Towards A GPU-Accelerated Stream Processing Engine Through Query Compilation

Florian Schmeller[1], Dwi P. A. Nugroho[2], Steffen Zeuch[2] and Tilmann Rabl[1]

[1]*Hasso-Plattner-Institut für Digital Engineering gGmbH, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany*
[2]*Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany*

Abstract

Over the last decade, data stream processing has emerged to provide real-time insights into large, unbounded volumes of data. At the same time, graphics processing units (GPU) have become an important accelerator for improving the performance of compute-bound applications. Nevertheless, state-of-the-art data streaming systems opt to scale-out and typically do not make efficient use of the underlying hardware. Recent work has shown that query compilation is a viable technique to support hardware advancements in query processing engines. However, it often comes with high development and maintenance costs. In particular, when the process involves hardware accelerators such as GPUs. In this paper, we propose a framework for compiling data stream queries to efficient GPU code in a developer-friendly manner. We demonstrate the feasibility of our framework by integrating it into the data management system NebulaStream. Our experiments show that frequent memory transfers between CPU and GPU impact the query processing throughput.

Keywords

Query compilation, Data stream processing, GPU Computing

1. Introduction

Large-scale data processing systems have become more heterogeneous with regard to their components in order to meet the increasing computational demands over the last decade. General-purpose graphics processing units are an important example of modern hardware devices that enable better performance for specific computations. Consequently, utilizing GPUs for query processing in heterogeneous computer systems has been a popular research topic in recent years [1, 2, 3, 4]. The adoption of query compilation in data processing systems has seen a huge impact since the strategy was popularized by Neumann [5] with the produce/consume model. With the increasing heterogeneity in computer systems following the end of Dennard scaling [6], query compilation is a well-positioned technique to capitalize on technological advancements in computer architecture and compiler technology. As a result, compiling queries to specialized hardware can unlock significant performance benefits in data stream processing by achieving high hardware utilization [7].

The database research community has investigated query compilation for heterogeneous processors over the last decade extensively [8, 9, 10, 3]. However, existing studies often disre-

LWDA'24: Lernen, Wissen, Daten, Analysen. September 23–25, 2024, Würzburg, Germany
florian.schmeller@hpi.de (F. Schmeller); d.nugroho@tu-berlin.de (D. P. A. Nugroho); steffen.zeuch@tu-berlin.de (S. Zeuch); tilmann.rabl@hpi.de (T. Rabl)
© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

gard maintainability in favor of processor-specific optimizations to improve query processing performance. In response, Grulich et al. propose the Nautilus [11] framework to unify query interpretation and query compilation by utilizing a trace-based, just-in-time compilation strategy. Thus, they provide better introspection capabilities during the software development process of query operators. In this work, we extend the idea behind Nautilus to the field of heterogeneous computing and bring trace-based query compilation to the GPU. To this end, we present a framework that enables developers to implement query operators for the GPU on a high abstraction level to support maintainable software development without re-implementing the underlying execution model of the query engine.

In particular, our work makes the following contributions:

- We design a framework that enables a seamless operator interface switch from tuple-at-a-time execution to batched execution, which is beneficial to GPU-based computing, to support the compilation of a GPU kernel into a query plan for trace-based query compilation.
- We integrate our framework into the data stream management system NebulaStream[1] [12] and demonstrate the feasibility of our framework by implementing three commonly used data stream operators.
- We show that low-bandwidth interconnects between CPU and GPU represent a performance bottleneck in throughput-oriented systems such as data streaming systems.

We organize the paper as follows. First, we introduce background information about general-purpose computing on the GPU and trace-based query compilation in Section 2. In Section 3, we present our framework for compiling data streaming queries to GPU code and the integration of our framework in NebulaStream. In Section 4, we showcase the experimental evaluation of our framework in an end-to-end benchmark using NebulaStream on state-of-the-art streaming workloads. In Section 5, we discuss related work. Lastly, we conclude our work in Section 6.

2. Background

In this section, we introduce the concept of general-purpose computing on the GPU. Furthermore, we outline the Nautilus [11] query compilation framework, which aims to unify query interpretation and query compilation.

2.1. General-purpose Computing on the GPU

Since Dennard scaling came to an end, computer systems have become increasingly heterogeneous [6]. The responsibility for computing a particular functionality has shifted from the central processing unit (CPU) to application-specific hardware. The GPU has emerged as a versatile co-processor for computer systems over the last two decades. Conceptually, we use a GPU to accelerate a compute-bound application by parallelizing the execution over a large number of threads. This has also sparked interest in the database community to accelerate

[1] Find NebulaStream and related information on https://nebula.stream.

query processing [9, 8, 10, 4]. Rosenfeld et al. [1] provide an in-depth survey on the use of GPUs in query processing.

The functionality of GPUs has grown to accommodate the execution of compiled, fully-programmable functions, i.e., *compute kernels*. The *compute kernel* is a function that specifies data-parallel computations on individual elements of a data collection [9]. Throughput-oriented co-processors employ the *kernel programming model* to develop and program compute kernels.

When we frequently transfer memory between the CPU and the GPU, a low-bandwidth interconnect between the two components is a performance bottleneck in GPU computing. To mitigate high data transfer times, we can employ different memory allocation techniques and memory bandwidth expansion using, e.g., NVLink [13]. *Page-locked*, or *pinned*, memory is an allocated chunk of memory, where the memory pages are pinned to their physical location. Here, the GPU leverages direct memory access (DMA) to transfer data to and from the CPU without host intervention because the physical location of page-locked memory allocation is known beforehand (in contrast to a *pageable* memory allocation). However, the size of page-locked memory is limited because the operating system cannot employ memory paging. Furthermore, excessive allocation of page-locked memory contributes to overloading the operating system's virtual memory management, which negatively impacts system performance [14].

2.2. Trace-based Query Compilation

In practice, the main engineering challenge for a compilation-based query engine is to balance performance and developer productivity [11]. The Nautilus framework by Grulich et al. [11] combines query compilation with a novel approach to code generation to achieve a developer-friendly programming experience for compilation-based query processing systems. Thereby, Nautilus allows developers to debug their code in an interpreted way, removing the indirection of running and inspecting the state of generated code. Nautilus employs a push-based query compilation strategy [5] that uses multiple phases for transforming a logical query plan to an executable query plan in the form of a binary shared object. The Nautilus framework is the foundation for the query compilation engine of NebulaStream.

A Nautilus operator is a query operator that implements a programming interface for push-based query processing. When compiling a Nautilus query plan, Nautilus creates a program trace from each Nautilus operator. During query compilation, a tracing module creates a comprehensive recollection, i.e., the program trace, of the recorded operations in a symbolic execution of the Nautilus query plan. With the program trace, we have a high abstraction level of the Nautilus operators that make up the query plan. The framework decouples the query operator implementation from target-specific code generation by the means of the Nautilus intermediate representation (Nautilus IR). After trace generation, the selected code generation back-end receives a control flow graph comprised of Nautilus IR instructions and generates the program code in the target language. Nautilus does not have a compilation back-end for GPUs.

3. Compilation-based Stream Processing on the GPU

In this section, we propose a framework for compiling data stream queries to GPU code. We present our solution for compiling data stream pipelines to GPU code using the trace-based

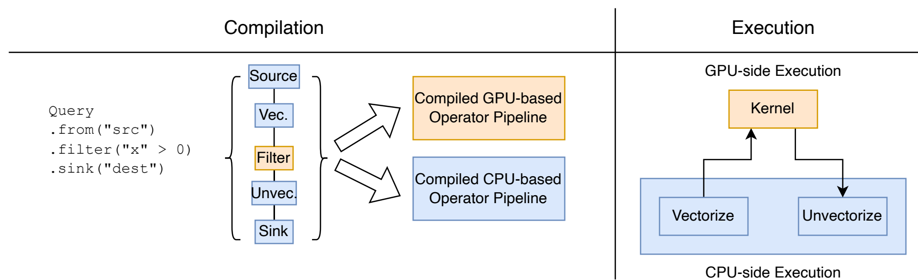


Figure 1: Our proposed framework.

Nautilus framework. We apply our framework to build three commonly used data stream operators: map, selection, and window aggregation. We show how to integrate our framework into the data stream management system NebulaStream [12].

3.1. Compiling Data Stream Pipelines to GPU Code

With our framework, we want to give a software developer the ability to program query operators with methods from the GPU programming model. At the same time, we intend to strike a fine balance between the high-performance benefits of GPU programming and high-level query code abstractions. To this end, we will use the Nautilus framework as a basis and develop an extension to generate query code for the GPU.

Internally, the query engine of NebulaStream works on tuple buffers but uses a Nautilus-style tuple-at-a-time interface to expose single tuples. Thus, we need to introduce a new operator programming interface that enables us to specialize the query execution for the GPU. We define a new class of *vectorizable* operators in Nautilus, which work on a tuple buffer instead of a single tuple. Furthermore, we introduce a new *Vectorize* operator to materialize in-flight tuples and pass them as a buffer to a vectorizable operator. To allow the reversal to a tuple-at-a-time operator interface, we also introduce the *Unvectorize* operator to unroll a tuple buffer into single tuples for downstream operators.

Figure 1 shows our proposed framework and its application. In the first part, we produce compiled CPU-based as well as GPU-based operator pipelines from a transformed query plan. In the second part, we execute our compiled pipelines as part of the query engine.

As a result, we can now traverse a query plan and identify an operator *Op* that is suitable for offloading to the GPU. Then, we transform the query plan around it using Vectorize and Unvectorize operators to support a batched GPU-based execution model, which we define as a *Stage* operator. To offload the processing of a query plan to the GPU, a software developer only needs to implement the vectorizable interface for desired operators while our framework deals with the hybrid execution. We can apply our approach to any GPU programming framework because we designed it to align with the high abstraction level of Nautilus rather than tailoring it to a specific GPU programming framework.

Figure 2 depicts the query plan transformation to the Stage operator using Vectorize and Unvectorize operators. Given a physical query plan, we select a physical operator *Op* and

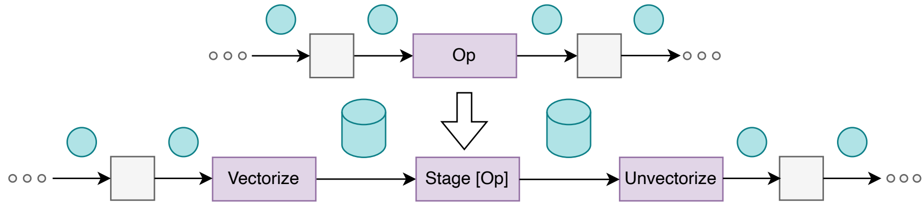


Figure 2: The Stage transformation changes the operator interface from tuple-at-a-time to buffer-at-a-time.

transform it into a physical Stage operator. Furthermore, we add physical Vectorize and Unvectorize operators to facilitate the change of operator interface from tuple-at-a-time to buffer-at-a-time.

To generate GPU code, we add a compilation back-end for CUDA C++ to Nautilus based on the C++ back-end. We introduce the *Kernel* operator to represent the execution of a kernel function on the GPU in a query plan. After we have transformed the query plan using Vectorize and Unvectorize operators, we traverse the query plan again to identify Stage operators and transform them into Kernel operators. During query compilation, we create a Nautilus trace of the operator *Op* in a symbolic execution of *Op*. Then, we convert the trace to Nautilus IR, which we have extended to account for the GPU programming model. Finally, we convert the IR to CUDA C++ source code and compile it to a binary shared object using a CUDA-compatible compiler.

3.2. Building Data Stream Operators for the GPU

With our framework, we can compile any suitable operator to GPU code and selectively offload parts of the query plan to the GPU. To demonstrate the feasibility of our framework, we have implemented three commonly used data stream operators (map, selection, and aggregation) in the data management system NebulaStream using our framework. We encapsulate the execution of the compiled kernel function in a *kernel wrapper* function, which is responsible for GPU-side state management and kernel invocation.

Due to its stateless nature, we describe the implementation of the map operator first. We assume that the input schema of a map operator is equal to its output schema, i.e., the map operator applies a function with the same input and output arity to a tuple. To support a map operator with different input and output arities, we need to size the GPU-side input and output buffers accordingly to ensure correct memory accesses. We use our extension of Nautilus IR instructions to symbolically calculate the thread index in the thread hierarchy according to the GPU programming model.

Listing 1 shows the difference between the original map operator (left) and our newly built, vectorizable map operator (right), respectively. Here, our framework wraps the Stage operator around the vectorizable map operator to generate a trace for the kernel compilation. The implementation of the map operator in our framework exhibits high structural similarity to a possible kernel code implementation while providing a high abstraction level.

Data stream operators such as aggregations and joins require some form of state management.

```cpp
void Map::execute(Context& ctx,
                  Record& record) {
    mapExpression->execute(record);
    if (hasChild()) {
        child->execute(ctx, record);
    }
}
```

```cpp
void VMap::execute(Context& ctx,
                   Buffer& buffer) {
    auto tid = OneDimThreadIdx();
    auto address = buffer.getBuffer();
    auto numRecords = buffer.getNumRecords();
    if (tid < numRecords) {
        auto record = read(address, tid);
        mapExpression->execute(record);
        write(tid, address, record);
    }
    if (hasChild()) {
        child->execute(ctx, buffer);
    }
}
```

Listing 1: The map operator implemented in the regular Nautilus-style operator interface (left) and the new vectorizable Nautilus-style operator interface (right).

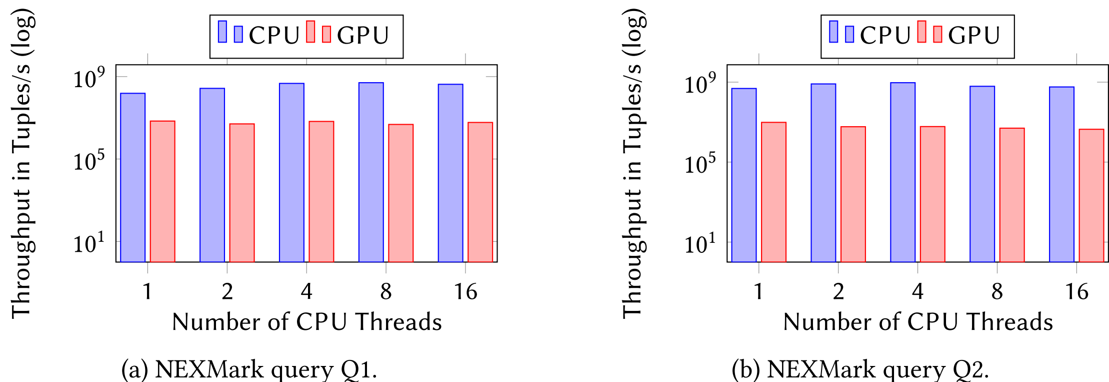
To combine our framework with state management, we have to convert between CPU-side state and GPU-side state and vice versa. As a proof of concept, we have opted for the selection operator to produce a bit vector filter. Therefore, we make the final selection on the CPU-side tuple buffer with the obtained bit vector converted from the GPU-side tuple buffer.

In NebulaStream, the aggregation pipeline is based on the stream slicing technique [15] and has two parts: pre-aggregation and slice merging. For any incoming tuple, the pre-aggregation operator retrieves the slice covering the timestamp of the tuple, updates partial aggregates and writes them to the slice store. The slice-merging operator creates a final aggregation tuple from the slices. We identify the pre-aggregation operator as a suitable candidate for a vectorizable implementation. Using our framework we only have to replace the pre-aggregation operator with a vectorizable pre-aggregation operator, leaving the other parts of the aggregation pipeline untouched.

We target non-keyed window-based aggregations because we only consider slices to store numerical types. We assume ingestion time as our notion of time such that all tuples in a tuple buffer share the same timestamp when the tuple buffer content was first registered in the system. Thus, all of the partial aggregates map to only one slice in the slice store. Furthermore, we have to create a GPU-side representation of the slice store and know how to convert it back on the CPU side. Since we know that the timestamp is the same across all tuples of a tuple buffer, we utilize parallel tree reduction to compute the partial aggregates for the entire tuple buffer in one kernel invocation. We only consider distributive aggregation functions because parallel reduction requires the binary reduction operator to be associative. To support keyed window-based aggregations, we can store GPU-based hash maps in a slice to identify matching keys and update partial aggregates accordingly. Having access to an abstraction of a GPU-based hash map can also help us implement streaming hash joins using our framework.

4. Evaluation

This section presents an experimental evaluation of our framework. First, we define the experimental setup. Next, we evaluate the end-to-end query processing throughput of our



(a) NEXMark query Q1.

(b) NEXMark query Q2.

Figure 3: End-to-end query processing throughput for NEXMark queries Q1 (left) and Q2 (right). The stage buffer size is set to 64 MB.

framework in the data stream management system NebulaStream. We conclude this section with a discussion of our findings.

4.1. Experimental Setup

In an end-to-end setting, we subject the NebulaStream platform to various workloads of the research-standard NEXMark benchmark suite [16]. NEXMark by Tucker et al. is a benchmark suite of data stream processing workloads modeling an online auction system. As a result, we obtain a realistic impression of the performance of the NebulaStream platform for state-of-the-art data streaming workloads when using our framework.

We conduct our experiment on a machine with 2x Intel Xeon Gold 5115 @ 2.40 GHz (10 cores per socket, with Intel Hyper-Threading Technology), 192 GB of system memory and a NVIDIA Tesla V100 GPU with 16 GB of memory. As for our operating system, we execute the experiment on Ubuntu 22.04 LTS using NebulaStream v0.5 and the CUDA Toolkit 11.7. We compile the NebulaStream project itself as well as the CPU-based queries with the `clang++` 16.0.1 compiler on the highest optimization level (i.e., `-O3`). Similarly, we compile GPU-based queries targeting compute capability 7.0 using the same compiler configuration.

4.2. End-to-End Query Processing Throughput

To quantify the performance of our framework, we select end-to-end query processing throughput and memory transfer latency as our two evaluation metrics. To this end, we execute NEXMark queries Q1 and Q2 on the NebulaStream data management system. We calculate the end-to-end query processing throughput as the number of tuples processed per second. For two distinct benchmark configurations, a higher throughput value indicates a better processing performance.

For comparison, we measure the query processing throughput for both CPU-based and GPU-based execution configurations. Furthermore, we show the latency induced by transferring memory between the CPU and the GPU in the GPU-based execution. In CPU-GPU applications,

Table 1
Benchmark for NEXMark query Q1.

	Stage Buffer Size in MB									
	Pageable Memory					*Pinned Memory*				
	64	128	512	1024	2048	64	128	512	1024	2048
Throughput in M Tuples/s	6.9	6.9	7.0	7.4	5.9	7.5	7.6	7.5	7.1	6.1
Bandwidth in GB/s										
$BW_{H,D}$	4.6	4.7	4.6	4.7	4.3	12.3	12.3	12.4	12.4	12.4
$BW_{D,H}$	4.7	4.7	4.7	4.7	4.6	13.1	13.1	13.2	13.2	13.7
BW_{Global}	520.5	510.3	502.2	503.4	500.6	518.2	510.3	501.7	501.9	498.3
$BW_{Request}$	130.2	127.6	125.6	125.9	125.1	129.6	127.6	125.4	125.5	124.6

the interconnect often represents a performance bottleneck [13]. Therefore, measuring the time it takes to move data back and forth between CPU and GPU yields a deeper insight into the throughput values.

Figure 3 depicts the measured query processing throughput numbers for both CPU-based and GPU-based execution on the NebulaStream platform for NEXMark queries Q1 (left) and Q2 (right). We use one GPU thread per tuple and do not overlap memory transfers. As we can see, the throughput is significantly higher in the CPU-based execution than in the GPU-based execution. Since both queries do not exhibit high computational complexity, external factors contribute to the comparatively slower processing performance for the GPU-based configuration. Thus, we investigate the impact of the latency induced by memory transfers between CPU and GPU on the query processing performance. Memory transfers are a prime candidate for optimization because of the high impact of low-bandwidth CPU-GPU interconnects on the overall performance when the application moves data between the two frequently.

Table 1 shows a more detailed picture of the benchmark results for NEXMark query Q1 with regards to the memory bandwidth bottleneck. We use mean values for the query processing throughput and the calculation of the effective inter-device bandwidth. Furthermore, we give the stage buffer size in megabyte (MB), which is the size of the memory buffer for in-flight tuple materialization. We determine the GPU memory buffer capacity as the quotient of the stage buffer size (e.g., 64 MB) and the input schema size (40 B), rounded down.

For reference, the theoretical bandwidth on PCI express (PCIe) 3.0 is 14.9 GB/s [1]. We observe an effective bandwidth $BW_{H,D}$ from host to device between 4.3 GB/s and 4.7 GB/s across different stage buffer sizes for pageable memory. Furthermore, we see an effective bandwidth $BW_{D,H}$ from device to host between 4.6 GB/s and 4.7 GB/s. In contrast, we achieve effective bandwidths $BW_{H,D}$ and $BW_{D,H}$ between 12.3 GB/s up to 12.4 GB/s and between 13.1 GB/s up to 13.7 GB/s, respectively, when we use pinned memory instead. We calculated that the computation of the kernel contributes around 0.7% and 2.1% to the total runtime of one batched execution for pageable and pinned memory, respectively.

We use the profiling tool `nvprof` [17] by NVIDIA to obtain two intra-device memory throughput values BW_{Global} and $BW_{Request}$. We denote the throughput for global memory transaction as

BW_{Global} in Table 1. We use requested global throughput to measure the bandwidth utilization of global memory transactions [18]. We denote the requested global throughput as $BW_{Request}$ in Table 1. We compare global throughput and requested global throughput to assess the resourcefulness of a kernel function. For reference, the theoretical bandwidth for an NVIDIA Tesla V100 (PCIe 16GB) GPU amounts to 898 GB/s [18]. The NVIDIA Tesla V100 (PCIe 16GB) GPU features L1 and L2 caches to improve the latency of global memory transactions. We obtain minimum and maximum throughput values of 500.6 GB/s and 520.5 GB/s as well as 498.3 GB/s and 518.2 GB/s for pageable and pinned memory, respectively. The minimum and maximum throughput values are 125.1 GB/s and 130.2 GB/s as well as 124.6 GB/s and 129.6 GB/s for pageable and pinned memory, respectively. It is evident that the requested global throughput $BW_{Request}$ is significantly lower than the global throughput BW_{Global}. The reason for the noticeable difference between $BW_{Request}$ and BW_{Global} is insufficient coalescing of memory accesses [18].

4.3. Discussion

To summarize our findings, we have demonstrated that our framework generates efficient GPU code but the query processing does not sufficiently hide the memory transfer latency. In particular, the low bandwidth of the CPU-GPU interconnect limits the rate to supply data to the GPU because the time to transfer memory between CPU and GPU exceeds the kernel execution time significantly. We observe that optimizing for a high tuple ingestion rate over CPU-GPU interconnects improves the query processing throughput. Therefore, we argue that our framework allows for optimizations that are orthogonal to our GPU-based execution. For example, we believe that employing NVLink to expand the memory bandwidth helps us in achieving higher throughput due to an improved ingestion rate [13]. We expect similar query processing throughput in a GPU-based execution when the CPU-GPU bandwidth is comparable or higher to the bandwidth between CPU and main memory (cf. NVIDIA Grace Hopper architecture).

In addition, we have illustrated that our framework improves the abstraction level for writing GPU-specific query code and enables developers to connect with existing components of NebulaStream's aggregation pipeline. Thus, we lay the foundation for the implementation of more complex query operators using our framework. We conclude that our framework is successful in providing the operator developer useful abstractions for a GPU-based query processing engine in a data streaming system. Note that such a data streaming system does not have to only use the GPU. Instead, we want to emphasize that query processing can benefit from CPU-GPU co-processing, which our framework allows seamlessly. By employing a hybrid execution model, we can use a dynamic workload-adaptive strategy for offloading work to the GPU.

5. Related Work

Menon et al. [19] introduce the relaxed operator fusion (ROF) query processing model to enable SIMD vectorization in compilation-based database systems. In the ROF model, the query compiler embeds staging points into the query plan for the purpose of strategic materialization.

Thus, the generated query code exploits data-parallelism in the buffered tuple data, which is not the case when the generated code processes one tuple-at-a-time. Nevertheless, Menon et al. [19] do not tailor the query execution towards the GPU execution model and do not emphasize on creating maintainable GPU-specific query code. In our work, we found that the idea of materializing tuples at specific points in the query plan is necessary for enabling GPU query processing using Nautilus. However, we also view the materialization as a necessary step towards transforming the processing flow from tuple-at-a-time to buffer-at-a-time at any suitable point in the query plan. With this reversible transformation, we can implement the kernel programming model inside a new class of operators in the Nautilus framework that receive tuple buffers instead of single tuples.

Chrysogelos et al. [3] introduce the HetExchange framework to support intra-device data parallelism (GPU), inter-core task-parallelism (CPU) and heterogeneous cross-device parallelism in query processing on modern CPU-GPU systems. HetExchange inserts control and data flow operators into the query plan to facilitate the transfer of control flow as well as data, respectively, between producers and consumers. HetExchanges uses operator templates to specialize the code generation for different devices. In contrast, we opt to provide a framework that enables the developer to program GPU-specific operators in a maintainable manner.

SABER by Koliousis et al. [20] is a hybrid data stream processing engine designed for use with multi-core CPUs and many-core GPUs. In SABER, a query task is a batch of data bundled together with the operator graph of a query. SABER executes a query task according to a novel look-ahead scheduling algorithm that assigns the task to the either the CPU or the GPU based on expected performance. For each operator, SABER populates pre-defined GPU code templates with query-specific information such as schemata, selection predicates and aggregation functions. This is in contrast to our work, where we dynamically generate a GPU kernel from a pipeline of GPU-specifc operators. To this end, we utilize the foundations of the Nautilus query compiler, i.e., operator tracing in conjunction with the produce/consume model [5].

Nugroho et al. [21] evaluate the performance of GPU-based stream join algorithms to identify the configuration parameter space for different workloads. They develop a guideline for practitioners to choose the correct parameters for their own use case. For a given workload, they come to the conclusion that the use of a GPU can improve throughput but can also have a negative impact on the performance with the wrong choice of parameters.

6. Conclusion

We proposed a framework for compiling data stream queries to efficient GPU code. In our experiments, we studied the performance of our framework when used as part of the NebulaStream platform. To this end, we measured the end-to-end query processing throughput by using state-of-the-art data streaming workloads. In our evaluation, we showed that the lack of existing techniques (e.g., bandwidth expansion) to mitigate the CPU-GPU memory bottleneck in NebulaStream also showed a clear divide in performance between the CPU-based and GPU-based configuration. Future work needs to evaluate and improve the fine balance between abstraction level and performance as well as sufficiently hide the memory transfer latency.

With our work, we pave the way for GPU-accelerated stream processing in a compilation-based query engine.

Acknowledgments

This work was partially funded by the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407) and by SAP.

References

[1] V. Rosenfeld, S. Breß, V. Markl, Query Processing on Heterogeneous CPU/GPU Systems, ACM Comput. Surv. 55 (2022). URL: https://doi.org/10.1145/3485126. doi:10.1145/3485126.

[2] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, G. Saake, Gpu-accelerated database systems: Survey and open challenges, Trans. Large Scale Data Knowl. Centered Syst. 15 (2014) 1–35. URL: https://doi.org/10.1007/978-3-662-45761-0_1. doi:10.1007/978-3-662-45761-0_1.

[3] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, A. Ailamaki, HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines, Proc. VLDB Endow. 12 (2019) 544–556. URL: https://doi.org/10.14778/3303753.3303760. doi:10.14778/3303753.3303760.

[4] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, X. Du, FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures, in: Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC'20, USENIX Association, USA, 2020.

[5] T. Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware, Proc. VLDB Endow. 4 (2011) 539–550. URL: https://doi.org/10.14778/2002938.2002940. doi:10.14778/2002938.2002940.

[6] S. Borkar, A. A. Chien, The Future of Microprocessors, Commun. ACM 54 (2011) 67–77. URL: https://doi.org/10.1145/1941487.1941507. doi:10.1145/1941487.1941507.

[7] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, V. Markl, Analyzing Efficient Stream Processing on Modern Hardware, Proc. VLDB Endow. 12 (2019) 516–530. URL: https://doi.org/10.14778/3303753.3303758. doi:10.14778/3303753.3303758.

[8] H. Funke, S. Breß, S. Noll, V. Markl, J. Teubner, Pipelined Query Processing in Coprocessor Environments, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1603–1618. URL: https://doi.org/10.1145/3183713.3183734. doi:10.1145/3183713.3183734.

[9] S. Breß, B. Köcher, H. Funke, T. Rabl, V. Markl, Generating Custom Code for Efficient Query Execution on Heterogeneous Processors, The VLDB Journal 27 (2017) 797–822.

[10] H. Funke, J. Teubner, Data-Parallel Query Processing on Non-Uniform Data, Proc. VLDB

Endow. 13 (2020) 884–897. URL: https://doi.org/10.14778/3380750.3380758. doi:10.14778/3380750.3380758.
[11] P. M. Grulich, A. P. Lepping, D. P. A. Nugroho, V. Pandey, B. Del Monte, S. Zeuch, V. Markl, Query compilation without regrets, Proc. ACM Manag. Data 2 (2024). URL: https://doi.org/10.1145/3654968. doi:10.1145/3654968.
[12] S. Zeuch, A. Chaudhary, B. Monte, H. Gavriilidis, D. Giouroukis, P. Grulich, S. Breß, J. Traub, V. Markl, The NebulaStream Platform: Data and Application Management for the Internet of Things, in: Conference on Innovative Data Systems Research (CIDR), 2020.
[13] C. Lutz, S. Breß, S. Zeuch, T. Rabl, V. Markl, Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1633–1649. URL: https://doi.org/10.1145/3318464.3389705. doi:10.1145/3318464.3389705.
[14] C. D. Yu, W. Wang, D. Pierce, A CPU-GPU Hybrid Approach for the Unsymmetric Multifrontal Method, Parallel Comput. 37 (2011) 759–770. URL: https://doi.org/10.1016/j.parco.2011.09.002. doi:10.1016/j.parco.2011.09.002.
[15] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Bress, A. Katsifodimos, T. Rabl, V. Markl, Scotty: General and Efficient Open-Source Window Aggregation for Stream Processing Systems, 2020. URL: https://www.redaktion.tu-berlin.de/fileadmin/fg131/Publikation/Papers/Traub_TODS-21-Scotty_preprint.pdf.
[16] P. A. Tucker, K. Tufte, V. Papadimos, D. Maier, Nexmark – a benchmark for queries over data streams, 2008. URL: https://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf.
[17] NVIDIA Corporation, CUDA Profiler Users Guide (v12.6), https://docs.nvidia.com/cuda/profiler-users-guide, 2024. Online, accessed 2024-08-15.
[18] NVIDIA Corporation, CUDA C++ Best Practices Guide (v12.6), https://docs.nvidia.com/cuda/cuda-c-best-practices-guide, 2024. Online, accessed 2024-08-15.
[19] P. Menon, T. C. Mowry, A. Pavlo, Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last, Proc. VLDB Endow. 11 (2017) 1–13. URL: https://doi.org/10.14778/3151113.3151114. doi:10.14778/3151113.3151114.
[20] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, P. Pietzuch, SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 555–569. URL: https://doi.org/10.1145/2882903.2882906. doi:10.1145/2882903.2882906.
[21] D. P. A. Nugroho, P. M. Grulich, S. Zeuch, C. Lutz, S. Bortoli, V. Markl, Benchmarking stream join algorithms on gpus: A framework and its application to the state-of-the-art, in: L. Tanca, Q. Luo, G. Polese, L. Caruccio, X. Oriol, D. Firmani (Eds.), Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28, OpenProceedings.org, 2024, pp. 188–200. URL: https://doi.org/10.48786/edbt.2024.17. doi:10.48786/EDBT.2024.17.