Exploring Query Processing on CPU-GPU Integrated Edge Device

Jiesong Liu, Feng Zhang, Hourun Li, Dalin Wang, Weitao Wan, Xiaokun Fang, Jidong Zhai, and Xiaoyong Du

Abstract—Huge amounts of data have been generated on edge devices every day, which requires efficient data analytics and management. However, due to the limited computing capacity of these edge devices, query processing at the edge faces tremendous pressure. Fortunately, in recent years, hardware vendors have integrated heterogeneous coprocessors, such as GPUs, into the edge device, which can provide much more computing power. Furthermore, the CPU-GPU integrated edge device has shown significant benefits in a variety of situations. Therefore, the exploration of query processing on such CPU-GPU integrated edge devices becomes an urgent need. In this article, we develop a fine-grained query processing engine, called FineQuery, which can perform efficient query processing on CPU-GPU integrated edge devices. Particularly, FineQuery can take advantage of both architectural features of edge devices and query characteristics by performing fine-grained workload scheduling between the CPU and the GPU. Experiments show that on TPC-H workloads, FineQuery reduces 42.81% latency and improves 2.39× bandwidth utilization on average compared to the implementation of using only GPU or CPU. Furthermore, query processing at the edge can bring significant performance-per-cost benefits and energy efficiency. On average, FineQuery at the edge brings 21× performance-per-cost ratio and 4× energy efficiency compared with processing the data on a discrete GPU platform.

Index Terms—CPU, GPU, integrated architecture, edge device, query processing

1 Introduction

Massive data are generated at the edge every day. These data also need to be processed and managed. However, edge devices usually have low computing power. The edge processors have fewer cores with low frequency to retain power efficiency. Accordingly, processing data at the edge poses great pressure for management of data. Fortunately, hardware vendors embedded a heterogeneous coprocessor at the edge, enhancing the data processing capability of these edge devices. We show an example of Nvidia Jetson XAVIER NX [1] in Fig. 1, which is a full computing system integrated with an embedded GPU. Specifically, it has extremely low power for energy efficiency, and it has high performance-per-cost ratio. Therefore, it is not only essential but also of great benefits to explore how to perform efficient data management on such CPU-GPU integrated edge devices.

Performing query processing on such CPU-GPU integrated edge devices introduces has great benefits. First, network transmission overhead of transmitting data to the cloud can be greatly reduced because most data are processed on edge devices with powerful embedded GPUs. Second, PCIe transmission cost, especially unnecessary copies in both the CPU memory and the GPU memory of the discrete CPU-GPU architecture, can be avoided. The integrated edge device uses a unified shared memory for both the CPU and the embedded GPU, instead of the discrete memory design for HPC GPU machines. Third, a query can be divided into several operators, and different operators can exhibit different performance behaviors on CPU and GPU. The unified design provides a new opportunity in distributing these operators to CPU and GPU adaptively for fine-grained collaboration.

Query processing on CPU-GPU integrated edge devices is challenging, though it can bring massive benefits. First, we need to utilize the unified shared memory with query optimization. The memory bandwidth limitation on edge devices still exists, and the bandwidth provided by edge devices is lower than HPC servers. Besides, CPU and GPU have diverse memory access patterns and can incur different pressure on unified shared memory and devices. Current query processing solutions do not consider these complicated situations on discrete architectures. Second, a comprehensive performance model needs to be built to assist query optimization at the edge. Operators can have different preferred devices, such as CPU and GPU. We need to build a performance model to measure and formulate the effectiveness of different processing strategies from various

- *Jiesong Liu, Feng Zhang, Hourun Li, Dalin Wang, Weitao Wan, Xiaokun Fang, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China, Beijing 100872, China. E-mail: {liujiesong, fengzhang, lihourun, sxwangdalin, wanweitao, fangxiaokun, duyong}@ruc.edu.cn.*
- *Jidong Zhai is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhaijidong@tsinghua.edu.cn.*

Manuscript received 14 Oct. 2021; revised 16 Apr. 2022; accepted 18 May 2022. Date of publication 26 May 2022; date of current version 23 Aug. 2022.
This work was supported in part by the National Natural Science Foundation of China under Grants 61732014, 62172419, U20A20226, and 62072459, in part by the Beijing Natural Science Foundation under Grant 4202031, and in part by the Tsinghua University-Peking Union Medical College Hospital Initiative Scientific Research Program under Grant 20191080594. This work was also supported by CCF-Tencent Open Research Fund.
(Corresponding author: Feng Zhang.)
Recommended for acceptance by P. D'Ambra.
Digital Object Identifier no. 10.1109/TPDS.2022.3177811

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

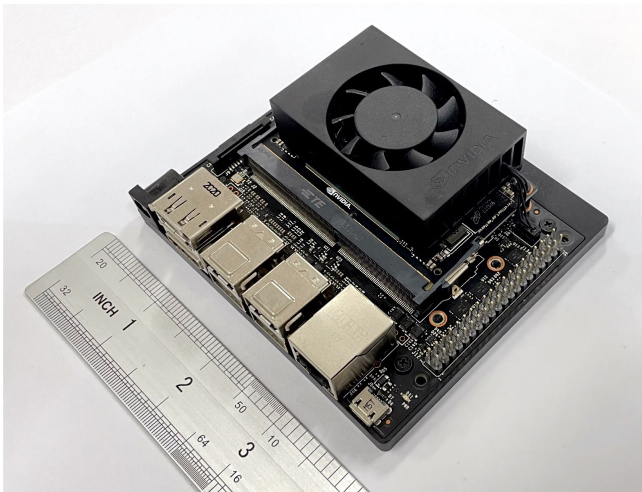


Fig. 1. Nvidia edge device: Jetson XAVIER NX.

perspectives. Third, task co-running combined with query characteristics is another difficulty. To achieve high efficiency, tasks are divided into subtasks, and we need to develop a novel subtask co-running strategy with high parallelism for CPU-GPU integrated edge devices.

Many studies have been conducted to utilize the CPU-GPU integrated architectures in data management applications [2], [3], [4]. However, none of these works explored query processing on integrated edge devices. For instance, He *et al.* [2] studied the *hash join* algorithm in database domain by partitioning the workload in each stage to obtain the optimal CPU-GPU co-running performance, but they did not consider the other operators. Zhang *et al.* [3], [4] developed a stream processing engine on integrated architectures, called FineStream, which can perform fine-grained workload partitioning for CPU and GPU. However, FineStream targets stream situation, and cannot handle the complicated queries among edge devices.

To address the aforementioned challenges, we develop a fine-grained query processing engine, called FineQuery, which can fully utilize the CPU-GPU integrated edge device for query processing. First, FineQuery can utilize the unified memory by enabling the CPU and GPU to access memory in a suitable pattern. Second, we propose a performance model. Based on the combination of statistic data collected in preprocessing and input data attributes, the model can indicate the processors on which the operators should be executed. Third, we develop a sub-task module to split a specified query into several operators. The sub-task module can distribute the operators into different sub-tasks, which can then be processed in parallel on CPU and GPU. Our preliminary work has been presented in FineQuery [5], which provides only a simple design without considering the edge situation. Compared with the preliminary work [5], we show new insights and optimizations. Besides, new benchmarks and datasets have been involved in experiments.

FineQuery can be applied in the database management systems at the edge [6], [7], [8]. As a new technology, FineQuery helps databases to accelerate queries on edge devices. The databases deployed at the edge in the network can use the CPU-GPU integrated edge device to speed up queries. At the same time, FineQuery is close to the source of information to facilitate data collection.

We evaluate FineQuery on one of the most powerful integrated edge devices, Nvidia JETSON AGX XAVIER, and an AMD's integrated platform, A10-7850K, covering different situations. Experiments show that FineQuery reduces 42.81% latency and improves 2.39× bandwidth utilization over the current query processing method on the integrated architectures. The same queries performed on the discrete architecture take 59.29% more processing time compared to the FineQuery version. Also, FineQuery can bring 21× performance-per-cost benefits with 4× energy efficiency.

As far as we know, FineQuery is the first work exploring query processing on CPU-GPU integrated edge devices. The contributions of this paper are summarized as follows.

- We present FineQuery, which is the first framework enabling fine-grained query processing on the CPU-GPU integrated edge device.
- We point out that fine-grained continuous operator model can be performed on integrated architectures. We unveil the challenges for fine-grained query processing at the edge and provide a series of solutions.
- We show a series of optimizations of FineQuery on edge devices, and demonstrate the benefits of utilizing integrated architectures for query processing from different perspectives.

2 Background and Premises

In this section, we show the background and premises of query processing on the CPU-GPU integrated edge devices.

2.1 Edge Devices

An edge device is a device that can serve as an entry into the networks of enterprises or service providers, and edge computing is a computing paradigm that moves data processing and storage close to the data sources. Edge computing becomes increasingly popular because it can save both transmission time and bandwidth. However, the computing capacity of edge devices is usually low.

CPU-GPU Integrated Edge Device. The CPU-GPU integrated edge device can solve the shortcomings of insufficient computing power of edge devices. First, the CPU-GPU integrated edge device still maintains the characteristics of low power consumption, but with high computing capacity. For example, the Nvidia integrated edge device, Jetson Xavier NX, fuses an ARM CPU with an Nvidia Volta GPU on the same chip, providing 14 TOPs computing capacity with only 10W power [1]. Second, both the CPU and the GPU of the integrated edge device share the same unified memory, which avoids PCIe transmission between CPU and GPU, and at the same time provides more opportunities for fine-grained query processing. Third, the integrated edge device still retains the edge benefits. It has low price and low space overhead. Currently, the CPU-GPU integrated edge devices are widely used in different domains, such as robotics and edge computing [9].

CPU-GPU Integrated Architecture Design. Fig. 2 shows an overview of the CPU-GPU integrated architecture design.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

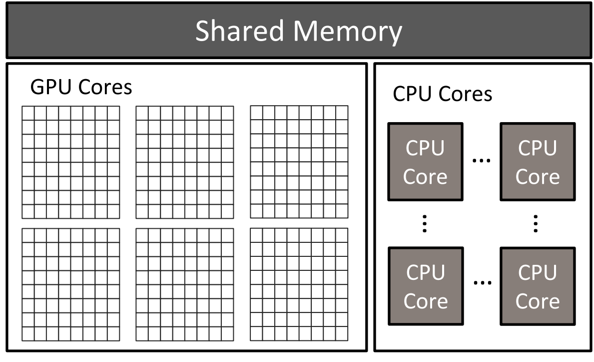


Fig. 2. Overview of the integrated edge device.

On the integrated architecture, both the CPU and the GPU are integrated on the same chip, sharing the same system main DRAM. Different from the discrete CPU-GPU architecture, a memory controller is used to schedule the memory accesses to the unified memory from both the CPU and the GPU. The unified shared memory, accessible to both CPU and GPU, is the most appealing characteristic of the integrated edge device. It brings new opportunities for fine-grained cooperation between CPU and GPU.

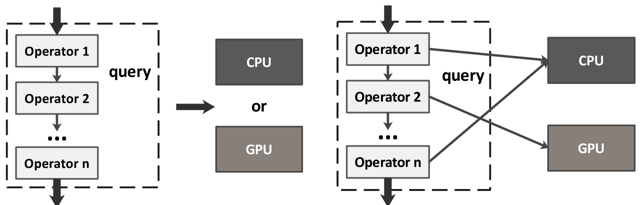
Opportunities. First, in an application environment with restrictions on power consumption, equipment area, and cost, the CPU-GPU integrated edge device still applies. Second, GPU can process the generated data directly without PCIe data transmission. Third, the operators of a query can be placed on their preferred devices and process the same data stored in the unified shared memory.

2.2 Query Processing

The process of extracting user-required data from a database is known as query processing. In big data era, relational database still plays an important role in data management [10], [11], [12], [13]. The core of the relational database is the relational data model. The standard language for relational databases is structured query language (SQL). Relational databases manage and maintain relational data through SQL to help users express their queries. Accordingly, we mainly explore SQL queries in this paper.

SQL Operators. A SQL operator is a reserved operation in SQL query statement. SQL queries are made up of operators, including *select*, *project*, *join*, *divide*, *union*, *except*, *intersection*, and *cartesian product*. Among these operators, *select*, *project*, *union*, *except*, and *cartesian product* are five basic operators. The other operators can be exported and defined through these basic operators. These operators are used for data maintenance.

GPU-Based Query Processing. GPUs have been used in many database engines to accelerate database query processing [14], [15], [16]. For example, GPUQP [14] is a relational GPU query processing engine, which accelerates operators on GPUs. MapD [15] is an efficient GPU-powered database platform, which can leverage massive GPU cores in processing SQL queries. However, these systems can only use GPUs to process a whole query, without further fine-grained workload partitioning on different devices. OmniDB [16] is an OpenCL-based heterogeneous query processing engine, built on GPUQP. OmniDB can be applied to various devices, including both CPU and GPU, but with only simple operators evaluated. Moreover, none of these works focus on edge devices.



(a) Bulk-synchronous parallel model. (b) Continuous operator model.

Fig. 3. Analysis of different processing granularities.

TPC-H Benchmark. A data management benchmark is the execution of a set of standard queries to measure the performance of a query processing engine. The Transaction Performance Management Committee (TPC) is currently one of the most well-known benchmark standardization organizations for data management system evaluation. In the past two decades, this organization has released a number of database evaluation benchmarks, including TPC-A, TPC-D, TPC-H, and TPC-DS, which have been widely used in the data management domain. Among these benchmarks, TPC-H is a decision-making benchmark, which simulates a set of business queries. The main purpose of TPC-H is to evaluate the decision-making support capabilities of specific queries and to measure the capabilities of the query processing engine in data mining, data analytics, and data processing. TPC-H consists of 22 complicated queries and has been widely used in previous research [17], [18]. In this paper, we use TPC-H in our experiments.

3 Motivation

We in this section show the motivation of fine-grained query processing on the CPU-GPU integrated edge device. We first revisit the current query processing models. Then, we exhibit our basic idea, and show the challenges in enabling query processing at the edge.

3.1 Preliminaries

Based on the processing granularity, the query processing models are classified into two categories: bulk-synchronous parallel model and continuous operator model [19], as shown in Fig. 3.

Bulk-Synchronous Model. The bulk-synchronous parallel model has been employed by traditional query processing engines, which distributes the whole query of operators on the same device, as shown in Fig. 3a. This design is suitable for discrete CPU-GPU architectures because frequent data communication between devices via PCIe can incur severe time overhead. Previous studies [14], [20] adopt this design to avoid PCIe data transmission between CPU and GPU.

Continuous Operator Model. The continuous operator model can distribute operators of the same query to different devices, as shown in Fig. 3b. Experiments show that different operators can have various device preferences [3], [4]. With the integrated architecture fusing different devices on the same chip, we have new opportunities to put operators on different devices, since the PCIe limitation has been eliminated. Please note that the continuous operator model is

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

not appropriate for discrete CPU-GPU design, because this model requires frequent data communication between CPU and GPU.

3.2 Fine-Grained Query Processing on CPU-GPU Integrated Edge Device

Basic idea: We apply a continuous operator model on CPU-GPU integrated architectures, with operators assigned to CPU and GPU according to their characteristics.

Our fundamental idea is to use the continuous operator model on CPU-GPU integrated architectures, as shown in Fig. 3b. Performing fine-grained query processing on integrated architectures has the following benefits. First, the PCIe data transmission bottleneck has been removed, which provides more opportunities to CPU and GPU for fine-grained co-processing. Second, CPU and GPU share the same unified memory so that they can operate on the same data together simultaneously. Third, device preference for different operators can be guaranteed since the most suitable device for different operators varies. Furthermore, we need to provide the appropriate solution given the limited computing resources.

Currently, no continuous operator model has been thoroughly investigated on CPU-GPU integrated edge devices. OmniDB [16] is a framework that can distribute operators to different devices on integrated architectures. However, OmniDB has not been evaluated with complicated queries such as TPC-H. FineStream [3], [4] is a novel stream processing framework utilizing the continuous operator model, but FineStream targets stream processing situations. Therefore, exploring the performance of the continuous operator model on CPU-GPU integrated architectures for query processing is both beneficial and necessary.

3.3 Challenges

Enabling fine-grained query processing on integrated edge devices requires to solve the following three challenges.

Challenge 1: Query optimization with CPU-GPU shared unified memory. Performing optimization towards the shared unified memory of integrated edge devices is challenging. First, although the integrated design eliminates the data copy via PCIe by providing unified physical memory, the memory bandwidth bottleneck still exists. Second, different parallel topology decisions and query execution strategies can cause different pressure on both shared global memory and devices dynamically. Third, query optimization needs to consider not only query characteristics but also hardware features. This highlights the significance of proper organization and resource allocation.

Challenge 2: Building a comprehensive performance model to help with query optimization. Utilizing parallel processing technology can potentially accelerate the SQL query at the edge, as discussed in Section 2.2. However, not all the operators are suitable to be carried on a single device, such as GPU or CPU. Partial operators prefer a certain kind of device. Besides, not all SQL queries can obtain significant performance improvement via fine-grained query processing on integrated architectures. In order to build a scalable and practical system at the edge, we need to establish a comprehensive performance model to evaluate and formulate the effective and practical processing strategy from various perspectives. Besides, complex practical situations and numerous operator combinations make developing a comprehensive performance model even more complicated.

Challenge 3: Sub-tasks co-running with query characteristics considered. For those SQL queries processing a large amount of data, there are great potential opportunities to accelerate queries by utilizing parallel processing on edge devices. To achieve this goal, we develop a new technique called *sub-task co-running*. Our method is responsible for appointing the execution order and operator-device mapping. However, this design is far more complicated than expected because it has to take a large number of influencing factors into consideration. First, the relation between different operators can be constantly changing in practice. Our method needs to analyze the irrelevant or dependent relation between two operators to determine whether they can be executed in parallel. Accordingly, operators with dependent relations cannot be executed arbitrarily. Second, data characteristics can influence the operator-device mapping. For example, massive atomic operations on *double* data type do not suit the GPU device. Third, the architectural characteristics also need to be considered, since the architectural characteristics influence the resource distribution due to the CPU and GPU architectural differences, such as computing capacity and cache size. Therefore, how to perform such a sub-task co-running on integrated edge device is challenging.

Overall, enabling fine-grained query processing on integrated edge devices is rewarding, but full of difficulties.

4 FineQuery Design

In this section, we propose a query processing engine, called FineQuery, for fine-grained SQL query processing on CPU-GPU integrated edge devices.

General Design. The general design of FineQuery is shown in Fig. 4. FineQuery consists of three major parts, which are sub-task module, performance model module, and dispatcher module. The sub-task module splits a certain SQL query into several different operators and then forms several independent sub-tasks based on the performance model for all the operators. The module of performance model builds operator-device mapping and adjusts the preliminary sub-task distribution strategy. The dispatcher module assigns operators to devices according to the performance model. The three modules work together for fine-grained query processing. The detailed design of the three modules is as follows.

- The sub-task module splits a SQL query into several separate operators. Then, it organizes the operators into a DAG and partitions the DAG into different sub-tasks that can be processed simultaneously and independently.
- The module of performance model first decides the devices on which the operators are processed based on the combination of collected statistic data and practical data attributes. Subsequently, it assesses the performance of present sub-task decisions in

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

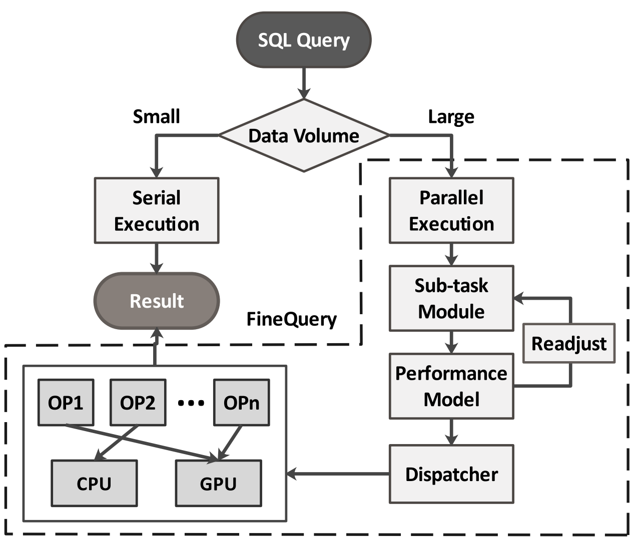


Fig. 4. FineQuery overview.

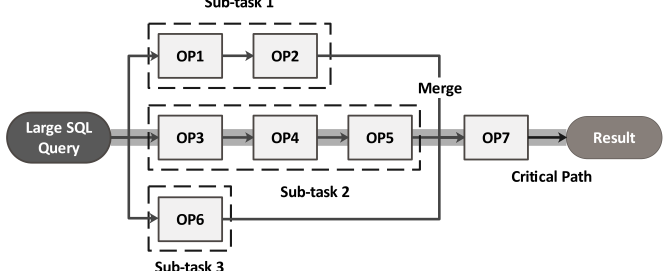


Fig. 5. An example of sub-task structure.

multi-dimensions, such as parallelism degree, hardware resource, and speedup ratio. Finally, it adjusts the sub-task strategy to achieve a suitable performance according to the previous analysis.

- The dispatcher module assigns data to operators on GPU or CPU according to the strategy made by the performance model.

Next, we discuss the general idea of FineQuery, including its workflow, sub-task co-running, and solutions to challenges.

Workflow. We show the workflow of the three major components as follows. When the program starts to process a SQL query, it judges whether it needs to be processed in parallel. If the data size is small, we process the data with only the CPU in FineQuery. Otherwise, FineQuery uses the above three modules to process input data. In detail, first, the sub-task module splits the whole query into separate operators, analyzes the relations between these operators, and distributes them into several sub-tasks for parallel processing. Second, the module of performance model makes a preliminary operator assignment strategy and subsequently assesses the feasibility and performance of current decision from different perspectives. Then, it readjusts the decision for higher performance. Third, according to the strategy made previously, the dispatcher module assigns the operators to the CPU or the GPU, and assigns corresponding data to each operator. After the process of the three above modules, FineQuery formally starts to execute each operator in parallel and generates the result.

Sub-Task Module and DAG. The SQL query plan can be described as a DAG by representing each single operator as a node. There is a directed edge between two arbitrary nodes if and only if data dependence relation exists between two nodes' corresponding operators. Similar to [3], [4], we use a DAG to describe the topology of a SQL query.

After constructing the corresponding DAG, we next perform a topological sorting algorithm on it, by which we can obtain the sub-task model of the SQL query. The number of sub-tasks is equal to the number of independent parts in the result of the topological sorting. The main reason for adopting the sub-task model is to figure out the parallel structure of the SQL query. Operators in different sub-tasks can be processed in parallel because no data-dependent relation exists between them. Note that operators in a single sub-task must be executed in a certain and restricted order because of data dependence. For example, in Fig. 5, the processing order of sub-task2 is $< OP3, OP4, OP5 >$, which cannot be disturbed. Next, we use Fig. 5 to explain our idea in detail.

We can see that there are three sub-tasks in the single DAG, which can be processed in parallel. Then, we need to merge the intermediate results of the three sub-tasks before executing OP7. Next, we use the terminology of *critical path* in [3], [4] to represent the longest stretch of dependent activities, so that we can measure the time required to finish the query. Similarly, in a SQL query, the sub-task with the most operators is more likely to become part of the critical path and determines the execution time of the whole query. Therefore, in the example of Fig. 5, sub-task2 and OP7 form the critical path and the total execution time is equal to the whole execution time of OP7 and sub-task2.

The main goal of using the DAG terminology is to help us figure out the critical path and to optimize the design effectively. Because in parallel execution, the time for the other sub-critical paths is shorter than the duration of the DAG execution, which means that we can focus on the critical path first to improve the DAG processing efficiency.

Solutions to Challenges. FineQuery can solve the challenges mentioned in Section 3.3. For the first challenge, FineQuery does not calculate the bandwidth or adjust the strategy in real time, because of the overhead. In contrast, we solve this challenge by optimizing our sub-task structure with the performance model. FineQuery adopts a strategy that can distribute the workloads on CPU and GPU evenly, which ensures the parallelism being carried within limited bandwidth. For the second challenge, the main goal of the performance model is to adjust the query execution structure and there are additional amounts of factors needed to be taken into consideration. Hence, we adopt three specified sub-models to constitute our overall performance model, which are sub-task model, pipeline model, and parameter tuning model, respectively. The three concrete sub-models solve the corresponding specific sub-problems. For the third challenge, after splitting a certain SQL query into different operators, we represent the whole SQL query as a DAG. In processing this DAG, we employ the topological sort algorithm to form a preliminary parallel structure, in which operators are distributed into several sub-tasks. By utilizing the performance model to take comprehensive factors into consideration, we further improve the parallel structure and can address the first challenge.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

5 Performance Model

Guideline: Building an efficient and lightweight model is both necessary and important for FineQuery. The strategy and organization we use are based on performance models, and different strategies can lead to different performance behaviors in practice, especially when FineQuery processing large amount of data. Therefore, a simple and light model not only guarantees the performance of FineQuery, but also keeps the modeling cost marginal.

We show our performance model in FineQuery in this section. FineQuery achieves performance acceleration by utilizing fully parallel and fine-grained processing. In detail, FineQuery's performance model is composed of three specific sub-models, which are sub-task model, pipeline model, and parameter tuning model. For the first two models, they focus on accelerating processing separately from inter sub-task and inner sub-task to achieve higher parallelism. In contrast, the parameter tuning model readjusts the detailed parameters to make the strategy more efficient.

5.1 Sub-Task Co-Running Model

We need to develop a sub-task co-running model because the DAG structure we obtain after employing topological sorting on DAG cannot be used directly. The reason is that the resources of the integrated architecture are not unlimited. As discussed in Section 4, the independent operators can execute in parallel and exceed the memory bandwidth limit. Accordingly, extra resource limitation needs to be considered and further sub-task co-running model needs to be designed.

For illustration, in Fig. 5, the three sub-tasks 1,2,3 can be processed in parallel initially. Ideally, we assume that there is enough bandwidth. CPU and GPU can simultaneously process the three sub-tasks without interaction, and the total execution time is equal to the sum of the time of sub-task2 and OP7 executed alone. However, in practice, due to the integration of both CPU and GPU devices, the bandwidth utilization is even tighter than that on discrete architectures. When the required bandwidth of the co-running sub-tasks exceeds the maximum bandwidth of the server, the total execution time extends accordingly.

To estimate the execution time, we assume that the ideal execution time with enough bandwidth is t_{total}. We define the ratio of exceeding bandwidth $r_{excessive}$ as the percentage of time the required bandwidth $B_{required}$ exceeds the maximum available bandwidth B_{max} under ideal conditions. We can express the execution time t'_{total} in Equation (1)

$$t_{total}' = r_{excessive} \cdot t_{total} \cdot \frac{B_{required}}{B_{max}} + (1 - r_{excessive}) \cdot t_{total}. \tag{1}$$

We need to further estimate $B_{required}$ for Equation (1). To calculate $B_{required}$, we need to estimate the bandwidth for each co-running sub-task, $B_{sub-task_i}$, which can be estimated by executing the sub-task on CPU and GPU. Assume the number of co-running sub-tasks is $n_{sub-task}$. Then, $B_{required}$ can be expressed in Equation (2)

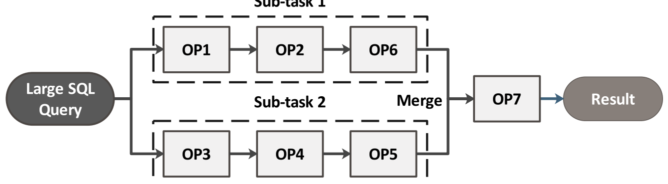


Fig. 6. Reconstructed sub-task structure.

$$B_{required} = \sum_{i=1}^{n_{sub-tasks}} B_{sub-task_i}. \tag{2}$$

Assume there are $n_excessive$ parts exceeding the maximum bandwidth. We can use Equation (3) to estimate t_{total}', combining Equations (1) and (2)

$$t_{total}' = \sum_{i=1}^{n_excessive} \left(r_{excessive_i} \cdot \frac{\sum_{i=1}^{n_{sub-tasks}} B_{sub-task_i}}{B_{max}} \cdot t_{total} \right) + \left(1 - \sum_{i=1}^{n_excessive} r_{excessive_i} \right) \cdot t_{total}. \tag{3}$$

Optimization. The uneven workload distribution for operators can lead to performance decline and prolong the execution time. For example, compared with the over-saturated state in Fig. 5, redundant resources can exist after OP4 in sub-task2. To better utilize bandwidth, we move the execution of OP6 to the tail of sub-task2, as shown in Fig. 6. To estimate the total time, we assume that the ideal execution times of sub_task_1, sub_task_2, and sub_task_3 are t_1, t_2, and t_3 respectively with enough bandwidth. Accordingly, the total execution time can achieve a new lower bound, t''_{total}, shown in Equation (4)

$$t_{total}'' = \max(t_1 + t_3, t_2) + t_{op_7}. \tag{4}$$

Generally, we can perform better sub-task scheduling by moving operators from the sub-task with full bandwidth utilization into the sub-task with surplus bandwidth. For example, in Fig. 7, OP1 and OP2 form a stage. To model the whole procedure, we quantitatively calculate t''_{total} as shown in Equation (5). Here, we introduce a new concept, called *stage*, to further divide the sub-task. Assume that there are n_{stages} stages in a sub-task structure. Each $stage_i$ has n_{stage_i,sub_tasks} sub-tasks. Besides, sub_task_j has its ideal execution time t_j with enough bandwidth. After re-constructing the sub-task structure, the numbers of operators in sub_task_j can change as well. We assume the execution time change caused by the subtle transformation as $t_{reshape_j}$ for each sub_task_j. Finally, the left operators have their execution time, t_{left}. Hence, the total time after optimization can be represented as Equation (5). The final optimized time can be represented as $min(t_{total}', t_{total}'')$

$$t_{total}'' = \left(\sum_{i=1}^{n_{stages}} \max_{1 \le j \le n_{stage_i,sub_tasks}} (t_j + t_{reshape_j}) \right) + t_{left}. \tag{5}$$

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

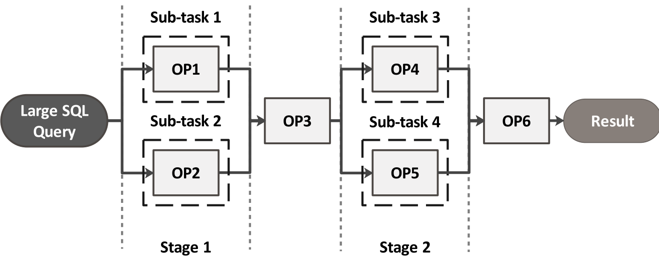


Fig. 7. Sub-task structure with stages.

5.2 Pipeline Model

We use the concept of critical path to analyze the entire execution time. In this part, we use pipeline to further shorten the execution time of critical path.

We use Fig. 8 to illustrate the basic idea of pipeline. Suppose OP1 is $select$ while OP2 is sum. The data dependency between OP1 and OP2 originally exists. We find that we can execute the sum task without completely $selecting$ all the target data. In order to realize our idea, we divide the data to be selected in OP1 into several blocks. The following sum operator is able to start to work after one data block being processed in OP1. Ideally, the time of OP1 and OP2 can be reduced to the sum of the processing time of OP1 and one data block (for OP2).

Assume that we split the whole data processed by OP1 into n blocks. The original execution times of OP1 and OP2 are t_1 and t_2 respectively. The ideal execution time is defined as $(t_1 + \frac{1}{n} \cdot t_2)$. Assume the bandwidth of the serial execution of OP1 and OP2 is $bandwidth$, then the improved bandwidth $bandwidth'$ can be defined in Equation (6). Note that in practice, how to determine the data blocks is intricate. For example, when the difference of processing time between OP1 and OP2 is large, the pipeline optimization effect will be amortized

$$bandwidth' = \min\left(\frac{t_1 + t_2}{t_1 + \frac{1}{n} \cdot t_2} \cdot bandwidth, B_{max}\right). \tag{6}$$

5.3 Parameter Tuning Model

In parameter tuning model, we evaluate our strategy and further improve it from parameter adjustment perspectives. Parameter tuning model mainly relates to operator distribution and device allocation.

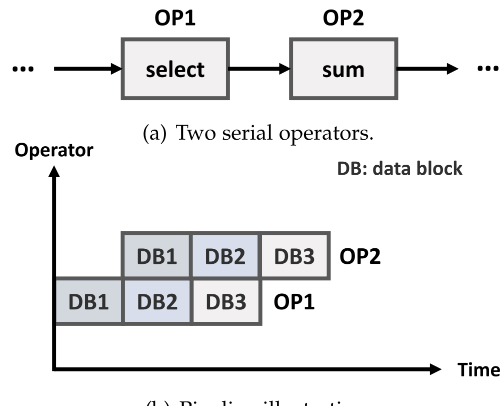


Fig. 8. Pipeline model.

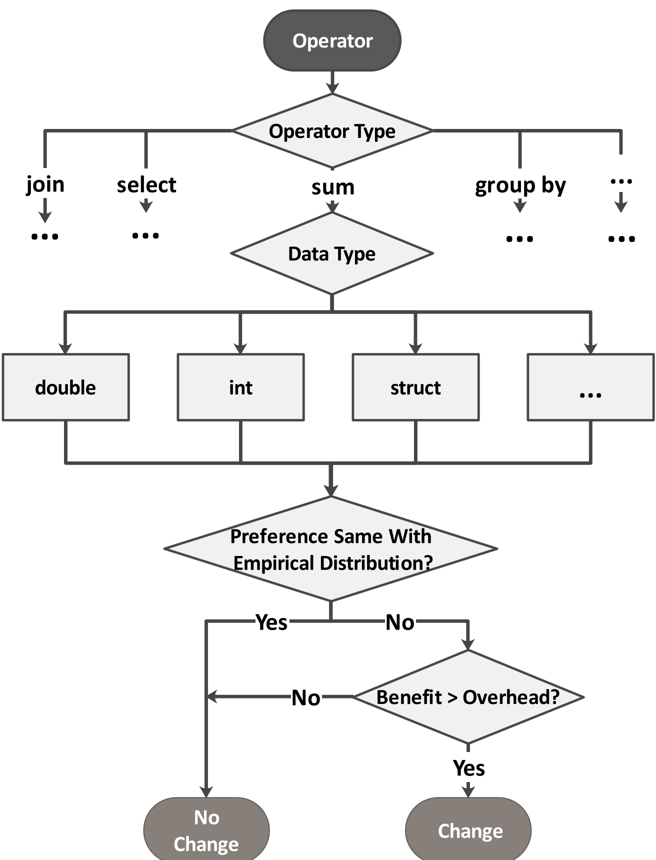


Fig. 9. Decision tree for operator distribution tuning.

For the first operator distribution, we find that operators exhibit different device preferences. For example, *selection* and *sum* operators have relatively high parallelism so they usually tend to be distributed on GPU. However, specified SQL queries may not obey the empirical distribution. In detail, certain data types are not suitable to be processed in parallel on GPUs. For instance, calculating the sum of data in *double* with synchronization can bring huge cost on GPU, but not on CPU. Out of intricate situations, we employ a decision tree to embody our operator distribution solution in our parameter tuning model, as elaborated in Fig. 9.

For the second device allocation, although we improve the sub-task structure in the sub-task model, we still cannot fully utilize the bandwidth all the time. Hence, in those phases, when the number of operators being processed is small, we can moderately increase the number of threads for each operator. Because the speedup ratio does not increase linearly with the number of threads, simply dividing the total number of threads by the number of operators in a phase can lead to a serious performance decline. We should guarantee the basic quantity of threads for each operator. Thus, we set a basic quantity for each operator in advance and decide whether to increase the quantity according to the practical processing situation. We assume $operator_i$ needs at least n_i (default 64) threads to guarantee the performance, and there are $n_{operators}$ operators processed on CPU or GPU and $n_{threads}$ threads in CPU or GPU. Hence, the optimized thread number n_{thread_i} for $operator_i$ is shown in Equation (7)

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

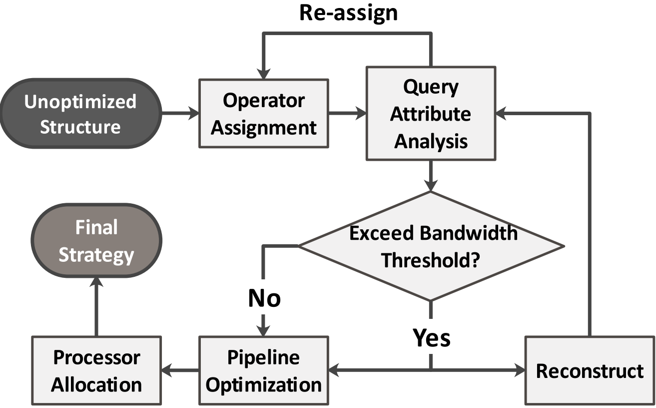


Fig. 10. Workflow of FineQuery in using the performance model.

$$n_{thread_i} = \max\left(n_i, \frac{n_{threads}}{n_{operators}}\right). \quad (7)$$

In general, we show the workflow of FineQuery in using the performance model in Fig. 10. The performance model takes unoptimized structures as input, and then follows the *sub-task co-running* and *parameter adjusting* sub-models. Specifically, we perform *operator assignment* and conduct *query attribute analysis* on the assignment. Based on the analytic results, we either move forward to the next step or re-assign the complicated operators. There is only one step left before we can move on to the next sub-model. That is, we check the bandwidth requirement of the current operator assignment and make corresponding adaptations. We then conduct a pipeline optimization following the *pipeline model*, and start the FineQuery engine by allocating the specific operators to their preferred processors.

6 Optimization

In this section, we show our explorations on further optimizing query processing on integrated edge devices.

6.1 Unified Memory

The unified memory is a distinguishing feature for the integrated edge device, and we in this part show our memory optimization at the edge.

Analysis of Unified Memory. The unified shared memory is a distinctive feature over the heterogeneous CPU-GPU discrete architecture. On discrete architecture, CPU and GPU access their separate memory, and the implementation of unified memory accessible to both CPU and GPU needs to be supported by a page migration engine. Due to the discrete CPU-GPU memory design, missing memory access can incur frequent data reallocation and migration [21], [22], which can cause serious performance degradation. However, the integrated edge device does not have this problem. As discussed in Section 2.1, both CPU and GPU of the edge device use the same unified shared memory, without PCIe data transmission.

Unified Memory Utilization. FineQuery utilizes the unified shared memory for query processing on edge devices. After allocating the data in the unified memory space, both CPU and GPU can operate the memory objects with fine-grained cooperation. The detailed process of FineQuery utilizing unified memory is as follows. First, FineQuery uses the CUDA API *cudaMallocManaged()* to allocate a buffer of unified memory for the input data. Second, FineQuery loads the input data into the unified memory buffer for CPU and GPU. Third, FineQuery performs fine-grained query processing according to the performance model.

6.2 Integrated Hierarchy

The integrated hierarchy design of edge devices fuses the CPU and GPU characteristics. In detail, CPU features a memory hierarchy to efficiently access data. In particular, the caches provide high-speed data retrieval from the memory, and data access becomes slower on peripheral storage. Likewise, the GPU also has local registers for each thread and independent L1 and L2 caches for streaming multiprocessors. Specifically, GPU provides a controllable cache, called *shared memory*, for threads within a CUDA thread block. The integrated edge device further provides a unified memory accessible to both CPU and GPU. FineQuery needs to consider these different characteristics between the CPU and GPU to achieve high efficiency. Accordingly, these optimizations are oriented towards the difference between CPU and GPU rather than different platforms.

Analysis of Integrated Hierarchy. Our general design is to allocate frequently accessed data to the upper layer of the memory hierarchy. We have an observation that the atomic *add* operation on the GPU kernel can incur a high conflict rate between threads. Specifically, if too many atomic *add*s are made to the *sum* operator, the parallel version of the *for* loop can lead to severe performance degradation. The reason is that one thread can prevent the other threads from accessing *sum* under the atomic locking mechanism. Such mechanism and context switching can have a negative impact on performance when capturing and releasing the *add*. Therefore, our basic idea is to narrow the access range of each thread in order to perform computation in shared memory.

Integrated Hierarchy Utilization. We have the following design. For a set of key-value objects, we need to perform operations on the objects whose keys satisfy a given condition. To fully utilize the memory hierarchy of the integrated design, we store the data that are frequently used in the upper-level storage. The utilization of CUDA shared memory for threads in the same block is the focus of FineQuery. Specifically, we allocate an array buffer with the size of the number of threads located in each block. At the running time, we check every key in each thread in parallel and set the key to 0 if that object does not satisfy the specified condition. Finally, we perform a parallel reduction to generate the sum of the values in the array.

6.3 Programming Model

In this part, we show our considerations from the perspective of programming models.

Analysis of Programming Models. OpenCL is a cross-platform standard for parallel computing in heterogeneous systems, which is able to control both CPU and GPU. In contrast, CUDA is specially designed for Nvidia GPUs. CUDA leaves out complicated details in GPU programming and provides programmers with simpler interfaces. Not all

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
TPC-H Queries Used in Evaluation [23]

Query	Detail
Q1	select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice (1 - l_discount)) as sum_disc_price, sum (l_extendedprice (1 - l_discount) (1 + l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count() as count_order from lineitem where l_shipdate <= date '1998-12-01' - interval '90' day (3) group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;
Q3	select l_orderkey, sum(l_extendedprice (1 - l_discount)) as revenue, o_orderdate, o_shippriority from customer, orders, lineitem where c_mktsegment = 'BUILDING' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-15' and l_shipdate > date '1995-03-15' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate;
Q6	select sum(l_extendedpricel_discount) as revenue from lineitem where l_shipdate >= '1994-01-01' and l_shipdate < '1994-01-01' + INTERVAL '1' YEAR and l_discount between 0.06 - 0.01 and 0.06 + 0.01 and l_quantity < 24;
Q12	select l_shipmode, sum(case when o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH' then 1 else 0 end) as high_line_count, sum(case when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH' then 1 else 0 end) as low_line_count from orders, lineitem where o_orderkey = l_orderkey and l_shipmode in ('MAIL', 'SHIP') and l_commitdate < l_receiptdate and l_shipdate < l_commitdate and l_receiptdate >= date '1994-01-01' and l_receiptdate < date '1994-01-01' + interval '1' year group by l_shipmode order by l_shipmode;
Q14	select 100.00 sum(case when p_type like 'PROMO%' then l_extendedprice (1 - l_discount) else 0 end) / sum(l_extendedprice (1 - l_discount)) as promo_revenue from lineitem, part where l_partkey = p_partkey and l_shipdate >= date '1995-09-01' and l_shipdate < date '1995-09-01' + interval '1' month;
Q17	select sum(l_extendedprice) / 7.0 as avg_yearly from lineitem, part where p_partkey = l_partkey and p_brand = 'Brand#23' and p_container = 'MED BOX' and l_quantity < (select 0.2 * avg(l_quantity) from lineitem where l_partkey = p_partkey);
Q19	select sum(l_extendedprice* (1 - l_discount)) as revenue from lineitem, part where (p_partkey = l_partkey and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG') and l_quantity >= 0 and l_quantity <= 10 and p_size between 1 and 5 and l_shipmode in ('AIR', 'AIR REG') and l_shipinstruct = 'DELIVER IN PERSON') or (p_partkey = l_partkey and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK') and l_quantity >= 11 and l_quantity <= 20 and p_size between 1 and 10 and l_shipmode in ('AIR', 'AIR REG') and l_shipinstruct = 'DELIVER IN PERSON') or (p_partkey = l_partkey and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG') and l_quantity >= 21 and l_quantity <= 30 and p_size between 1 and 15 and l_shipmode in ('AIR', 'AIR REG') and l_shipinstruct = 'DELIVER IN PERSON');

platforms support these programming models. For example, Jetson AGX Xavier does not support OpenCL for CPU and GPU, and A10-7850K is unable to support CUDA.

Programming Model Utilization. We develop different versions of FineQuery operators to adapt to various platforms. For the Jetson AGX Xavier platform, we use CUDA for GPU and use OpenMP to release the CPU parallel potentials. Even though the CPU on the edge device has only eight cores, it is good at handling situations that are not suitable for parallelism. Furthermore, for operations combining conditional branching and instruction jumping, OpenMP is able to achieve good performance. Following this guideline, we incorporate into the performance model the choice of using OpenMP for operators in queries. We also use an Nvidia performance profiling tool, *nvprof*, to facilitate our optimization. For sub-components in our integrated engine, we also explore standard functions available in `thrust` library for better results. For example, we replace a GPU *select* kernel for `thrust::copy_if` in Q19 of Table 1 that completes the same task, and find that the thrust function can improve performance by reducing 1.5× query time.

Adaptability. We implement the operators using various programming models, including OpenCL, OpenMP, and CUDA, for adaptability. The performance model in Section 5 is independent of the specific programming implementations of each operator. By incorporating the performance statistics from different platforms into the performance model, FineQuery can deliver the optimal query processing solution for a given situation.

7 Evaluation

In this section, we evaluate the performance of FineQuery at the edge from various perspectives.

7.1 Experimental Setup

Methodology. We measure the performance of FineQuery on the CPU-GPU integrated edge device. In FineQuery, operators are executed on both the CPU and GPU cooperatively in a fine-grained mechanism. We also measure the performance of FineQuery on only the CPU, denoted as "CPU-only", and FineQuery on only the GPU, denoted as "GPU-only", respectively. Besides, we also compare FineQuery with the query processing on the discrete architecture.

Platforms. We conduct experiments on a CPU-GPU integrated edge device, Nvidia Jetson AGX Xavier. This platform provides eight Carmel ARM CPU cores and 512 Nvidia Volta GPU cores with a unified 32GB DRAM. To

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
TPC-H Dataset

Dataset	Size	Order #	Lineitem #	Part #
A	1.5 GB	3M	11,997,996	0.4 M
B	3.6 GB	7.5 M	29,999,795	1 M
C	6 GB	12 M	45,999,898	1.4 M
D	7.3 GB	15 M	59,986,052	2 M

achieve a comprehensive understanding of FineQuery, we also conduct experiments on an integrated architecture, AMD A10-7850K, which integrates CPU and GPU on the same chip. In addition, we compare with a discrete CPU-GPU platform, equipped with an Intel i7-8700K CPU and an Nvidia GeForce GTX1080Ti GPU.

Datasets. TPC-H is a well-known benchmark for data management systems and it is widely used to evaluate the query performance of database engines [17], [18]. We evaluate FineQuery on the workloads generated by TPC-H with different scale factors for the edge platform. The details of the datasets are shown in Table 2.

Benchmarks. We use seven typical queries from TPC-H [23] for evaluation. Q1 shows the amount of billed, shipped and returned business. Q3 extracts the top ten unshipped orders. Q6 quantifies the revenue increase due to removing discounts. Q12 explores whether using less expensive shipping modes has a negative influence on critical-priority orders. Q14 calculates the revenue percentage from promotional activities. Q17 calculates the average annual revenue lost due to the no longer filled orders. Q19 retrieves the accumulative avenue by items transmitted on air and delivered in person. These seven queries cover most of the operators. The details of the seven queries are shown in Table 1.

7.2 Performance

We report the performance of latency and bandwidth utilization of FineQuery in this part.

Latency. We show the latency comparison result in Fig. 11. Fig. 11a reports the latency result on the Jetson platform, while Fig. 11b reports the latency result on the A10-7850K platform. In this work, latency is defined as the end-to-end duration from the time a query starts to the time it ends. In general, we have the following observations.

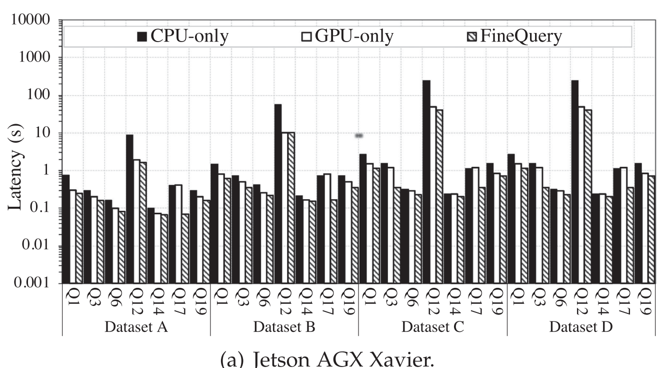
First, FineQuery can achieve performance benefits in all cases. In general, FineQuery can reduce the latency by 42.81% on average. Compared with "CPU-only", FineQuery reduces the latency by 56.13%. Compared with "GPU-only", FineQuery reduces the latency by 29.48% on average. Take Q17 for illustration; the query mainly contains three selection operators and a join operator. With FineQuery, the three selection operators take about 50ms, and the join operator takes 35ms on Dataset A. Without FineQuery, they could take up to 381ms and 238ms on Dataset A, respectively.

Second, we find that FineQuery exhibits various performance behavior in different queries, which relates to the operator-device preferences. We take Q14 and Q17 for comparison. For Q14, FineQuery can reduce the latency by 52.78% over CPU-only, while only 22.24% over GPU-only. However, as to Q17, it shows an opposite performance trend.

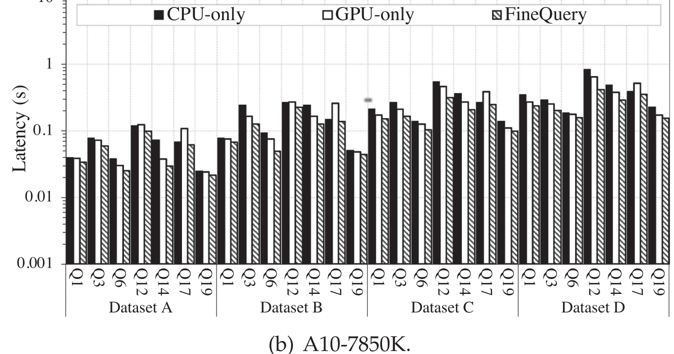
Third, when we compare the performance of the seven queries under different datasets, we find that the processing latency of the SQL queries increases with the size of the datasets. The size of dataset B is 2.51× over dataset A, and the average latency for dataset B is about 2.40× longer over that for dataset A. The size of dataset D is 5.06× over dataset A, and the average latency for dataset D is 5.95× longer over that of dataset A.

Fourth, when we compare the two CPU-GPU integrated platforms of Xavier and A10-7850K, we find that the A10-7850K can provide much lower latency. The reason is that both the CPU and the GPU on A10-7850K are much powerful than those of Xavier. Accordingly, we can conclude that when the edge device integrates more powerful processors, the system's query processing capacity also increases.

Bandwidth Utilization. The comparison result of bandwidth utilization is shown in Fig. 12. In this work, bandwidth utilization is defined as the size of used data divided by the processing time. We have the following observations. First, FineQuery has the highest bandwidth utilization among all the methods. The average bandwidth utilization of FineQuery is 3.26 GB/s, which is 2.95× over the GPU-only method and 1.82× over the CPU-only method. Second, we find that FineQuery shows different bandwidth utilization among these queries. For example, Q3 and Q6 exhibit high bandwidth utilization, while Q12 and Q17 exhibit relatively low bandwidth utilization. Third, from Fig. 12, we can observe how the bandwidth utilization changes with the size of the data. As the amount of data increases, bandwidth utilization gradually decreases in all cases, but the decline rate gradually slows down. Besides, the A10-7850K integrated platform can provide more bandwidth.



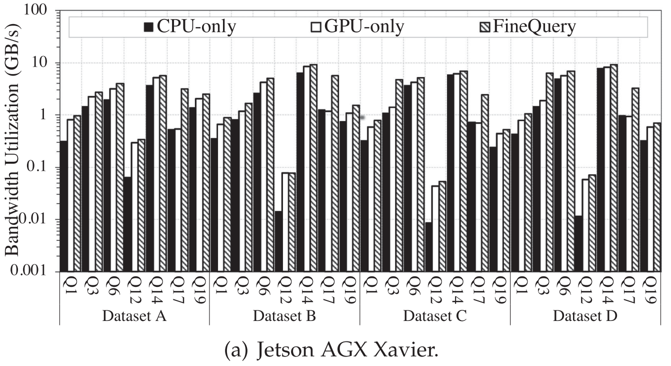
(a) Jetson AGX Xavier.



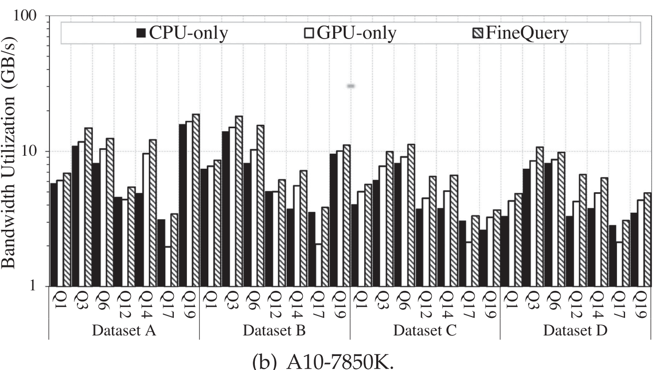
(b) A10-7850K.

Fig. 11. Latency evaluation on different datasets.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.



(a) Jetson AGX Xavier.



(b) A10-7850K.

Fig. 12. Bandwidth utilization on different datasets.

7.3 Optimization Analysis

The CPU-GPU integrated edge device features a hierarchical memory architecture. We investigate the speedups achieved by our optimization by comparing the execution time with different strategies on the edge device Jetson AGX Xavier. We show the comparison results of the query time of *select* in Q19 in Table 3. Other operators have similar performance behaviors. The "CPU version" column records the query processing time of the CPU version. The column of "FineQuery (w/o optimization)" represents the query time without the optimization in Section 6, while the column of "FineQuery" represents the query time with full optimization. The optimized FineQuery outperforms the CPU version and the FineQuery without optimization by 3.0× and 2.1×, respectively. The reason is that when shared memory is used effectively, there are fewer thread collisions. Furthermore, the shared memory, which is the controllable GPU cache, also improves the efficiency of memory access.

7.4 Comparison With Discrete GPU

In this part, we further compare FineQuery on the Jetson integrated edge device with the query processing on the discrete GPU. Because FineQuery also targets edge environments, cost-effectiveness and energy-efficiency are important performance metrics. We evaluate all seven queries on two platforms with the largest dataset for illustration.

Latency. The latency result is shown in Fig. 13. FineQuery on the Jetson platform can reduce the latency by 56.69% compared to the query processing on the discrete architecture. Overall, finequery on the Jetson platform can reduce the latency by For the majority of datasets, FineQuery on the Jetson platform can save Among them, Q6 has the best improvement, which reduces the latency by 71.69%.

Energy Efficiency. Thermal design power (TDP) is an indicator used to describe energy consumption. The TDP of the Jetson platform is only 30 W, while the TDP of the 1080Ti platform is 250 W. In this work, We define energy efficiency as the amount of data that can be processed per watt. The result is shown in Fig. 14a. FineQuery's energy efficiency is 21× over that of discrete GPU on average.

Cost Effectiveness. We also compare the cost effectiveness between FineQuery on integrated architecture and query processing on discrete architecture. In this work, we use the amount of data that can be processed per US dollar to define the cost effectiveness. The price of the Jetsson platform is $699 and the price of 1080Ti is $1100. The result is shown in Fig. 14b. FineQuery's performance-per-cost ratio is 4× over that of the discrete GPU on average.

TABLE 3
Analysis of Query Time for *Select* in Q19

Dataset	CPU version	FineQuery (w/o optimization)	FineQuery
A	33 ms	27 ms	10 ms
B	52 ms	34 ms	24 ms
C	156 ms	55 ms	38 ms
D	162 ms	103 ms	46 ms

7.5 Summary of Technical Contributions

FineQuery exhibits clear advantages in latency, bandwidth utilization, energy efficiency, and cost-effectiveness at the edge, which can be applied to a wide range of application scenarios.

First, our carefully designed model can leverage the advantage of the integrated design of the edge device to maximize its ability to process queries. The design of FineQuery can shed light on the SQL-based query processing in other CPU-accelerator integrated heterogeneous environments. Moreover, when it comes to distributed databases built on edge devices, FineQuery can still work by adding a data transmission model between nodes. In the new performance model, a data transfer network module can be added to handle communication between edge devices.

Second, we use TPC-H, a hallmark industry benchmark, in evaluation. The experimental result implies that FineQuery can obtain significant performance in complicated situations. It is especially impressive given the low power consumption required for the edge device, a manifestation of its high cost-efficiency and energy-efficiency. As a result, for today's query workloads at the edge, our method can

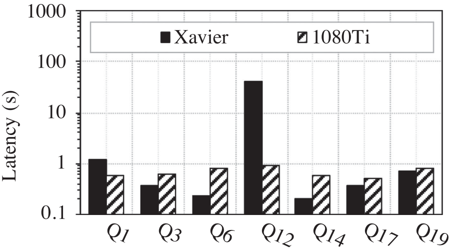
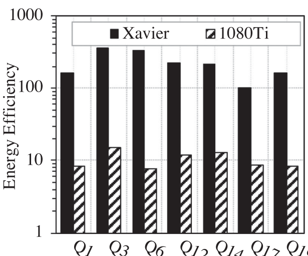
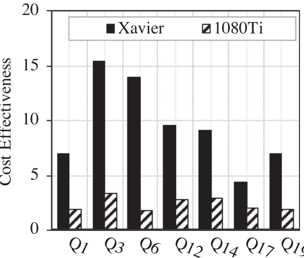


Fig. 13. Energy efficiency comparison of FineQuery on integrated architecture and query processing on discrete architecture.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.



(a) Energy efficiency analysis.



(b) Cost effectiveness analysis.

Fig. 14. Analysis of energy efficiency and cost effectiveness.

significantly reduce processing latency and improve user response experience.

Third, we implement both the CUDA and OpenCL versions of FineQuery, so FineQuery can adjust to a wide range of HPC platforms. This effectively demonstrates its portability and the breadth of the application scope of FineQuery.

8 Related Work

Query processing [10], [11], [12], [13], [24], [25], [26], [27], [28], [29], and heterogeneous systems [3], [4], [20], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44] are hot research topics in recent years. The closest works to ours are FineStream [3], [4] and FinePar [33], [34]. FineStream [3] is a fine-grained window-based stream processing engine, and is built on CPU-GPU integrated architectures. FineStream supports queries on stream data, and processes stream data at batch granularity. It distributes the operators of a query to different devices via a performance model. Similarly, both FineStream and FineQuery belong to the continuous operator model, in contrast to the bulk-synchronous parallel model [19], [45]. However, FineStream and FineQuery differ in three major aspects. First, their application scenarios are different. FineStream focuses on querying stream data, but FineQuery focuses on the processing of static data, whose queries are more complicated. Second, their models are different. For example, FineQuery does not have the concepts of widow and batch as in FineStream. Third, FineQuery also targets edge devices, where cost-effectiveness and energy-efficiency are important. As to FinePar [34], it is a fine-grained workload partitioning framework targeting irregular applications, such as SpMV and BFS. FinePar is also built on CPU-GPU integrated architectures. Because GPU is sensitive to irregular workload, we distribute the regular part of the workload to GPU while remain the irregular part to CPU to achieve high performance. Different from FinePar, FineQuery focuses on data processing in database domain and co-runs different operators of kernels on different devices, instead of co-running the same kernel.

Fine-Grained Data Processing. As CPU-GPU integrated architectures have been proved to be of great use to various domains, many works have been conducted on fine-grained data processing on CPU-GPU integrated architectures. Zhang *et al.* [46] studied the co-running performance on CPU-GPU integrated architectures, and then developed a co-running benchmark and a performance prediction model [22]. Moreover, fine-grained stream data processing [3], [4] and irregular workload partitioning [33], [34] have also been developed. Popular algorithms can also benefit from the CPU-GPU integrated architectures. For example, Daga *et al.* [47] developed a hybrid BFS algorithm, where top-down traversal is conducted on CPU and bottom-up traversal is conducted on GPU. Zhang *et al.* [48] further developed a performance model to describe the hybrid BFS algorithm on the integrated architectures. Since machine learning becomes a hot research topic in recent years, Zhang *et al.* [49] developed a machine learning benchmark, called iMLBench, for studying the performance of machine learning workloads on the integrated architectures.

Heterogeneous Query Processing. CPU-GPU integrated architectures can accelerate database applications, and many researchers apply integrated architectures to query processing. Hetherington *et al.* [50] characterized the performance of key-value store applications on integrated architectures, and exhibited the benefits of integrated architectures over discrete architectures. He *et al.* [51] proposed an in-cache query co-processing method, which utilizes CPU to assist GPU to cache data, targeting integrated architectures with shared last level cache. He *et al.* [2] developed a novel hash join algorithm with CPU and GPU co-running in different steps on integrated architectures. Zhang *et al.* [52] further proposed DIDO, which is a dynamic pipeline for in-memory key-value stores on integrated architectures. Daga *et al.* [53] developed a parallel B+ tree algorithm on integrated architectures. Wang *et al.* [54] proposed a CPU-FPGA heterogeneous system for handling distance-related algorithms (e.g., K-means and KNN). Chen *et al.* [55] developed a MapReduce framework, utilizing both CPU and GPU on integrated architectures.

CPU-GPU Integrated Edge Computing. There is a large amount of work on integrated edge computing in recent years. For example, Ukidave *et al.* [56] evaluated the CPU-GPU integrated edge device, Nvidia Jetson TK1, in HPC domain. Lee *et al.* [57] applied the Jetson TX1 edge device for car plate recognition. Rungsuptaweekoon *et al.* [58] evaluated the power efficiency of machine learning inferences on Jetson TX2. Amert *et al.* [59] studied the scheduling problem on Nvidia TX2. Davidson *et al.* [60] accelerated the error resilient image processing application on the Jetson TX1 edge device. Mittal [61] surveyed the deep learning models built on Nvidia Jetson platforms. Jose *et al.* [62] developed FaceNet and MTCNN on Jetson TX2.

9 Conclusion

Integrated architectures have shown great promise in edge computing. Since the CPU and GPU share the same unified memory, data transmission via PCIe has been eliminated, and CPU and GPU can perform fine-grained cooperation. We apply CPU-GPU integrated edge device to fine-grained query processing and exhibit the benefits of utilizing CPU-GPU co-running as well as zero-copy optimization. Experiments show that FineQuery achieves 42.81% performance benefits over the current methods.

References

[1] JETSON XAVIER NX, 2020. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

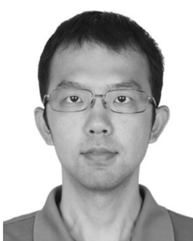
Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

[2] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled CPU-GPU architecture," *Proc. VLDB Endowment*, vol. 6, no. 10, pp. 889–900, 2013.
[3] F. Zhang *et al.*, "FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 633–647.
[4] F. Zhang *et al.*, "Fine-grained multi-query stream processing on integrated architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2303–2320, Sep. 2021.
[5] D. Wang, F. Zhang, W. Wan, H. Li, and X. Du, "FineQuery: Fine-grained query processing on CPU-GPU integrated architectures," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2021, pp. 355–365.
[6] K. Jaiswal, S. Sobhanayak, A. K. Turuk, S. L. Bibhudatta, B. K. Mohanta, and D. Jena, "An IoT-cloud based smart healthcare monitoring system using container based virtual environment in edge device," in *Proc. Int. Conf. Emerg. Trends Innov. Eng. Technol. Res.*, 2018, pp. 1–7.
[7] Y. Yang, Q. Cao, and H. Jiang, "EdgeDB: An efficient time-series database for edge computing," *IEEE Access*, vol. 7, pp. 142 295–142 307, 2019.
[8] J. Paparrizos *et al.*, "VergeDB: A database for IoT analytics on edge devices," in *Proc. Conf. Innov. Data Syst. Res.*, 2021, pp. 1–8.
[9] Jetson AGX Xavier Developer Kit, 2020. [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
[10] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit.*, 2009, pp. 79–90.
[11] B. Chandramouli *et al.*, "Trill: A high-performance incremental query processor for diverse analytics," *Proc. VLDB Endowment*, vol. 8, pp. 401–412, 2014.
[12] S. Breß and G. Saake, "Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS," *Proc. VLDB Endowment*, vol. 6, pp. 1398–1403, 2013.
[13] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. 3rd Workshop Gen.-Purpose Comput. Graph. Process. Units*, 2010, pp. 94–103.
[14] R. Fang *et al.*, "GPUQP: Query co-processing using graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 1061–1063.
[15] C. Root and T. Mostak, "MapD: A GPU-powered big data analytics and visualization platform," in *Proc. ACM SIGGRAPH Talks*, 2016, pp. 1–2.
[16] S. Zhang *et al.*, "OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures," *Proc. VLDB Endowment*, vol. 6, no. 12, pp. 1374–1377, 2013.
[17] J. Paul, J. He, and B. He, "GPL: A GPU-based pipelined query processing engine," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1935–1950.
[18] P. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*, 2013, pp. 61–76.
[19] S. Zhang *et al.*, "Hardware-conscious stream processing: A survey," *ACM SIGMOD Rec.*, vol. 48, no. 4, pp. 18–29, 2020.
[20] A. Koliousis *et al.*, "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 555–569.
[21] D. Guide, "CUDA C programming guide," NVIDIA, Santa Clara, CA, USA, Jul. 2013.
[22] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 905–918, Mar. 2017.
[23] TPC-H Vesion 2 and Version 3, 2022. [Online]. Available: https://www.tpc.org/tpch/
[24] W. Xia, C. Wei, Z. Li, X. Wang, and X. Zou, "NetSync: A network adaptive and deduplication-inspired delta synchronization approach for cloud storage services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2554–2570, Oct. 2022.
[25] W. Xia *et al.*, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2016, pp. 101–114.
[26] F. Zhang *et al.*, "TADOC: Text analytics directly on compression," *VLDB J.*, vol. 30, pp. 163–188, 2021.
[27] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.
[28] F. Zhang *et al.*, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proc. VLDB Endowment*, vol. 11, pp. 1522–1535, 2018.
[29] F. Zhang *et al.*, "CompressDB: Enabling efficient compressed data direct processing for various databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2022.
[30] Z. Tang and Y. Won, "Multithread content based file chunking system in CPU-GPGPU heterogeneous architecture," in *Proc. 1st Int. Conf. Data Compression Commun. Process.*, 2011, pp. 58–64.
[31] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.
[32] W. Xia *et al.*, "The design of fast content-defined chunking for data deduplication based storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2017–2031, Sep. 2020.
[33] F. Zhang, J. Zhai, B. Wu, B. He, W. Chen, and X. Du, "Automatic irregularity-aware fine-grained workload partitioning on integrated architectures," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 3, pp. 867–881, Mar. 2021.
[34] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2017, pp. 27–38.
[35] F. Zhang *et al.*, "G-TADOC: Enabling efficient GPU-based text analytics without decompression," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 1679–1690.
[36] S. Tang, B. He, S. Zhang, and Z. Niu, "Elastic multi-resource fairness: Balancing fairness and efficiency in coupled CPU-GPU architectures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 875–886.
[37] S. Tang, B. He, C. Yu, Y. Li, and K. Li, "A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 71–91, Jan. 2022.
[38] S. Tang, C. Yu, and Y. Li, "Fairness-efficiency scheduling for cloud computing with soft fairness guarantees," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2020.3021084.
[39] B. Feng *et al.*, "APNN-TC: Accelerating arbitrary precision neural networks on ampere GPU tensor cores," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–13.
[40] B. Feng *et al.*, "Palleon: A runtime system for efficient video processing toward dynamic class skew," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 427–441.
[41] Y. Wang *et al.*, "GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 515–531.
[42] Y. Wang, B. Feng, and Y. Ding, "QGTC: Accelerating quantized graph neural networks via GPU tensor core," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 107–119.
[43] F. Zhang, Z. Chen, C. Zhang, A. C. Zhou, J. Zhai, and X. Du, "An efficient parallel secure machine learning framework on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2262–2276, Sep. 2021.
[44] Z. Pan *et al.*, "Exploring data analytics without decompression on embedded GPU systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1553–1568, Jul. 2022.
[45] S. Venkataraman *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 374–389.
[46] F. Zhang, J. Zhai, W. Chen, B. He, and S. Zhang, "To co-run, or not to co-run: A performance study on integrated architectures," in *Proc. IEEE 23rd Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2015, pp. 89–92.
[47] M. Daga, M. Nutter, and M. Meswani, "Efficient breadth-first search on a heterogeneous processor," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 373–382.
[48] F. Zhang *et al.*, "An adaptive breadth-first search algorithm on integrated architectures," *J. Supercomput.*, vol. 74, no. 11, pp. 6135–6155, 2018.
[49] C. Zhang, F. Zhang, X. Guo, B. He, X. Zhang, and X. Du, "iMLBench: A machine learning benchmark suite for CPU-GPU integrated architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1740–1752, Jul. 2021.
[50] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2012, pp. 88–98.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.

[51] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 329–340, 2014.
[52] K. Zhang, J. Hu, B. He, and B. Hua, "DIDO: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 671–682.
[53] M. Daga and M. Nutter, "Exploiting coarse-grained parallelism in B+ tree searches on an APU," in *Proc. SC Companion: High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 240–247.
[54] Y. Wang, B. Feng, G. Li, L. Deng, Y. Xie, and Y. Ding, "STPAcc: Structural TI-based pruning for accelerating distance-related algorithms on CPU-FPGA platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 5, pp. 1358–1370, May 2022.
[55] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.
[56] Y. Ukidave, D. Kaeli, U. Gupta, and K. Keville, "Performance of the NVIDIA Jetson TK1 in HPC," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 533–534.
[57] S. Lee, K. Son, H. Kim, and J. Park, "Car plate recognition based on CNN using embedded system with GPU," in *Proc. 10th Int. Conf. Hum. Syst. Interact.*, 2017, pp. 239–241.
[58] K. Rungsuptaweekoon, V. Visoottiviseth, and R. Takano, "Evaluating the power efficiency of deep learning inference on embedded GPU systems," in *Proc. 2nd Int. Conf. Inf. Technol.*, 2017, pp. 1–5.
[59] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 104–115.
[60] R. L. Davidson and C. P. Bridges, "Error resilient GPU accelerated image processing for space applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1990–2003, Sep. 2018.
[61] S. Mittal, "A survey on optimized implementation of deep learning models on the NVIDIA Jetson platform," *J. Syst. Archit.*, vol. 97, pp. 428–442, 2019.
[62] E. Jose, M. Greeshma, M. T. P. Haridas, and M. H. Supriya, "Face recognition based surveillance system using FaceNet and MTCNN on Jetson TX2," in *Proc. 5th Int. Conf. Adv. Comput. Commun. Syst.*, 2019, pp. 608–613.

Jiesong Liu is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined MOE in 2020. His major research interests include database systems, and parallel and distributed systems.



Feng Zhang received the bachelor's degree from Xidian University, in 2012, and the PhD degree in computer science from Tsinghua University, in 2017. He is an associate professor with DEKE Lab and the School of Information, Renmin University of China. His major research interests include database systems, and parallel and distributed systems.

Hourun Li is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2020. His major research interests include database systems, and parallel and distributed systems.

Dalin Wang received the bachelor's degree from the Renmin University of China, in 2020. He is currently working toward the master degree with the Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2019. His major research interests include database systems, and parallel and distributed systems.

Weitao Wan received the bachelor's degree from the Renmin University of China, in 2022, and is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined MOE in 2019. His major research interests include database systems and distributed systems.

Xiaokun Fang received the bachelor's degree from the Renmin University of China, in 2022. He is currently working toward the master degree with the Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2020. His major research interests include parallel and distributed systems and machine learning.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China, in 2003, and the PhD degree in computer science from Tsinghua University, in 2010. He is an associate professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis, and modeling of parallel applications.

Xiaoyong Du received the BS degree from Hangzhou University, Zhejiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the 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.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Renmin University. Downloaded on November 03,2022 at 07:09:15 UTC from IEEE Xplore. Restrictions apply.