU architectures.

Crystal [22] proposed an efficient operator-level mapping for discrete CPU-GPU architectures, which utilizes the predefined device sensitivity of each operator. However, the query plan in this approach is predetermined during compilation stages and cannot effectively cope with dynamically changing conditions such as user requirements and traffic fluctuations.

Recently, DYNO [30] introduced a dynamic operator-level mapping algorithm that uses a tree-based machine learning approach to assign queries to devices. However, the requirement for an additional GPU to train workloads makes a direct comparison with dSTREAM challenging.

3) Limitations of Existing Studies

Previous studies have inherent limitations that hinder optimal performance.

One such limitation is the coarse-grained mapping, which overlooks individual device preferences per operator [17]–[20]. It is widely acknowledged that each operator in SQL queries exhibits distinct device preferences [21], [28]. Thus, exploiting only a single device during query processing fails to optimize the performance further.

While the fine-grained mapping considers the device preference of each operator, a specific study [21] focuses on assigning each operator to heterogeneous devices, assuming that transferring stream data from main memory to GPU memory carries no additional cost. In discrete CPU-GPU architectures, the state information generated by each operator's processing is transferred between heterogeneous devices. Given that the size of state information varies dynamically at runtime based on user needs and input data size, the overhead of data transition needs to be carefully considered.

Moreover, many studies employing static device preferences [17]–[19], [22] struggle to adapt to the changing device preferences of operators during runtime. They often establish a static table detailing the device preferences for each operator through profiling or rely on GPU-specific libraries that are pre-bound to GPUs. While these methods can enhance query performance for traffic patterns similar to those profiled, their capacity to guarantee such performance under diverse conditions is limited. Table 1 summarizes the differences between our study and existing studies.

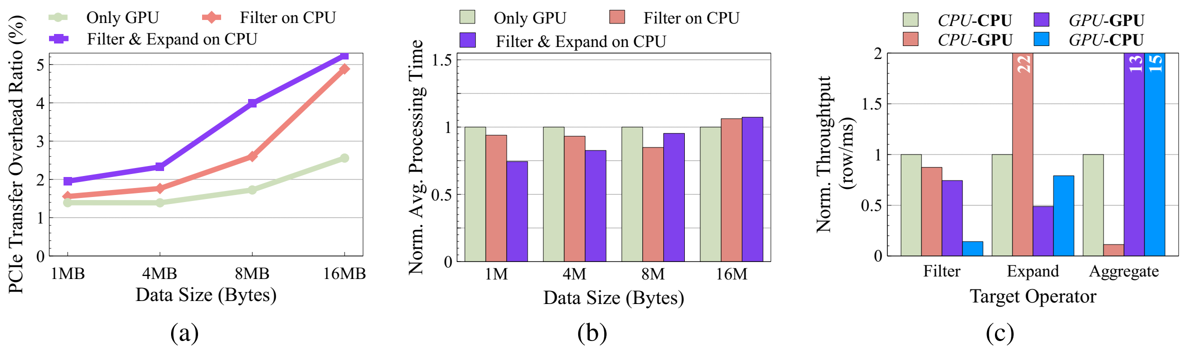


FIGURE 2. (a) PCIe transfer overhead ratio for different batch size and operator mapping case on discrete CPU-GPU architectures. (b) Normalized average query processing time for operator mapping scenarios across various data sizes. (c) Normalized throughput for each target operator by the location of preceding operators.

D. MOTIVATIONS

1) Low PCIe Overhead for Small Data Sizes

In discrete CPU-GPU architectures, the PCIe transfer overhead has been recognized as a performance bottleneck in query processing. Before GPU processing takes place, stream data needs to be transferred from host memory to GPU memory via the PCIe bus. Based on this, previous studies [17]–[20] allocated all operators to a single device to eliminate data movement. Nevertheless, our observation indicates that the overhead is insignificant for small data sizes.

Figure 2 (a) shows the proportion of time needed for data transfer via the PCIe bus to the total execution time, which is a leading factor of PCIe transfer overhead. We used a synthetic *select-filter-expand* query to investigate the overhead in query processing, with detailed experimental settings outlined in Section V-A. We further defined three operator mapping cases: (1) mapping all operators to the GPU (Only GPU), (2) mapping only the *filter* to the CPU (Filter on CPU), and (3) mapping the *filter* and *expand* to the CPU and the *select* to the GPU (Filter & Expand on CPU). In our experiment, we generated traffic with a constant data size per second to simulate a fixed-rate data inflow. To measure the PCIe transfer time during query processing, we utilized NVIDIA Nsight Systems [31], which profiles various GPU metrics.

As shown in Figure 2 (a), the PCIe transfer overhead ratio rises with larger data size. Interestingly, the PCIe overhead remains minimal (i.e., less than 2.5%) when the data size is smaller than 4 MB, regardless of the operator mapping cases. Even as the data size increases to 16 MB, the PCIe overhead is still low. This observation suggests that the PCIe overhead resulting from data movement does not significantly contribute to performance degradation. Thus, in discrete CPU-GPU architectures, implementing operator-level mapping may lead to enhanced query performance.

2) Impact of Data Size

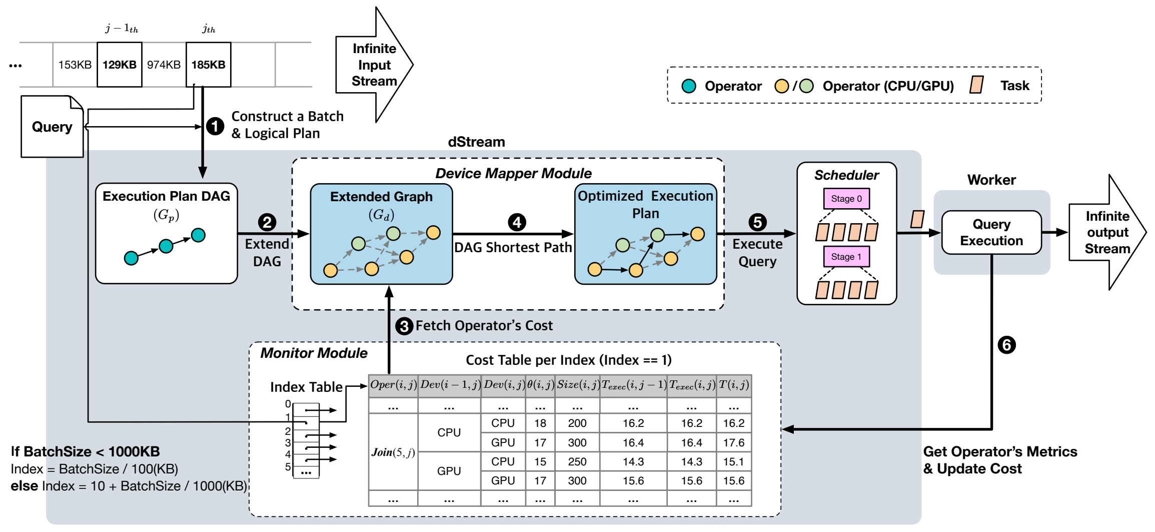
The data stream frequently experiences irregular and unpredictable fluctuations in input data size [32]. During promotional events or holiday sales, an e-commerce platform may encounter spikes in data volume, leading to highly volatile events and variations in data size. This indicates that each operator processes different amounts of data. For example, an operator that calculates averages can accelerate computation by distributing subsets of data across multiple GPU cores. However, if the operator is instead mapped to the CPU and the incoming data size is huge, utilizing the CPU may not be efficient. Maintaining a fixed device preference for each operator fails to guarantee optimal performance, especially as data size fluctuates during runtime. Thus, it becomes crucial to evaluate how dynamic data size influences the performance of operators mapped to heterogeneous devices.

To explore the relationship between data size and the performance of operators mapped to computing devices, we used the identical query as described in Section II-D1. Figure 2 (b) illustrates the changes in query processing time with respect to data size and operator mapping scenarios. The y-axis is the average processing time for the operator mapping scenario, which has been normalized against the Only GPU scenario. When the data size is small (i.e., less than 8 MB), Filter & Expand on CPU exhibits the shortest processing time. However, as the data size reaches 8 MB, Filter on CPU surpasses the other cases. This suggests that performance improves when both the CPU and GPU are utilized rather than relying on a single device. Furthermore, when the data size reaches 16 MB, performing all operators on the GPU demonstrates the best performance. These results indicate that the device preferences of each operator change dynamically based on varying data sizes.

3) Impact of Location of Consecutive Operators

FineStream [21] proposed an operator-level query mapping method for integrated CPU-GPU architectures. This study assumed that there is no data transition overhead between CPUs and GPUs. In contrast, in dedicated CPU-GPU architectures that communicate data over a PCIe bus, the data transition overhead cannot be ignored.

To facilitate the execution of operators on GPUs, parallel computing libraries such as CUDA [33] or OpenCL [34] should be utilized. These libraries provide a comprehensive set of built-in functions designed for GPU compatibility, enabling data to be transformed into formats optimized for GPU processing. However, when an operator is mapped to a device different from its preceding operator, synchronous data struc-



Cost Table per Index (Index == 1)

$Oper(i,j)$	$Dev(i-1,j)$	$Dev(i,j)$	$\theta(i,j)$	$Size(i,j)$	$T_{exec}(i,j-1)$	$T_{exec}(i,j)$	$T(i,j)$
...	...	...	...	...	...	...	...
Join$(5,j)$	CPU	CPU	18	200	16.2	16.2	16.2
		GPU	17	300	16.4	16.4	17.6
	GPU	CPU	15	250	14.3	14.3	15.1
		GPU	17	300	15.6	15.6	15.6
...	...	...	...	...	...	...	...

FIGURE 3. Overall architecture of dSTREAM.

ture conversion is required. This can disrupt seamless query execution and degrade overall performance. Essentially, the query performance can vary based on the devices to which the preceding and subsequent operators are mapped.

To confirm the correlation between operator placement and query performance, we conducted an experiment using a synthetic query composed of *filter*, *expand*, and *aggregate* operators. For the experiment, we also introduced two terms: target operator and preceding operator. Here, the target operator refers to an operator executed following its immediately preceding operators. Figure 2 (c) presents the throughput of each target operator (i.e., *filter*, *expand*, and *aggregate*) when both the target operator and other preceding operators are mapped to either CPUs or GPUs. In the figure, *CPU*-**GPU** indicates a scenario where the target operator is mapped to the **GPU** while the preceding operators are mapped to the *CPU*. Additionally, the throughput of each target operator is normalized against the throughput in the *CPU*-**CPU** case.

As shown in Figure 2(c), all operators have distinct performance patterns. The throughput in the *CPU*-**GPU** case with the *expand* operator exceeds that in the *CPU*-**CPU** case by about 22 times. This is because the *expand* operator duplicates rows and fills them with values without relying on the results of other operations. This dependency-free operation can be accelerated through the parallel execution capabilities of thousands of GPU cores.

On the other hand, the *aggregate* operator exhibits a different pattern. For instance, the *GPU*-**CPU** and *GPU*-**GPU** cases with the *aggregate* operator outperform other cases. This suggests that regardless of where the *aggregate* operator is allocated, the optimal choice is to map the preceding operators to the GPU. This finding highlights the sensitivity of device preference to the location of preceding operators and the target operator.

III. DESIGN OF dSTREAM

This section introduces dSTREAM, a dynamic operator-level query mapping approach designed for discrete CPU-GPU architectures. We outline the architecture of dSTREAM and describe two core modules in detail. The notations used to explain our mechanism are summarized in Table 2.

A. OVERALL ARCHITECTURE

TABLE 2. Notations used in dSTREAM.

Notation	Description
$G_p = (V_p, E_p)$	A graph G_p representing a physical plan consists of a set V_p of operators v and a set E_p of directed edges (u, v).
$G_d = (V_d, E_d)$	A graph G_d comprises a set V_d of operators v and a set E_d of directed edges (u, v).
i	Sequence of operator.
j	Sequence of batch in the index table according to batch size.
$v.distance$	The accumulated cost from the source operator to the operator v.
$v.\pi$	The predecessor of operator v maintains $v.distance$.
$v.edges$	Directed edges with operators after operator v.
$Oper(i,j)$	The i_{th} operator executed in j_{th} batch belonging to G_d.
$Dev(i,j)$	The device that $Oper(i,j)$ is performed on.
$T(i,j)$	The final cost of $Oper(i,j)$.
$T_{trans}(i,j)$	A estimated cost of data transition of $Oper(i,j)$.
$T_{exec}(i,j)$	The j_{th} predicted execution cost by estimating the cost of processing $Oper(i,j-1)$ and $Oper(i,j)$.
$\theta(i,j)$	The latest execution time of $Oper(i,j)$ in j_{th} batch.
T_{init}	Initialization costs associated with data transition.
$Size(i,j)$	The data size of $Oper(i,j)$.
BW	PCIe bandwidth between CPU and GPU.

Figure 3 depicts the detailed system architecture and operational flow of dSTREAM. dSTREAM enhances Spark Streaming with two key modules for operator-level query mapping: the *Monitor* module and the *Device Mapper* module, illustrated as two dotted rectangular boxes in the figure. The *Monitor* module collects performance metrics of each operator from the previous query executions and constructs the cost table by estimating the mapping cost of each operator. The *Device Mapper* module identifies an optimized execution plan with the lowest mapping cost among the potential mapping cases and triggers the scheduler to execute the query.

In dSTREAM, unbounded stream data is ingested in real-time and continuously accumulated into a buffer. When a query is submitted, dSTREAM constructs a batch unit of execution called a *micro-batch* from the buffer.

Once the micro-batch is constructed, dSTREAM analyzes the submitted query with user-defined operators and creates a logical plan. This logical plan is then converted into a physical plan, known as an execution plan DAG G_p ❶. Next, the *Device Mapper* module extends the execution plan DAG G_p to include GPU-based operators, thereby creating an extended graph G_d ❷. The module also updates the cost of each edge between operators in G_d by retrieving the operator's cost from the cost table managed by the *Monitor* module ❸. As a result, G_d represents all potential operator mapping scenarios, with each set of operators assigned to a specific device. Finally, the *Device Mapper* module employs a DAG shortest path algorithm to identify an optimized execution plan with the lowest cost among all operator mapping scenarios ❹. When the optimal execution plan is selected, dSTREAM delivers it to the scheduler for parallel execution ❺. The *Monitor* module collects metrics throughout the execution of a query. After the query finishes, it leverages these metrics to estimate the cost for each operator. These estimated costs are subsequently updated in the cost table ❻. The following subsection details the two primary tables, the index table and the cost table, managed in the *Monitor* module, and explains how these tables are used to calculate the total operator mapping cost.

B. MONITOR MODULE

1) Index Table Management

The *Monitor* module maintains an index table to access the cost table, with the index determined by the input batch size, as shown in Figure 3. Since all metric values maintained in the cost table depend on batch sizes, keeping information for each batch size leads to high memory consumption. Additionally, searching the cost table for each distinct batch size can increase the processing time. To address these issues, the *Monitor* module uses a separate cost table for each range of batch sizes, enabling the *Device Mapper* module to retrieve the operator's cost that previously achieved the best performance for similar batch sizes.

For example, if the batch size is smaller than a configurable threshold value, such as 1000 KB, the index is managed in units of 100 KB. For batch sizes exceeding this threshold, the index is managed in units of 1000 KB. Consequently, both the current batch size of 185 KB and the previous batch size of 129 KB in Figure 3 are handled under index 1 and utilize the same cost table. Since the method of dividing batch sizes can impact overall performance, we tuned the threshold value to achieve optimal results prior to experimentation.

2) Cost Table Management

The cost table in the *Monitor* module consists of eight entries to estimate the operator mapping cost. The first entry is the $Oper(i, j)$, which corresponds to the operator type in the execution plan DAG, where *i* denotes the execution order of each operator within the DAG. Since the DAG contains multiple operators, and may include multiple instances of the same operator type, it is crucial to collect metrics separately for each instance of the same operator, considering their unique execution orders during query processing. For instance, a query may include multiple instances of the *filter* operator to extract data based on various conditions such as $\leq$ or $\neq$. Identical instances of operators can exhibit different execution times and handle varying data sizes depending on their position within the DAG. The *j* indicates the execution order of batches within the same index value. If the current batch number within any index is j, the batch number of a similar size processed previously is $j - 1$. For example, in Figure 3, if the batch number for the size of 185 KB is j, the batch number for the size of 129 KB is $j - 1$.

The entries $Dev(i, j - 1)$ and $Dev(i, j)$ respectively indicate the devices on which the i_{th} operator was executed during the previous and current batch executions. If the i_{th} operator is executed on a GPU in the current batch and was executed on a CPU in the previous batch, then $Dev(i, j)$ is set to GPU and $Dev(i, j - 1)$ is set to CPU, updating only the relevant metrics associated with this configuration.

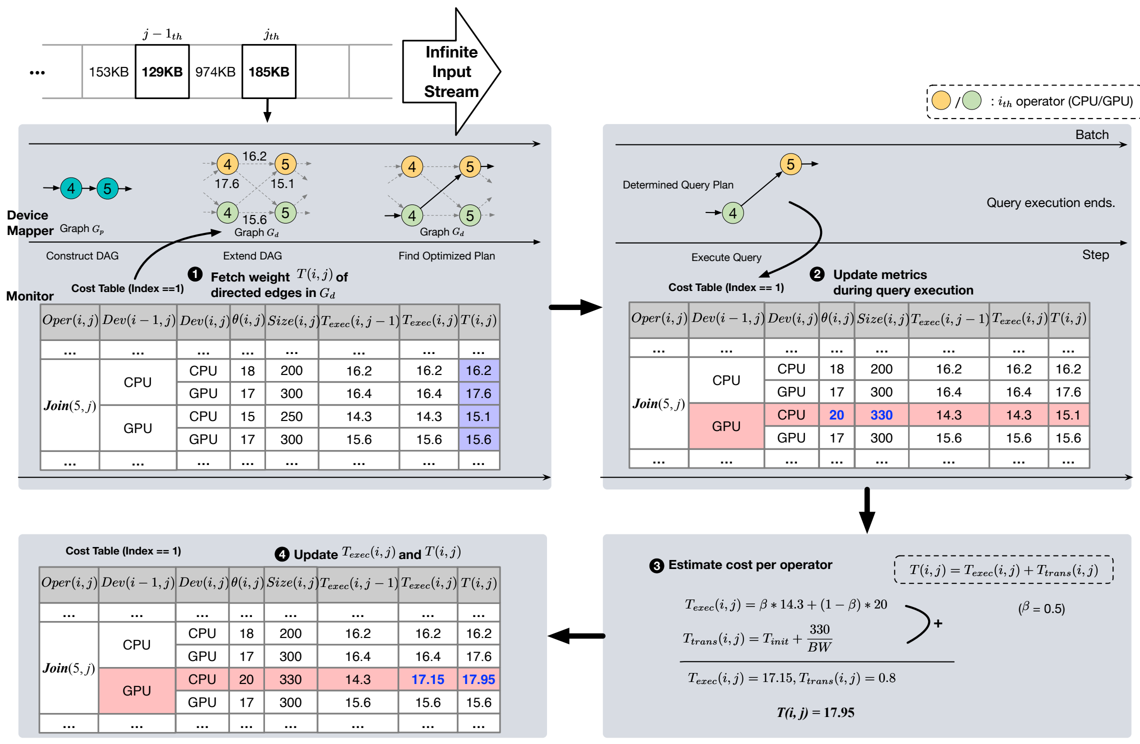
The entry $\theta(i, j)$ denotes the actual execution time of the i_{th} operator in the j_{th} batch. In DSPS, operators are generally processed using multiple partitions for map-reduce programming [35], which allows for parallel processing of each partition. As a result, the execution time for each partition is measured, and the average of these values is stored in the cost table.

The entry $Size(i, j)$ represents the amount of data processed by the i_{th} operator in the j_{th} batch. This entry is used to estimate the cost of transferring data between CPUs and GPUs via the PCIe bus. As operators process a batch, they generate different intermediate states, resulting in fluctuations in the amount of data handled by each operator. It is assumed that the dataset processed by the current operator is identical to that of the same operator in the previous execution, provided the batch sizes are similar.

Finally, the entries $T_{exec}(i, j - 1)$ and $T_{exec}(i, j)$ correspond to the estimated execution times of the i_{th} operator for the previous and current batch executions, respectively. After updating all metrics, the final estimated cost $T(i, j)$ is calculated using the formulas described in Section III-B3 and then updated in the cost table.

3) Cost Estimation

Based on the entries in the cost table, dSTREAM estimates the total operator mapping cost to determine the optimal placement for each operator. The cost of executing i_{th} operator $T(i, j)$ is the sum of the estimated execution time $T_{exec}(i, j)$ and data transition time $T_{trans}(i, j)$ as shown in Equation 1.



$j-1_{th}$
j_{th}
... 153KB 129KB 974KB 185KB
Infinite Input Stream
: i_{th} operator (CPU/GPU)
Device Mapper
Graph G_p
Graph G_d
Construct DAG
Extend DAG
Find Optimized Plan
Monitor
Cost Table (Index ==1)
❶ Fetch weight $T(i,j)$ of directed edges in G_d
Batch
Determined Query Plan
Query execution ends.
Execute Query
Step
❷ Update metrics during query execution
❹ Update $T_{exec}(i,j)$ and $T(i,j)$
❸ Estimate cost per operator
$T(i,j) = T_{exec}(i,j) + T_{trans}(i,j)$
$(\beta = 0.5)$
$T_{exec}(i,j) = \beta * 14.3 + (1-\beta) * 20$
$T_{trans}(i,j) = T_{init} + \frac{330}{BW}$
$T_{exec}(i,j) = 17.15, T_{trans}(i,j) = 0.8$
$T(i,j) = 17.95$

Cost Table (Index ==1)

$Oper(i,j)$	$Dev(i-1,j)$	$Dev(i,j)$	$\theta(i,j)$	$Size(i,j)$	$T_{exec}(i,j-1)$	$T_{exec}(i,j)$	$T(i,j)$
...	...	...	...	...	...	...	...
Join$(5,j)$	CPU	CPU	18	200	16.2	16.2	16.2
		GPU	17	300	16.4	16.4	17.6
	GPU	CPU	15	250	14.3	14.3	15.1
		GPU	17	300	15.6	15.6	15.6
...	...	...	...	...	...	...	...

Cost Table (Index == 1)

$Oper(i,j)$	$Dev(i-1,j)$	$Dev(i,j)$	$\theta(i,j)$	$Size(i,j)$	$T_{exec}(i,j-1)$	$T_{exec}(i,j)$	$T(i,j)$
...	...	...	...	...	...	...	...
Join$(5,j)$	CPU	CPU	18	200	16.2	16.2	16.2
		GPU	17	300	16.4	16.4	17.6
	GPU	CPU	20	330	14.3	14.3	15.1
		GPU	17	300	15.6	15.6	15.6
...	...	...	...	...	...	...	...

Cost Table (Index == 1)

$Oper(i,j)$	$Dev(i-1,j)$	$Dev(i,j)$	$\theta(i,j)$	$Size(i,j)$	$T_{exec}(i,j-1)$	$T_{exec}(i,j)$	$T(i,j)$
...	...	...	...	...	...	...	...
Join$(5,j)$	CPU	CPU	18	200	16.2	16.2	16.2
		GPU	17	300	16.4	16.4	17.6
	GPU	CPU	20	330	14.3	17.15	17.95
		GPU	17	300	15.6	15.6	15.6
...	...	...	...	...	...	...	...

FIGURE 4. Example of cost table management and cost estimation.

$$T(i,j) = T_{exec}(i,j) + T_{trans}(i,j) \tag{1}$$

To calculate the estimated execution time $T_{exec}(i,j)$, we used the exponential moving average (EMA) [36], which computes a weighted average of historical trend values and recent observations over time. In contrast to a simple moving average (SMA), which assigns equal weight to all data within the window, EMA uses an exponentially decreasing weight for older values. By weighting recent data more heavily, EMA provides a responsive measure of trends, making it ideal for applications where rapid response to recent changes is essential. That is, $T_{exec}(i,j)$ is calculated as the sum of $T_{exec}(i,j-1)$ and $\theta(i,j)$ as shown in Equation 2.

$$T_{exec}(i,j) = \beta * T_{exec}(i,j-1) + (1-\beta) * \theta(i,j) \tag{2}$$

The estimated execution time $T_{exec}(i,j)$ depends on whether it focuses on recent or historical trends. The hyperparameter β represents the weight assigned to current observations relative to historical trends. We set β to 0.5, as this value demonstrated the most stable performance across multiple experiments. A detailed explanation for choosing a β of 0.5 is provided in Section V-E.

Data transition occurs when the previous operator and the current operator run on different devices. Therefore, $T_{trans}(i,j)$ represents the time required to transfer data over the PCIe bus between these heterogeneous devices. We calculated the data transition time $T_{trans}(i,j)$ as the sum of an initialization time T_{init} and the data transfer time as shown in Equation 3, where BW is the PCIe bandwidth between the CPU and the GPU.

$$T_{\text{trans}}(i,j) = \begin{cases} T_{\text{init}} + \dfrac{Size(i,j)}{BW}, & \text{if Dev}(i-1,j) \neq \text{Dev}(i,j), \\ 0, & \text{if Dev}(i-1,j) = \text{Dev}(i,j), \end{cases} \tag{3}$$

Figure 4 illustrates how the *Monitor* module updates its cost table when the current operator *Join* is the 5_{th} operator in the execution plan DAG. Assuming the size of the j_{th} batch is 185 KB and the size of the $j-1_{th}$ batch is 129 KB, both batches use the same index (i.e., index 1) and share the same cost table. Firstly, the *Device Mapper* module retrieves all metrics from the cost table that were updated during the processing of the $j-1_{th}$ batch and updates the cost of directed edges in the extended graph G_d ❶. Once all costs have been updated, the *Device Mapper* module determines an optimized execution plan by computing the shortest path in the DAG. In this scenario, the mapping case where the 4_{th} operator is mapped to the GPU and the 5_{th} operator is mapped to the CPU represents the execution plan with the minimum cost (i.e., 15.1). Secondly, the *Device Mapper* module executes the query based on the optimized execution plan and updates the corresponding entries ❷. For example, $\theta(5,j)$ is updated from 15 to 20, and $Size(5,j)$ is also updated from 250 to 330. Next, the cost of executing the 5_{th} operator $T(5,j)$ is calculated using Equation 1 ❸. In this case, the estimated execution time $T_{exec}(5,j)$ becomes 17.15 because the previously estimated execution time $T_{exec}(5,j-1)$ was 14.3 in the cost table, and the actual execution time $\theta(5,j)$ is 20 (assuming β is 0.5). The data

Algorithm 1: DAG shortest path algorithm

```
Input: $sortedList$ - Linked list preserving
       topological order of graph $G_d$
       $fetchCost(u, v)$ - A function that fetches the
       cost of $v$'s edges from the cost table
Output: $Q$ - List for optimized plan
1  Create $startVertex$ where
     $startVertex.distance = 0$ and
     $startVertex.\pi = NIL$
2  // Initialize all vertices in the $sortedList$
3  foreach $vertex\ v \in sortedList$ do
4      $v.distance \leftarrow \infty$
5      $v.\pi \leftarrow NIL$
6      if $v.edges$ is empty then
7          add a directed edge $(startVertex, v)$ to
             $startVertex.edges$
8  Insert $startVertex$ onto the front of the $sortedList$
9  // Shortest path estimation
10 foreach $vertex\ u \in sortedList$ do
11     foreach $vertex\ v \in u.edges$ do
12         $cost \leftarrow fetchCost(u, v)$
13         if $v.distance > u.distance + cost$ then
14             $v.distance = u.distance + cost$
15             $v.\pi \leftarrow u$
16 // Find a shortest path
17 $currentVertex \leftarrow$ last elements of the $sortedList$
18 while $currentVertex$ is not null do
19     insert $currentVertex.\pi$ to $Q$
20     $currentVertex \leftarrow currentVertex.\pi$
```

transition time $T_{trans}(5, j)$ is calculated as 0.8 using the amount of data processed by the operator $Size(5, j)$, which is 330. Therefore, $T(5, j)$ becomes 17.95 by adding 17.15 and 0.8. Once all calculations are completed, $T_{exec}(5, j)$ and $T(5, j)$ are updated in the cost table ❹. The final cost, $T(5, j)$, is used by the *Device Mapper* module to calculate the optimized execution plan for the upcoming batch $j + 1$. This process enables the *Monitor* module to estimate the cost of each operator with minimal overhead, even without prior knowledge. The additional overhead of our modules are discussed in Section V-F.

C. DEVICE MAPPER MODULE

1) Extension of Execution Plan DAG

A primary function of the *Device Mapper* module is to extend an execution plan DAG G_p into an extended graph G_d, incorporating both CPU and GPU-based operators.

Assume the execution plan DAG $G_p{=}(V_p, E_p)$ consists of a set of vertices V_p and a set of directed edges E_p, where each vertex in V_p denotes an operator and each edge in E_p indicates the antecedent relationship between any operators u and v in V_p. Using this graph, the *Device Mapper* module constructs a new graph $G_d{=}(V_d, E_d)$ by extending V_p to V_d and assigning a corresponding cost to each extended edge in E_d. For example, let S denote the subset of V_p that represents operators capable of running only on CPUs. Each vertex $v \in S$ is extended to $v_{cpu} \in V_d$, while each vertex $v \in V_p - S$ is extended as either v_{cpu} or $v_{gpu} \in V_d$. Here, v_{cpu} represents a CPU-based operator, and v_{gpu} represents a GPU-based operator. The set of edges E_p is also extended to E_d based on the outcomes derived from the updated vertex set V_d.

Note that the new graph G_d includes a single source vertex (a vertex with zero in-degree) and a single sink vertex (a vertex with zero out-degree). The source vertex is intentionally added as a dummy operator to optimize the search operation, while the sink vertex represents a materialize operator that exclusively runs on the CPU. As a result, the maximum number of vertices and edges in G_d is $2 \times |V_p|$ and $4 \times (|V_p|{-}1)$, respectively. It is also worth noting that G_d maintains a DAG structure, consistent with the original graph G_p being structured as a DAG.

2) Finding an Optimized Plan

In the *Device Mapper* module, quickly finding the best execution plan among several candidates is crucial. This is because the process of creating the best execution plan occurs synchronously within the critical path. As the time to find the optimal execution plan increases, processing latency also rises. Therefore, it is imperative to calculate the cost of the execution plan as early as possible. Additionally, depending on the query type, operator mapping can be complex. This increases time complexity and ultimately prolongs the search for the best execution plan.

The graph G_d can have up to four edges per operator, resulting in $4^{|V_p|}$ possible combinations. For example, in Figure 4, the 4_{th} operator of G_p is extended to both CPU and GPU-based operators in G_d, each connected to two 5_{th} operators via gray dotted lines. Consequently, the two 4_{th} operators of G_d have four edges in total.

To optimize the search for the optimal path, the *Device Mapper* module employs a DAG shortest path algorithm with topological sorting [37]. This approach is chosen because the time complexity of the shortest path algorithm for a DAG $G{=}(V, E)$ with a single source is $\Theta(V{+}E)$. As part of this strategy, a dummy vertex, referred to as $startVertex$, is inserted before the source vertices. For example, in scenarios involving the join operator, which takes two datasets as input and may have multiple source vertices, the *Device Mapper* module includes a dummy vertex labeled $startVertex$. This dummy vertex is connected to the source vertices to facilitate efficient path-finding. As a result, each source vertex uniformly recognizes $startVertex$ as its only antecedent vertex. This enables the *Device Mapper* module to leverage the single-source DAG shortest path algorithm.

The *Device Mapper* module also utilizes the relaxation technique [37], which is employed for predicting the shortest path in a DAG. This technique involves updating the short-

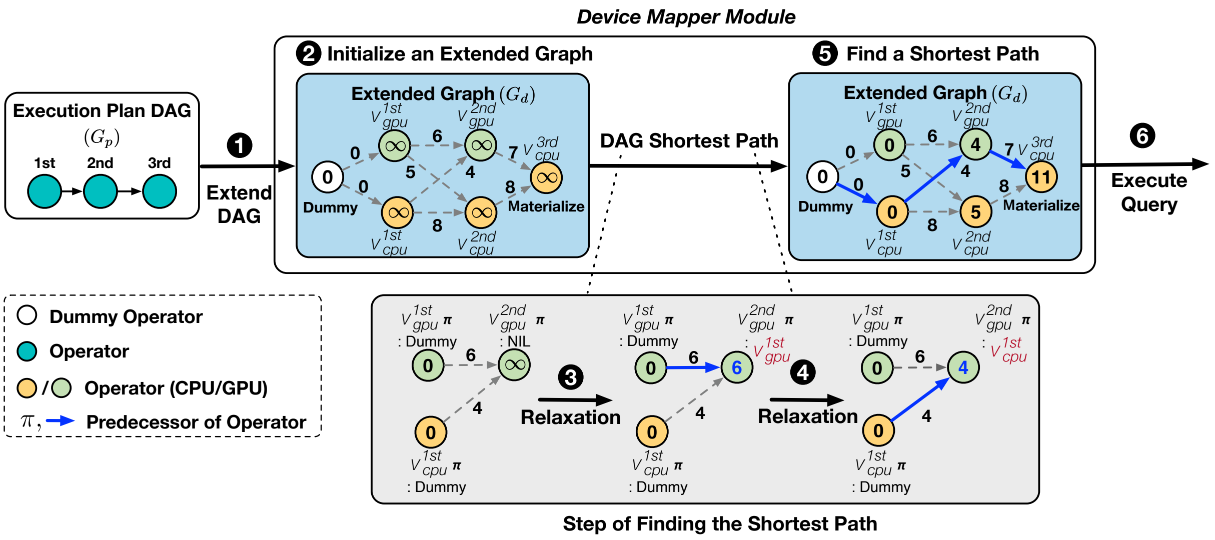


FIGURE 5. The process of *Device Mapper* module.

est distance to a vertex when a shorter path is discovered through the edge currently under consideration. To implement this technique, each vertex $v \in V_d$ maintains attributes $v.distance$, $v.\pi$, and $v.edges$. Here, $v.distance$ represents an upper bound on the shortest path from the source s to v, while $v.\pi$ denotes the predecessor that maintains the shortest path to v. The set of all possible edges from vertex v to its direct descendants w_1, w_2, ... , w_n is denoted as $v.edges$. Initially, the cost of the operators is set to zero to establish metrics for different paths. Details of the algorithm can be found in Algorithm 1.

Figure 5 illustrates how *Device Mapper* module determines an optimized execution plan. When the micro-batch is constructed, dSTREAM generates an execution plan DAG denoted as G_p and extends it to G_d as detailed in Section III-C1 ❶. In G_d, each operator, except for the materialize operator, can be assigned to either a CPU or GPU device. For instance, the first operator in G_p is assigned to v^{1st}_{cpu} or v^{1st}_{gpu} in G_d. Additionally, a dummy vertex $startVertex$ is added as a predecessor to both v^{1st}_{cpu} and v^{1st}_{gpu}. The $distance$ for all operators, except for $startVertex$, is initialized to ∞ and their predecessor (π) is set to NIL[1] ❷.

After the extended graph is created, the *Device Mapper* module employs the DAG relaxation technique to identify the minimum cost path. When v^{1st}_{gpu} connects to v^{2nd}_{gpu}, the relaxation is performed ❸. Since the distance from v^{1st}_{gpu} to v^{2nd}_{gpu} is 6, which is less than the initial distance with ∞, the distance of v^{2nd}_{gpu} is updated to 6, and its predecessor π is set to v^{1st}_{gpu}. In the subsequent step, the *Device Mapper* module applies relaxation from v^{1st}_{cpu} to v^{2nd}_{gpu} ❹. Here, the sum of the distance from v^{1st}_{cpu} and the cost to reach v^{2nd}_{gpu}, which is 4, is less than 6. Consequently, both the distance to v^{2nd}_{gpu} and its predecessor π are updated to 4 and v^{1st}_{cpu}, respectively. This process iterates through all vertices, continually

[1]NIL represents a null or empty value.

updating distances and predecessors to determine the shortest path. Upon finishing the DAG shortest path algorithm, the optimized execution plan is established by tracing back from the sink vertex's π ❺. Finally, dSTREAM executes the query based on the optimized plan ❻.

With these features, the *Device Mapper* module finds the optimized execution plan through the DAG shortest path algorithm within a reasonable time by leveraging the fact that the extended graph G_d is a DAG.

IV. IMPLEMENTATION

We implemented dSTREAM on Spark Streaming [5] v3.2.3, which supports SQL-based optimized query processing. To enable the dynamic operator-level query mapping on discrete CPU-GPU architectures, we modified Spark SQL [27]. This module is responsible for establishing the query execution plan and mapping each operator to the appropriate computing device.

Additionally, we integrated the Spark-Rapids library [17] to implement the *Device Mapper* module. This library provides GPU-compatible functions for each operator. To collect performance metrics from each operator at runtime, we utilized Spark accumulator. The accumulator is a global mutable variable that can be used to aggregate data across multiple tasks in parallel. This shared variable supports union and commutative functions, ensuring accurate metric collection, even when operations are partitioned in parallel [38].

We implemented the overall system components of dSTREAM using the Scala language. The source code is available at *https://github.com/siblue202/dStream*.

dSTREAM performs operator-level query mapping by leveraging the fact that all query operators form a DAG. Although it was implemented in Spark Streaming, we anticipate that the main concepts proposed by dSTREAM can be easily applied to various DSPS that support SQL and operator-based queries.

V. EVALUATION

A. EXPERIMENTAL METHODOLOGY AND SETUP

This section compares the overall performance of dSTREAM with existing coarse-grained and fine-grained mapping techniques on discrete CPU-GPU architectures.

1) Experimental Configurations

We conducted a series of experiments using a Spark cluster consisting of one master node and two worker nodes. This configuration was chosen as it is commonly used in existing studies and sufficient to demonstrate the feasibility and effectiveness of our core design. Each worker node ran a Spark executor with 8 CPU cores and 48GB of memory, using 8 data partitions to enable parallel processing. Both dSTREAM and the comparison targets were executed on the Oracle JVM 8 with the G1 garbage collector. Detailed hardware configurations can be found in Table 3.

TABLE 3. Hardware configurations.

CPU	AMD Ryzen 9 3900X 12-core 3.80 GHz
GPU	NVIDIA GeForce RTX 3070
Memory	DDR4, 64 GB
Storage	Samsung SSD 970 EVO NVMe SSD
Ethernet	1 Gbps

2) Workloads and Stream Traffic Types

The workloads and query details used for our experiments are summarized in Table 4. We utilized two real-world streaming benchmarks: the Linear Road Benchmark (LRB) [39] and the Cluster Monitoring Benchmark (CMB) [40].

For example, LR1 query includes a join operator, which is compute-intensive. To execute a join, the streaming engine must locate and co-locate matching keys from different datasets, leading to substantial data movement across worker nodes. In contrast, CM queries involve aggregate operators, such as counting rows and computing averages. These operators typically work on a single dataset, allowing them to perform calculations locally on each partition and then combine the results with minimal data movement.

We set watermark [41] on all queries to prevent the state generated during stream processing from growing indefinitely. This means that the streaming engine waits for up to the watermark duration for late data to arrive for a given window before finalizing the results. In our experiments, the watermark duration was set to 60 seconds. Additionally, we generated both random and constant traffic to observe performance patterns under different traffic scenarios. The detailed description of the traffic scenarios is as follows.

- **Random traffic** describes a scenario where datasets of varying sizes arrive at regular intervals every second. The sizes follow a normal distribution, so the average size converges to the specified traffic rate. We used a normalized random traffic rate of 256 KB for the LR1 and CM1 workloads and 4 MB for the LR2 and CM2 workloads.
- **Constant traffic** describes a scenario where a fixed dataset size arrives at regular intervals every second. For the LR1 and CM1 workloads, the dataset size generated every second ranges from approximately 25 KB to 1024 KB. For the LR2 and CM2 workloads, the dataset size ranges from approximately 1 MB to 16 MB.

In both traffic scenarios, all queries receive varying amounts of data. To simulate a practical streaming system, we utilized Kafka [42] as the message broker. The Kafka producer sends chunks of the dataset to the message broker, while the Kafka consumer stores the dataset in the local file system of the master node. The streaming engine on the worker node then fetches the datasets from the message broker and begins processing the queries. Each experiment was repeated ten times, with each run lasting thirty minutes. The final results were determined by averaging the outcomes of these ten repetitions.

3) Comparison Targets

To evaluate the effectiveness of dSTREAM's design approach, we selected two coarse-grained mapping approaches that map an entire query to either the CPU or the GPU, along with one fine-grained mapping approach that assigns each operator to a suitable heterogeneous device based on static device preferences.

We determined the static device preference for each operator by profiling each query based on traffic rate and collecting the device preferences that exhibited optimal performance for each operator. Leveraging this prior knowledge of static device preferences per operator, the fine-grained mapping approach performs the operator-level mapping for comparison. The three approaches used for the comparison are outlined below.

- **Only CPU** represents a coarse-grained mapping approach where entire queries are assigned to the CPU.
- **Only GPU** denotes a coarse-grained mapping approach where entire queries are assigned to the GPU.
- **Static Preference** represents a fine-grained mapping approach that assigns each operator to an appropriate device based on static device preference. This approach is used in FineStream [21] and Crystal [22].

B. AVERAGE LATENCY AND THROUGHPUT

In this section, we analyze the average latency and throughput of dSTREAM compared to other approaches under different traffic scenarios.

1) Average Latency Under a Random Traffic Scenario

Figure 6 shows the average latency according to traffic rate under a random traffic scenario. The x-axis represents the traffic rate in bytes per second, and the y-axis represents the normalized average latency based on dSTREAM. Latency is defined as the total duration from the creation of a batch to the

TABLE 4. Query details of real-world streaming workloads used in our experiments.

Benchmark	Notation	Query Detail
Linear Road [39]	LR1	SELECT L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment FROM SegSpeedStr [range 30 (slide 5)] as A, SegSpeedStr as L WHERE (A.vehicle == L.vehicle)
	LR2	SELECT timestamp, highway, direction, segment, COUNT(vehicle) as num Vehicle FROM SegSpeedStr [range 30 slide 15] GROUPBY (highway, direction, segment)
Cluster Monitoring [40]	CM1	SELECT timestamp, category, SUM(cpu) as totalCpu FROM TaskEvents [range 30 (slide 30)] GROUPBY category ORDERBY SUM(cpu)
	CM2	SELECT jobId, AVG(cpu) as avgCpu FROM TaskEvents [range 30 slide 5] WHERE (eventType == 1) GROUPBY jobId

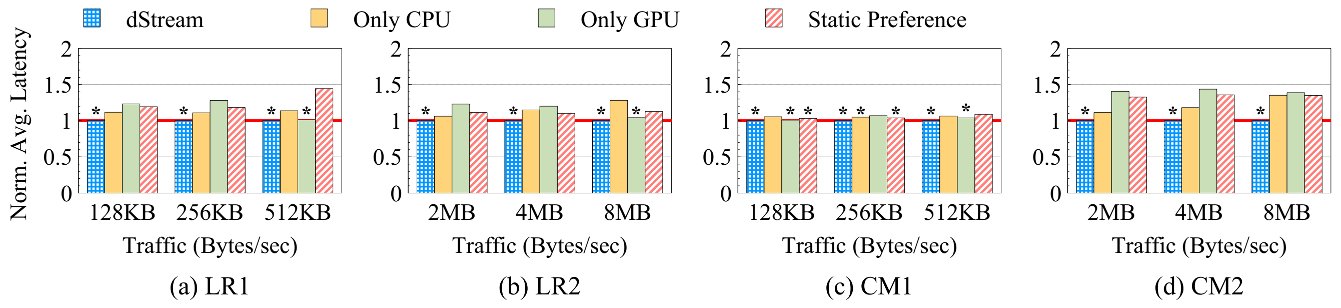


FIGURE 6. Normalized average latency of four queries under a random traffic scenario. A random record is generated every second so that its average converges to each traffic rate. The results are normalized to the average latency of dSTREAM. The targets with the lowest latency are marked with an asterisk (*). Targets with latency within 5% of the best-performing target are also marked with an asterisk.

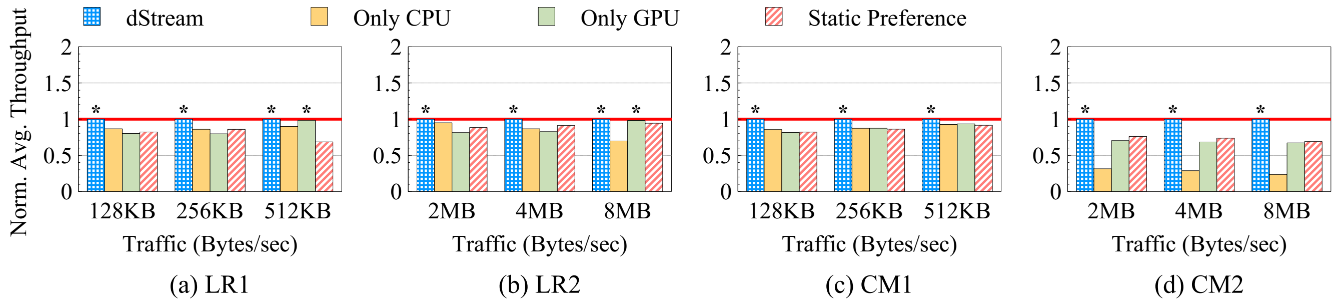


FIGURE 7. Normalized average throughput of four queries under a random traffic scenario. A random record is generated every second so that its average converges to each traffic rate. The results are normalized to the average throughput of dSTREAM. The targets with the highest throughput are marked with an asterisk (*). Targets with latency within 5% of the best-performing target are also marked with an asterisk.

completion of the query processing. Lower values indicate better performance.

Overall, dSTREAM outperforms the comparison targets regardless of the traffic rate and query type. In Figure 6 (a), dSTREAM shows a decrease in average latency of approximately 13.8%, 18.7%, and 16% compared to Only CPU, Only GPU, and Static Preference, respectively, when the traffic rate is 128 KB. As the traffic increases to 512 KB, dSTREAM achieves a significant reduction in average latency, with a decrease of up to 30.6%. Despite this, the average latency of Only GPU is not significantly different from that of dSTREAM. This is because the LR1 query includes a join operator, which is computationally intensive and time-consuming. Only GPU speeds up the computation of the join operator by distributing large amounts of data across many GPU cores. Similarly, dSTREAM efficiently maps the join operator to GPUs as much as possible through its *Device Mapper* module.

In contrast, dSTREAM shows comparable performance with other comparison targets across all traffic rates, as shown in Figure 6 (c). Note that the CM1 query uses the *sort* operator to organize the results of aggregated data in sorted order and stores them as an intermediate state, which consumes a significant portion of the total duration. The streaming engine accumulates the state at runtime until the watermark duration expires. As the state size increases, so does the time it takes to manage the state. The sort operator in Spark operates internally as the Timsort algorithm [43], which uses insertion sort involving frequent, small-scale swaps that result in scattered memory access patterns. Due to this feature, Spark typically forces the sort operator to run only on CPUs. This indicates that the performance gains from dynamic operator-level query mapping are diluted by the necessity of running the sort operator exclusively on CPUs.

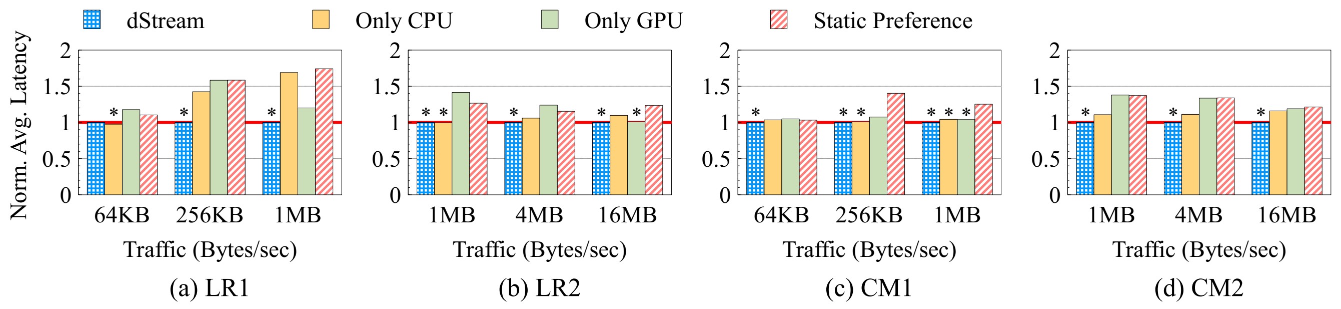


FIGURE 8. Normalized average latency of four queries under a constant traffic scenario. The results are normalized to the average latency of dSTREAM. The targets with the lowest latency are marked with an asterisk (*). Targets with latency within 5% of the best-performing target are also marked with an asterisk.

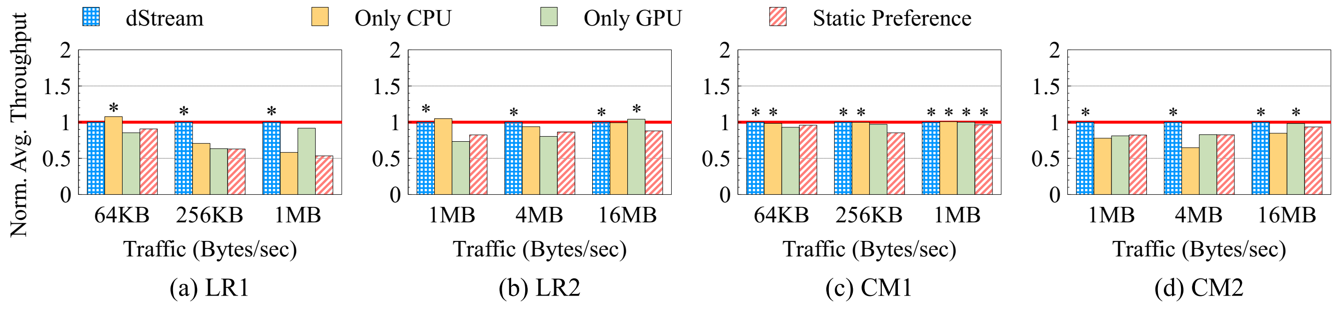


FIGURE 9. Normalized average throughput of four queries under a constant traffic scenario. The results are normalized to the average throughput of dSTREAM. The targets with the highest throughput are marked with an asterisk (*). Targets with throughput within 5% of the best-performing target are also marked with an asterisk.

2) Average Throughput Under a Random Traffic Scenario

Figure 7 shows the average throughput across different traffic rates in a random traffic scenario. Once again, the results indicate that dSTREAM achieves higher throughput compared to the other comparison targets. In Figure 7 (a), dSTREAM increases the throughput by up to about 18% (128 KB), 25% (256 KB), and 45% (512 KB) compared to Only CPU, Only GPU, and Static Preference, respectively.

Interestingly, dSTREAM exhibits different performance characteristics based on the query type. While Figure 7 (c) shows that dSTREAM's throughput improvements are not significantly superior to the comparison targets, Figure 7 (d) highlights a notable increase of up to 70% compared to Only CPU. These differences stem largely from the inherent characteristics of the queries. For instance, the CM1 query includes a *sum* operator that aggregates values by key, updating a single variable with each new value insertion. Because the sum operator is not computationally intensive, the benefits of dynamic operator-level mapping are not evident, regardless of the traffic rate. In contrast, the CM2 query involves an *average* operator, which performs more complex computations. This operator initially stores values for each key in an array list, computes the total sum, and then calculates the average. As traffic rates fluctuate, causing a significant increase in the number of values stored in the array list, the computational workload grows substantially. Consequently, dSTREAM significantly enhances performance by efficiently mapping compute-intensive operators to the most suitable device, thereby accelerating their execution.

3) Average Latency Under a Constant Traffic Scenario

To evaluate dSTREAM's effectiveness under various traffic scenarios, we conducted experiments using the same workloads under a constant traffic scenario.

Remarkably, dSTREAM consistently demonstrates low latency across a wide range of traffic scenarios for all queries. However, it does not consistently achieve the lowest latency in all cases. In Figure 8 (a), Only CPU shows the lowest latency when the traffic rate is 64 KB. This suggests that for small data sizes, it is efficient to run operators solely on the CPU, avoiding the overhead of data transfer associated with offloading operators to the GPU. However, as the traffic rate increases, latency sharply rises. This is because coarse-grained mapping assigns the operator to a specific device without considering the varying sizes of incoming data.

In contrast, dSTREAM achieves latency reductions of approximately 36.7% and 42.5% for data sizes of 256 KB and 1 MB, respectively. Our *Device Mapper* module consults the cost table to determine the cost associated with each operator that has previously achieved optimal performance for similar batch sizes. This enables the establishment of an optimal query execution plan at runtime, effectively reducing latency even amid traffic fluctuations.

In addition to variations in data size, the placement of consecutive operators within the device also plays a crucial role in performance optimization. In Figure 8 (c), dSTREAM and the two coarse-grained mapping approaches exhibit similar latency at all traffic rates, while Static Preference shows higher latency at 256 KB and 1 MB. As noted earlier, the CM1 query is not compute-intensive. For queries with low computational demands like CM1, assigning the entire query

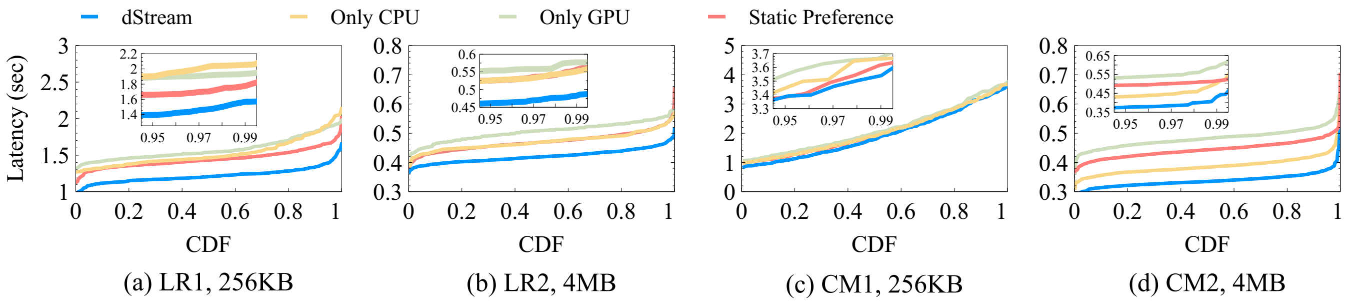


FIGURE 10. The cumulative distribution function (CDF) of tail latency per query under a random traffic scenario. An inset figure provides a detailed view of the 95_{th} to 99_{th} percentile latency.

exclusively to a single device can enhance performance. Although dSTREAM employs operator-level query mapping, it effectively considers data transition overhead and strategically devises a query execution plan to minimize these potential issues. By dynamically adapting to fluctuations in data size and transition overhead, dSTREAM consistently delivers robust performance across diverse scenarios.

4) Average Throughput Under a Constant Traffic Scenario

As shown in Figure 9 (a)-(d), dSTREAM also demonstrates superior throughput compared to the comparison targets. For example, in Figure 9 (a), dSTREAM achieved up a 38% increase in throughput compared to Only CPU, Only GPU, and Static Preference at 256 KB. While some comparison targets occasionally exhibit slightly higher throughput, the maximum difference is 7% or less, observed particularly with Only CPU (LR1 workload at 64 KB). Since the incoming data size remains constant rather than random, other methods also handle the workload reliably. Nonetheless, the marginal difference emphasizes that dSTREAM consistently delivers robust performance.

5) Summary

dSTREAM consistently outperforms the comparison targets in terms of average latency and throughput across various traffic rates and query types in both traffic scenarios. This superior performance is attributed to the effectiveness of our design approach. Unlike other comparison targets, dSTREAM employs dynamic operator-level query mapping, which dynamically assigns each operator to the proper device based on runtime conditions. Furthermore, dSTREAM optimizes device assignments by considering varying input data sizes and the placement of consecutive operators. This capability allows dSTREAM to excel in enhancing performance, especially for scenarios with fluctuating traffic conditions and compute-intensive query types.

C. TAIL LATENCY

In latency-sensitive applications, prolonged response times can significantly deteriorate user experiences [44]. Therefore, minimizing tail latency is a critical objective in DSPS. DSPS typically retrieve data from sources and process it. When bursts of traffic occur due to fluctuations, query processing times increase. Meanwhile, data continues to accumulate in the data source until the previous query processing completes. This accumulation affects subsequent query processing times, resulting in high tail latency [45].

1) Tail Latency Under a Random Traffic Scenario

Figure 10 illustrates a cumulative distribution function (CDF) of the tail latency per query under a random traffic scenario. Typically, the 95_{th} and 99_{th} percentiles are used to measure tail latency, representing the longer response times experienced by a small fraction of requests [46], [47]. To enhance clarity, we include a nested figure that highlights the tail latency from the 95_{th} to the 99_{th} percentiles.

As shown in Figure 10, dSTREAM reduced the $95_{th}/99_{th}$ percentile latency for most queries compared to the other targets. The LR1 query, in particular, showed the most remarkable improvement. In Figure 10 (a), the 95_{th} percentile tail latency of dSTREAM is reduced by 36% and 28% compared to Only CPU and Only GPU, respectively. Additionally, dSTREAM decreased tail latency by 27% compared to Static Preference. The comparison targets, which fail to account for dynamic data sizes at runtime, are unable to effectively mitigate the surge in tail latency due to traffic fluctuations.

In contrast, for the CM1 query, dSTREAM shows no significant difference in the $95_{th}/99_{th}$ percentile latency compared to the other targets. This is because the time-consuming *sort* operator is executed on the CPU, limiting the effectiveness of dynamic operator-level query mapping in reducing tail latency.

2) Summary

In addition to improving average latency and throughput, dSTREAM effectively reduces tail latency compared to the other targets. dSTREAM maintains a cost table that estimates the cost of mapping operators to either CPUs or GPUs at runtime. Using this cost table, dSTREAM determines the query execution plan with the least cost path among all operators, thereby minimizing tail latency as much as possible.

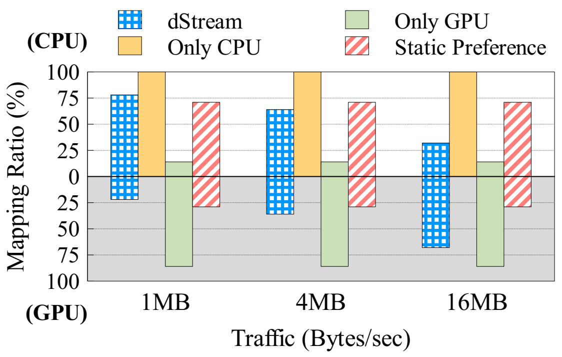


FIGURE 11. Mapping ratio between CPU and GPU for the LR2 query under a constant traffic scenario. The y-axis represents the mapping ratio, which indicates the proportion of operators in the query assigned to each device.

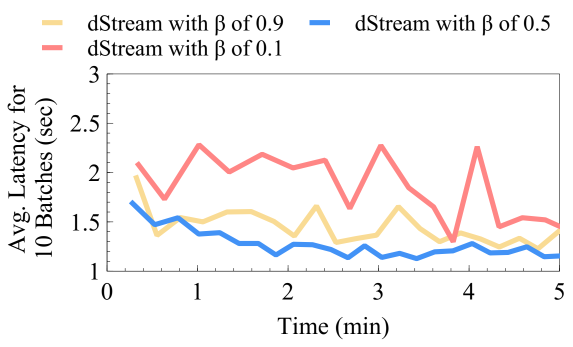


FIGURE 12. Average latency of the LR2 query over time in a random traffic scenario. The time represents the completion point of the last bath in a series of 10 consecutive batches.

D. DYNAMICS OF OPERATOR MAPPING

Given that query performance significantly depends on the optimal mapping of each operator to a specific device, it is crucial to analyze how effectively operators are assigned to the most appropriate devices at runtime. This section evaluates the effectiveness of dynamic operator-level query mapping proposed in dSTREAM. We used the LR2 query and measured the operator-device mapping ratio of all methods according to traffic rate under a constant traffic scenario.

1) Operator-Device Mapping Ratio

Figure 11 illustrates the operator-device mapping ratio based on the traffic rate. The x-axis represents the traffic rate in bytes per second, while the y-axis shows the mapping ratio between the heterogeneous devices. A mapping ratio closer to 100% for both computing devices indicates that a substantial number of operators are mapped to the respective device. However, the mapping ratio of Only GPU does not reach 100% because, as detailed in Section III-C, a materialize operator is exclusively executed on the CPU. Note that in Figure 8 (b), Only CPU and Only GPU exhibit the lowest latency at traffic rates of 1 MB and 16 MB, respectively. These results indicate that mapping operators to the CPU at 1 MB and to the GPU at 16 MB are optimal choices. Given that dSTREAM demonstrates similar performance to Only CPU at 1 MB and Only GPU at 16 MB, it is interesting to explore how dSTREAM dynamically assigns operators.

As shown in Figure 11, dSTREAM predominantly assigns operators to the CPU at 1 MB and shifts to the GPU at 16 MB, achieving performance similar to Only CPU and Only GPU, respectively. dSTREAM demonstrates a gradual increase in the mapping ratio to the GPU, reaching 22%, 36%, and 68%, respectively, as the traffic rate escalates from 1 MB to 16 MB. This explains dSTREAM's capability to adapt to the dynamic device preferences of each operator in response to increasing data sizes. It contrasts with Static Preference, which does not sufficiently account for these variations in device preferences across different data sizes.

2) Summary

Dynamic operator-level query mapping in dSTREAM effectively adapts to the varying device preferences of each operator in response to runtime changes in data size. This capability enhances overall performance by dynamically optimizing operator assignments, thereby mitigating the impact of traffic fluctuations.

E. SENSITIVITY ANALYSIS OF COST ESTIMATION

The estimated execution time of operators is highly sensitive to the value of the hyperparameter β, as discussed in Section III-B3. This section explains why we set β to 0.5 to optimize dSTREAM's performance. To illustrate this, we used the LR2 query, which handled an average of 4 MB of data per second under a random traffic scenario and varied β for each experiment. A β value close to 0 places more emphasis on the current execution time of each operator, while a β value near 1 relies more on historical trends for estimating execution time.

1) Sensitivity of β

Figure 12 presents the average latency of dSTREAM for three different β values (0,1, 0.5, and 0.9) across batch sequences. To smooth out significant fluctuations in latency for each batch, we calculated the average latency over ten consecutive batches. The x-axis of the graph represents the time at which the last batch of 10 consecutive batches ended. The y-axis denotes the average latency of 10 consecutive batches.

As shown in Figure 12, dSTREAM with a β of 0.1 does not converge and fluctuates significantly. This suggests that relying too heavily on current observations can lead to inaccurate estimations of execution time and sub-optimal operator mappings. On the other hand, while dSTREAM with a β of 0.9 experiences less fluctuation, it does not consistently achieve low latency. This indicates that depending solely on past trends or current data is insufficient for maintaining stable performance under varying traffic conditions. By balancing historical trends and current observations with a β of 0.5, dSTREAM effectively lowers average latency over time. This approach enables accurate cost estimation and optimal operator mapping to the most suitable device for execution.

TABLE 5. Time taken for each phase of query processing, represented in milliseconds (ms). The grey rows indicate the additional overhead introduced by dSTREAM. The total overhead duration refers to the sum of the time taken by the phases highlighted in the gray rows.

Phase	LR1	LR2	CM1	CM2
Create graph	0.128	0.109	0.154	0.109
Find shortest path	0.008	0.011	0.01	0.01
Map device	0.306	0.059	0.138	0.071
Collect metrics	0.128	0.112	0.119	0.114
Execute query	1291	431	2797	341
Total overhead duration	0.57	0.291	0.421	0.24

2) Summary

For workloads with fluctuating traffic, incorporating both past and current trends for estimating execution time provides a reliable prediction of future values. By setting β to 0.5, dSTREAM effectively estimates operator execution times and maintains low latency. This balanced method ensures consistent performance improvements over time.

F. OVERHEAD ANALYSIS

dSTREAM collects historical metrics to update the cost table and uses this table to determine the optimal query execution plan. These additional phases have been integrated into the original Spark Streaming framework. This section analyzes the time taken for each phase during query processing to assess the overhead incurred by these additional phases.

1) Execution Time and Overhead in Query Processing

Table 5 presents the time measurements of four queries (LR1, LR2, CM1, and CM2) from the experiment comparing average latency under a random traffic scenario, as discussed in Section V-B. The gray-colored rows represent the additional time incurred by dSTREAM. Overall, the extra phases in dSTREAM account for less than 1% of the total workload time. For instance, in the LR1 query, the total query execution time is approximately 1291.57 ms, with the additional overhead from dSTREAM being only 0.57 ms. This negligible overhead demonstrates that dSTREAM enhances query processing efficiency with minimal impact.

2) Summary

dSTREAM integrates two core modules into the original Spark Streaming framework to implement a dynamic operator-level query mapping scheme, considering varying data sizes and the placement of consecutive operators. These modules enhance overall query processing performance in dSTREAM while introducing less than 1% overhead.

VI. CONCLUSION

In this paper, we present dSTREAM, a stream processing system designed to dynamically adapt to each operator's device preference in discrete CPU-GPU architectures at runtime. dSTREAM efficiently updates the cost table entries online without requiring prior knowledge and determines the optimal operator-level query execution plan by selecting the mapping with the lowest cost from all possible options, with minimal overhead. Our extensive experiments with two real-world workloads show that dSTREAM consistently outperforms other comparison targets, particularly in environments with fluctuating traffic conditions and compute-intensive queries.

Our novel operator-level query mapping technique is applicable to a wide range of stream queries composed of SQL operators. Moreover, dSTREAM is designed to optimize query performance across different DSPS platforms that support operator-based queries. In future work, we plan to develop a system that determines the optimal operator-level query mapping while ensuring each query meets its deadline.

REFERENCES

[1] H. Yamaoka, K. Itakura, E. Takahashi, G. Nakagawa, J. Michaelis, Y. Kanemasa, M. Ueki, T. Matsumoto, R. Take, S. Tanie *et al.*, "Dracena: A real-time iot service platform based on flexible composition of data streams," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, 2019, pp. 596–601.

[2] A. Y. Z. Mohammadreza Hoseinyfarahabady, "Geo-distributed analytical streaming architecture for iot platforms," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 263–274.

[3] W. D. Xu, M. J. Burns, F. Cherqui, and T. D. Fletcher, "Enhancing stormwater control measures using real-time control technology: a review," *Urban Water Journal*, vol. 18, no. 2, pp. 101–114, 2021.

[4] H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of distributed stream processing frameworks for iot applications in smart cities," *Journal of Big Data*, vol. 6, pp. 1–24, 2019.

[5] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, p. 601–613.

[6] M. H. Iqbal, T. R. Soomro *et al.*, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[8] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, p. 239–250.

[9] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[10] S. N. Nag, "Technical analysis of pcie to pcie 6: A next-generation interface evolution," *World Journal of Engineering and Technology*, vol. 11, no. 3, pp. 504–525, 2023.

[11] V. Rosenfeld, S. Breß, and V. Markl, "Query processing on heterogeneous cpu/gpu systems," vol. 55, no. 1, pp. 1–38, 2022.

[12] X. Cheng, B. He, and C. T. Lau, "Energy-efficient query processing on embedded cpu-gpu architectures," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, 2015.

[13] P. Chrysogelos, "Efficient analytical query processing on cpu-gpu hardware platforms," EPFL, Tech. Rep., 2022.

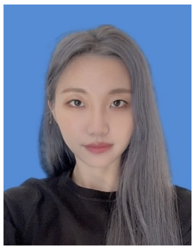
[14] K. Zhang, J. Hu, and B. Hua, "A holistic approach to build real-time stream processing system with gpu," *Journal of Parallel and Distributed Computing*, vol. 83, pp. 44–57, 2015.

[15] J. Liu, F. Zhang, H. Li, D. Wang, W. Wan, X. Fang, J. Zhai, and X. Du, "Exploring query processing on cpu-gpu integrated edge device," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4057–4070, 2022.

[16] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto, "Gasser: An auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.
[17] A. S. Foundation, "Spark-rapids." [Online]. Available: https://nvidia.github.io/spark-rapids/
[18] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li, "Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1275–1288, 2018.
[19] Z. Chen, J. Xu, J. Tang, K. A. Kwiat, C. A. Kamhoua, and C. Wang, "Gpu-accelerated high-throughput online stream data processing," *IEEE Transactions on Big Data*, vol. 4, no. 2, pp. 191–202, 2018.
[20] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data*, June 2016, pp. 555–569.
[21] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "FineStream: Fine-Grained Window-Based stream processing on CPU-GPU integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Jul 2020, pp. 633–647.
[22] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of gpus and cpus for database analytics," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, p. 1617–1632.
[23] D. Vohra and D. Vohra, "Apache parquet," *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pp. 325–335, 2016.
[24] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "Mcuda: An efficient implementation of cuda kernels for multi-core cpus," in *Languages and Compilers for Parallel Computing: 21th International Workshop*, 2008, pp. 16–30.
[25] Y. Ohno, S. Morishima, and H. Matsutani, "Accelerating spark rdd operations with local and remote gpu devices," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 2016, pp. 791–799.
[26] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for gpu computing," in *2013 International Conference on Parallel and Distributed Systems*, 2013, pp. 275–282.
[27] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.
[28] S. Lee and S. Park, "Performance analysis of big data etl process over cpu-gpu heterogeneous architectures," in *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, Apr 2021, pp. 42–47.
[29] Y. Zhang and F. Mueller, "Gstream: A general-purpose data streaming framework on gpu clusters," in *2011 International Conference on Parallel Processing*, Sep 2011, pp. 245–254.
[30] Sejeong, G. E. Oh, S. Moon, and Park, "Ml-based dynamic operator-level query mapping for stream processing systems in heterogeneous computing environments," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 226–237.
[31] K. Iyer and J. Kiel, "Gpu debugging and profiling with nvidia parallel nsight," *Game Development Tools*, pp. 303–324, 2016.
[32] Y. Cheng, Z. Hao, R. Cai, and W. Wen, "Hpc2-ars: An architecture for real-time analytic of big data streams," in *2018 IEEE International Conference on Web Services (ICWS)*, 2018, pp. 319–322.
[33] NVIDIA, "Cuda(compute unified device architecture) toolkit." [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[34] K. Group, "Opencl(open computing language)." [Online]. Available: https://www.khronos.org/opencl/
[35] J. Dittrich and J.-A. Quiané-Ruiz, "Efficient big data processing in hadoop mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2014–2015, 2012.
[36] J. S. Hunter, "The exponentially weighted moving average," *Journal of quality technology*, vol. 18, no. 4, pp. 203–210, 1986.
[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.
[39] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the Thirtieth international conference on Very large data bases*, vol. 30, 2004, pp. 480–491.
[40] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, vol. 1, pp. 1–14, 2011.
[41] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proc. VLDB Endow.*, vol. 6, no. 11, p. 1033–1044, Aug 2013.
[42] K. M. M. Thein, "Apache kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.
[43] N. Auger, C. Nicaud, and C. Pivoteau, "Merge Strategies: from Merge Sort to TimSort," Dec. 2015, working paper or preprint. [Online]. Available: https://hal.science/hal-01212839
[44] P. Tennage, S. Perera, M. Jayasinghe, and S. Jayasena, "An analysis of holistic tail latency behaviors of java microservices," in *2019 IEEE 21st International Conference on High Performance Computing and Communications*, 2019, pp. 697–705.
[45] S. Lee, Y. Jeong, K. Park, G. Jung, and S. Park, "zstream: towards a low latency micro-batch streaming system," *Cluster Computing*, vol. 26, no. 5, pp. 2773–2787, 2023.
[46] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: avoiding long tails in the cloud," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2013, p. 329–342.
[47] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb 2013.

GYEONGHWAN JUNG received the B.S degree in computer science from Sangmyung University, South Korea, in 2019, the M.S degree in computer science and engineering from Sogang University, South Korea, in 2024. He is currently a researcher with LG Electronics, South Korea. He is interested in cloud computing, resource management, and parallel and distributed systems. His current research focuses on optimizing web engine.

YEONWOO JEONG received the B.S. degree in computer software from Kwangwoon University, South Korea, in 2016 and the M.S degree in computer science and engineering in Sogang University, South Korea, in 2020. He is currently pursuing Ph.D. degree in computer science and engineering from Sogang University. His current research focuses on the way of optimizing query processing on distributed stream processing systems.



KYULI PARK received the B.S degree in computer science and engineering from Sogang University, South Korea, in 2022 and the M.S degree in computer science and engineering in Sogang University, in 2024. She is currently pursuing Ph.D. degree in computer science and engineering from Sogang University. She is interested in cloud computing, streaming system and resource management.

DONGJAE LEE received the B.S degree in computer science and engineering as part of a double major program from Sogang University, South Korea, in 2024. He is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Sogang University. His current research focuses on optimizing state management in distributed stream processing systems and its interaction with key-value store.

HONGSU BYUN received the B.S degree in computer science and engineering from Sogang University, South Korea, in 2021. He is currently pursuing the M.S. degree leading to Ph.D. degree in integrated program with the Department of Computer Science and Engineering, Sogang University. His research interests include operating systems, file and storage systems, parallel and distributed system, and high performance computing.

SUYEON LEE is a Ph.D. student in Computer Science at Georgia Institute of Technology. Her research interests are control planes of distributed systems to accelerate data-intensive applications, focusing on the intersection of networking and memory subsystems. Recent works include disaggregated memory systems, near-data/memory computing, and CXL. She obtained her B.S. and M.S. degrees from Sogang University, South Korea.

SUNGYONG PARK (Member, IEEE) is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.