Towards A GPU-Accelerated Stream Processing Engine Through Query Compilation

LWDA'24
September 23-25, 2024, Würzburg, Germany

Florian Schmeller,

Dwi P. A. Nugroho,

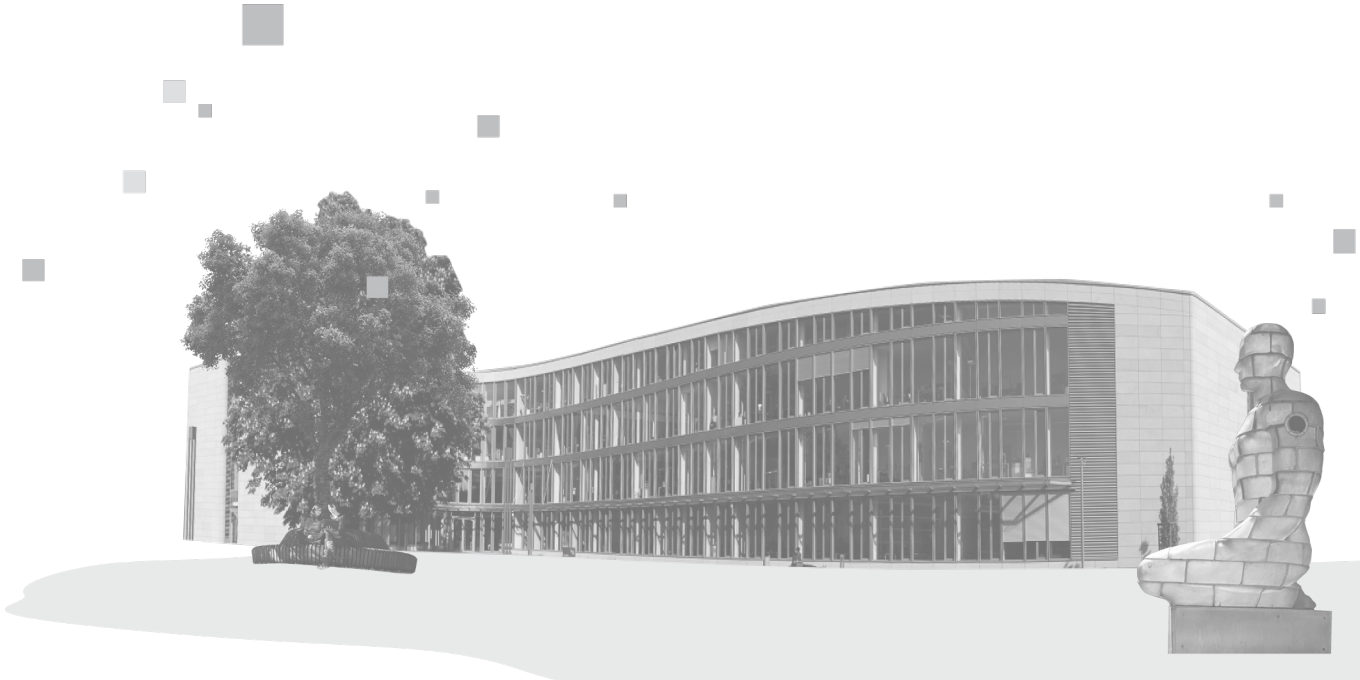
Steffen Zeuch,

Tilmann Rabl

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Design IT.
Create Knowledge.

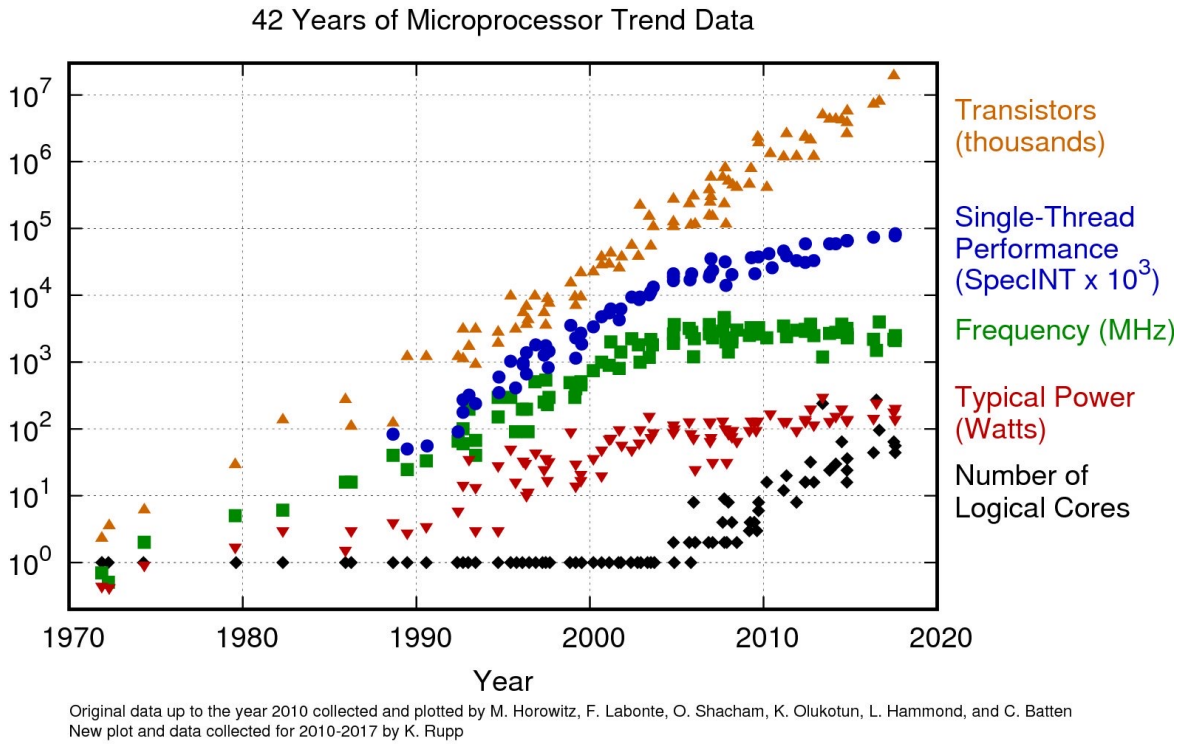
www.hpi.de



Motivation

Hardware Trends

- Frequency scaling stagnated since 2006
  - Computer systems have become increasingly heterogeneous
  - Increase in specialized hardware, cores
- GPUs are popular co-processors for throughput-oriented applications
  - Large number of low complexity cores
  - Data-parallel execution model



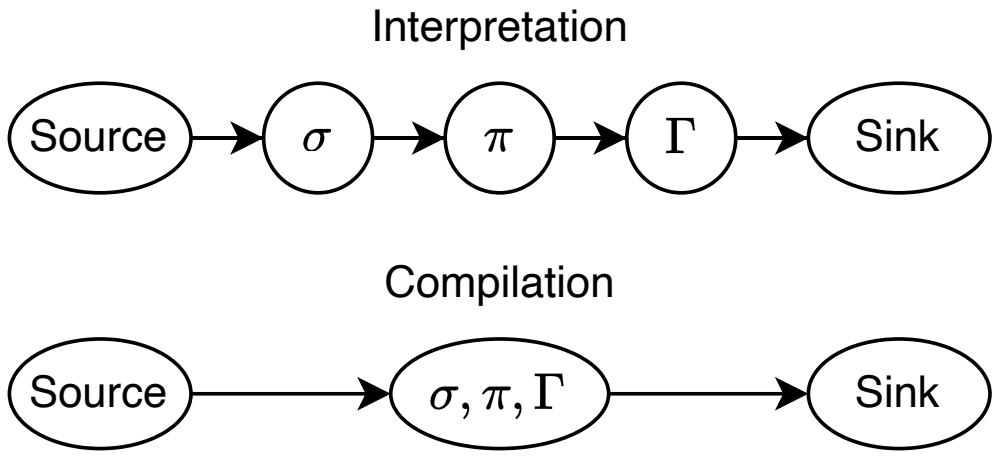
Motivation

Stream Processing Systems

- Current stream processing systems do not fully exploit modern hardware
  - Query compilation leverages execution environment for better performance
  - Generated code is hard to understand, debug and profile

- Current approaches to query compilation for GPU are not developer-friendly

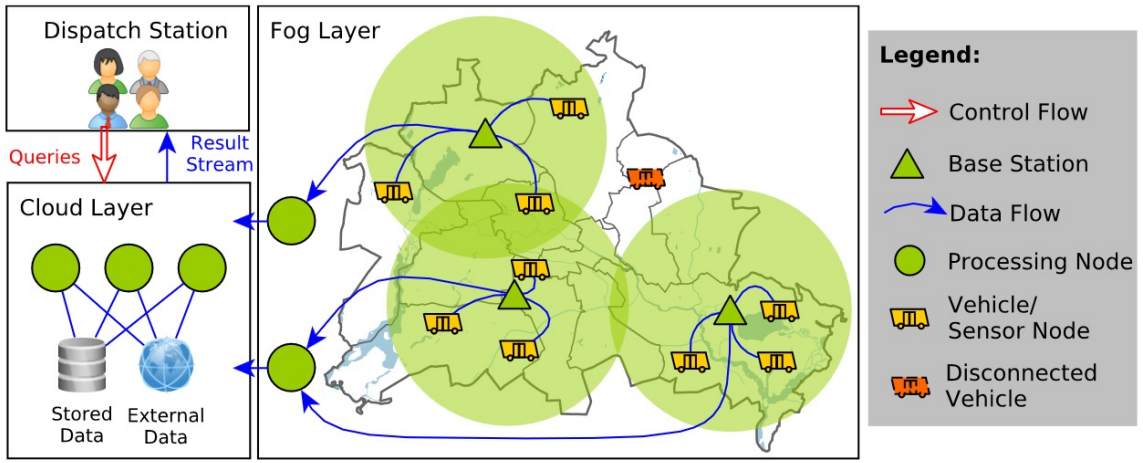
- This talk: Maintainable GPU code generation for streaming operators



Adapted from Zeuch et al., *Analyzing Efficient Stream Processing on Modern Hardware*, VLDB 2019

NebulaStream

- NebulaStream [Ze20] is a data management system for the Internet of Things (IoT)
  - Dynamic, streaming workloads
  - Task-based parallelism
- Nautilus [Gr24] framework unifies query interpretation and query compilation
  - One operator implementation for both
  - Creates trace from symbolic execution
  - Used in NebulaStream query engine



IoT application (from [Ze20])

[Gr24] Grulich et al., *Query Compilation Without Regrets*, SIGMOD 2024
[Ze20] Zeuch et al., *The NebulaStream Platform for Data and Application Management in the Internet of Things*, CIDR 2020

Related Work

- Relaxed Operator Fusion model [Me17] enables SIMD in compilation-based engines
  - Staging points in query plan allow strategic materialization
- SABER [Ko16] is a hybrid CPU-GPU stream processing engine
  - Compiled execution using code templates
- HetExchange [Ch19] is a framework for hybrid CPU-GPU query processing
  - Operator templates for different devices

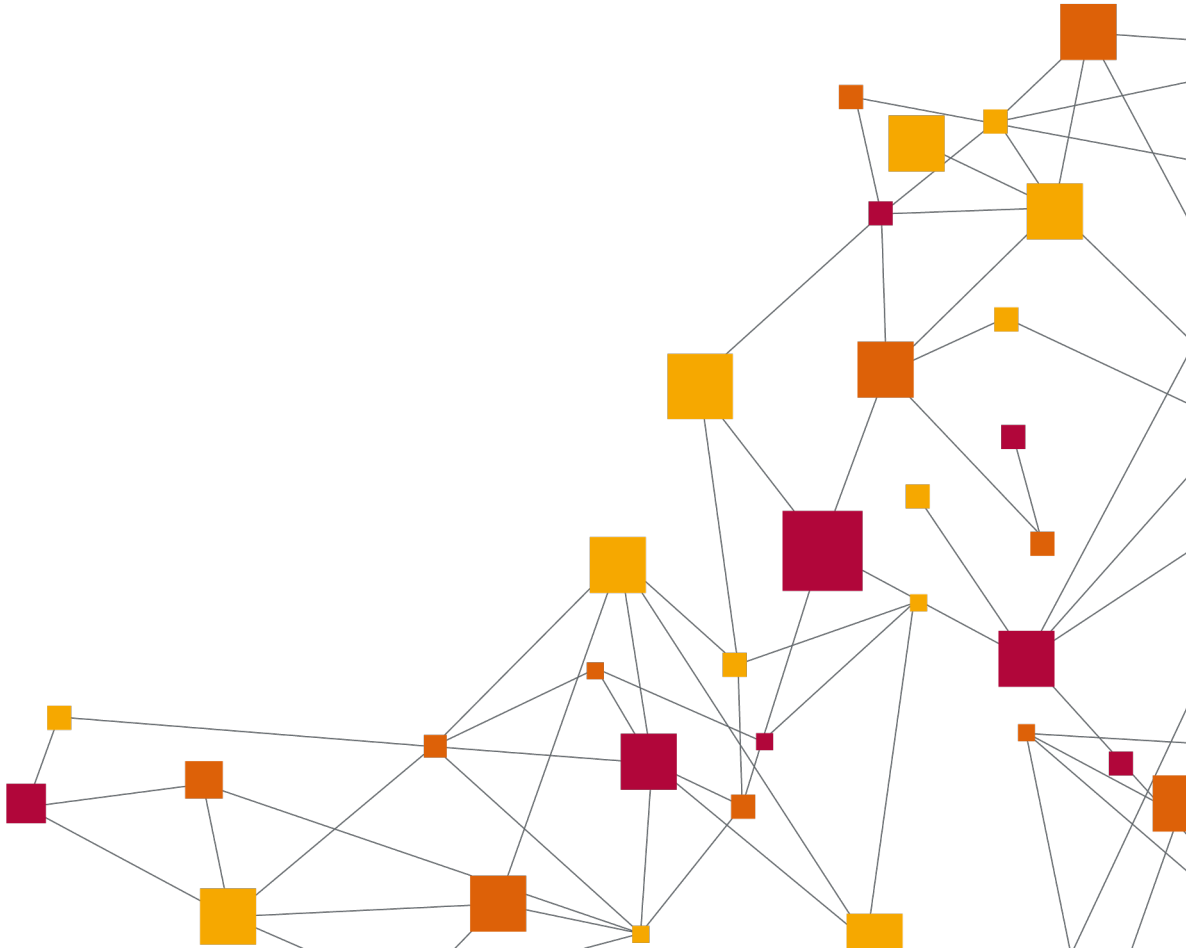
[Me17] Menon et al., *Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last*, VLDB 2017
[Ko16] Koliousis et al., *SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures*, SIGMOD 2016
[Ch19] Chrysogelos et al., *HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines*, VLDB 2019

Agenda

1. Compilation-based Stream Processing on the GPU
2. Evaluation
3. Conclusion

Compilation-based Stream Processing on the GPU

Hasso Plattner Institut
Digital Engineering · Universität Potsdam



Design IT.
Create Knowledge.

www.hpi.de

Compilation-based Stream Processing on the GPU

Contributions

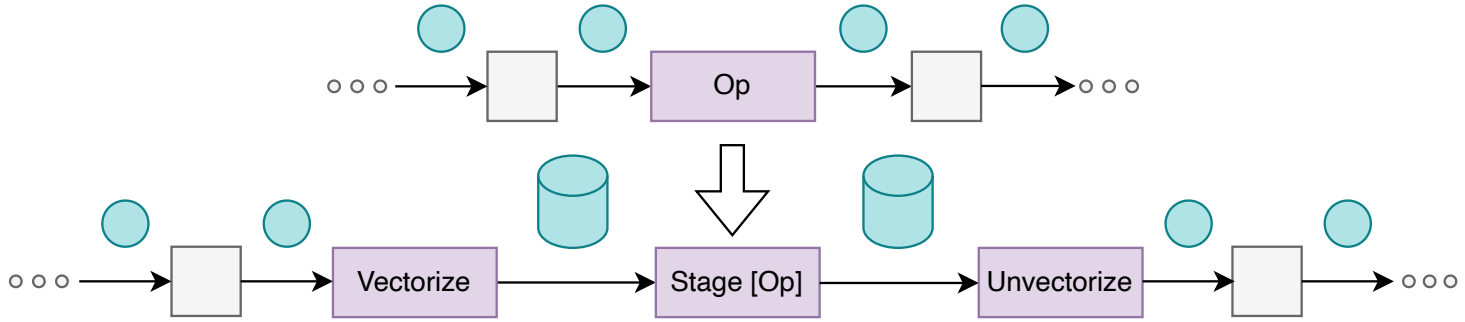
1. Designed a framework to support the compilation of a GPU kernel into a query plan

2. Integrated our framework into NebulaStream and implemented three data stream operators

3. Demonstrated the impact of low memory bandwidth on throughput-oriented systems

Compilation-based Stream Processing on the GPU

From Query Plan to GPU Code



1. Identify vectorizable operators

a) Traverse query plan for operator *Op*

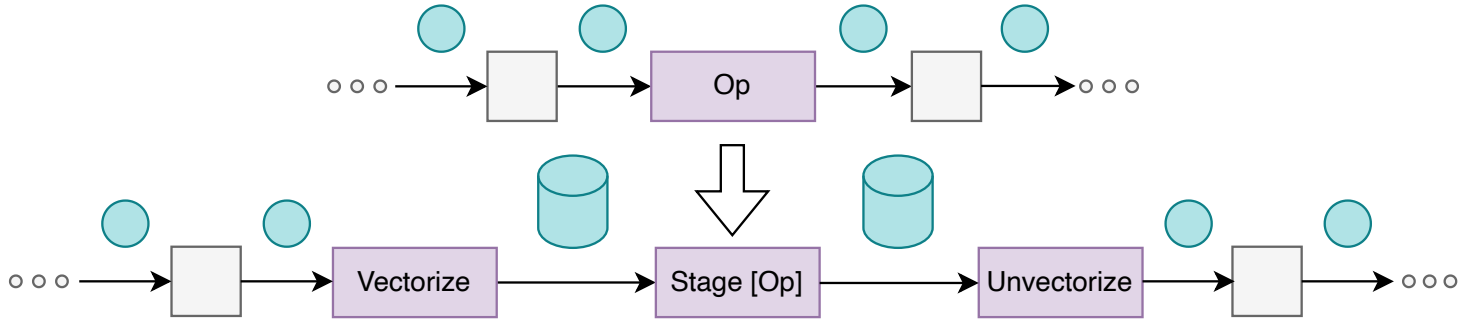
b) Add Vectorize, Unvectorize around *Op*

c) Wrap *Op* in Stage operator

d) Replace Stage with Kernel operator

Compilation-based Stream Processing on the GPU

From Query Plan to GPU Code



1. Identify vectorizable operators

a) Traverse query plan for operator *Op*

b) Add Vectorize, Unvectorize around *Op*

c) Wrap *Op* in Stage operator

d) Replace Stage with Kernel operator

2. Compile Kernel operator to GPU code

a) Traverse query plan for Kernel operator

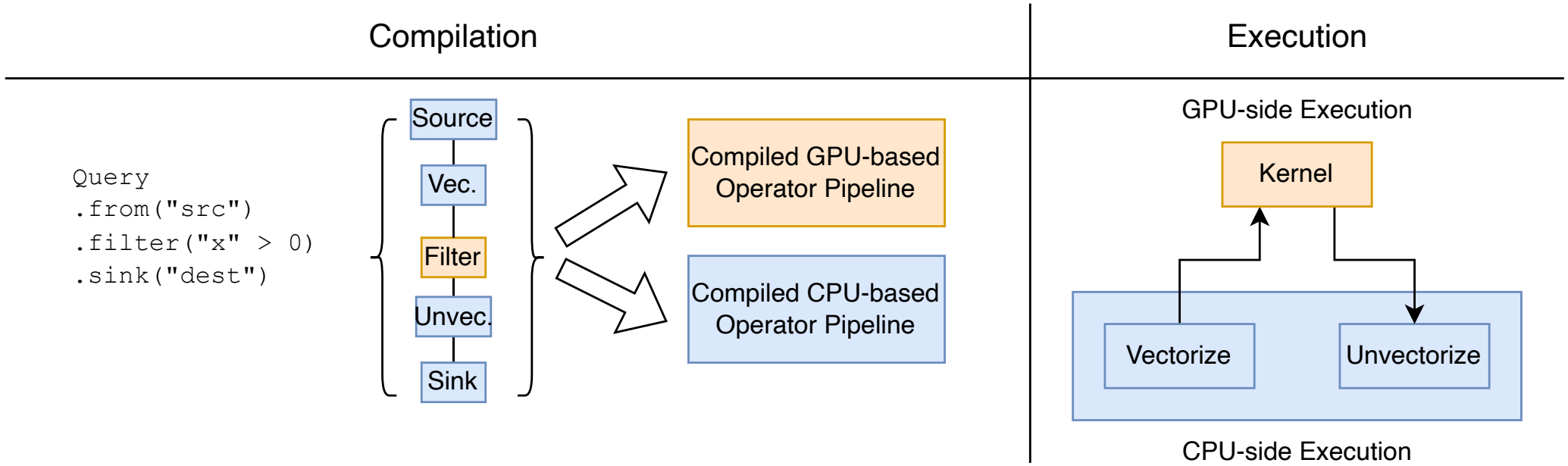
b) Trace symbolic execution of *Op*

c) Transform trace to NES IR

d) Transform NES IR to CUDA C++

Compilation-based Stream Processing on the GPU

From Compilation to Execution



We compile the query plan to CPU and GPU executable code.

Compilation-based Stream Processing on the GPU

Nautilus-style Operator Implementation

Regular Map Operator

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

Vectorized Map Operator

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

Compilation-based Stream Processing on the GPU

Nautilus-style Operator Implementation

Regular Map Operator

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

Vectorized Map Operator

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

Compilation-based Stream Processing on the GPU

Nautilus-style Operator Implementation

Regular Map Operator

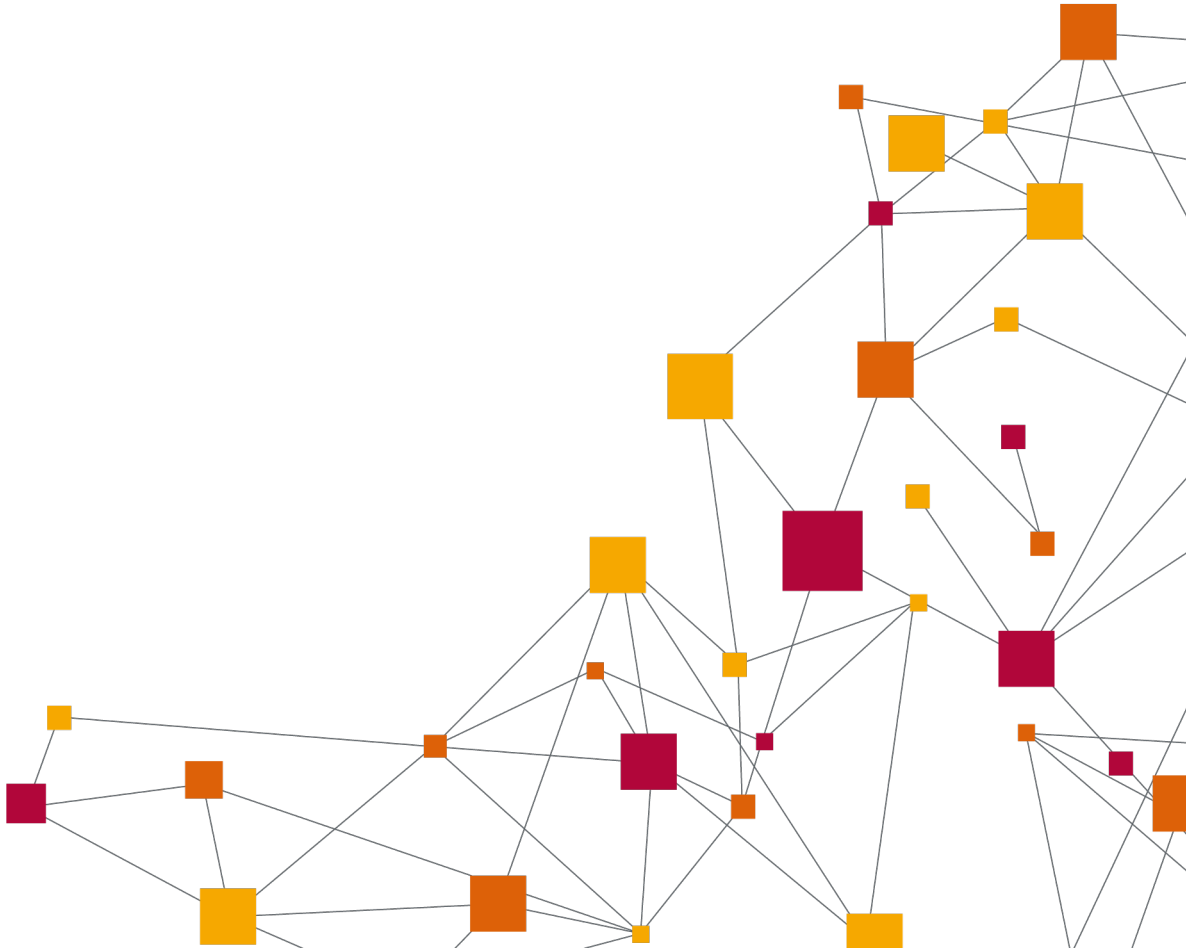
```
void Map::execute(Context& ctx, Record& record)
{
    mapExpression->execute(record);
    if (hasChild()) {
        child->execute(ctx, record);
    }
}
```

Vectorized Map Operator

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

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Evaluation



Design IT.
Create Knowledge.

www.hpi.de

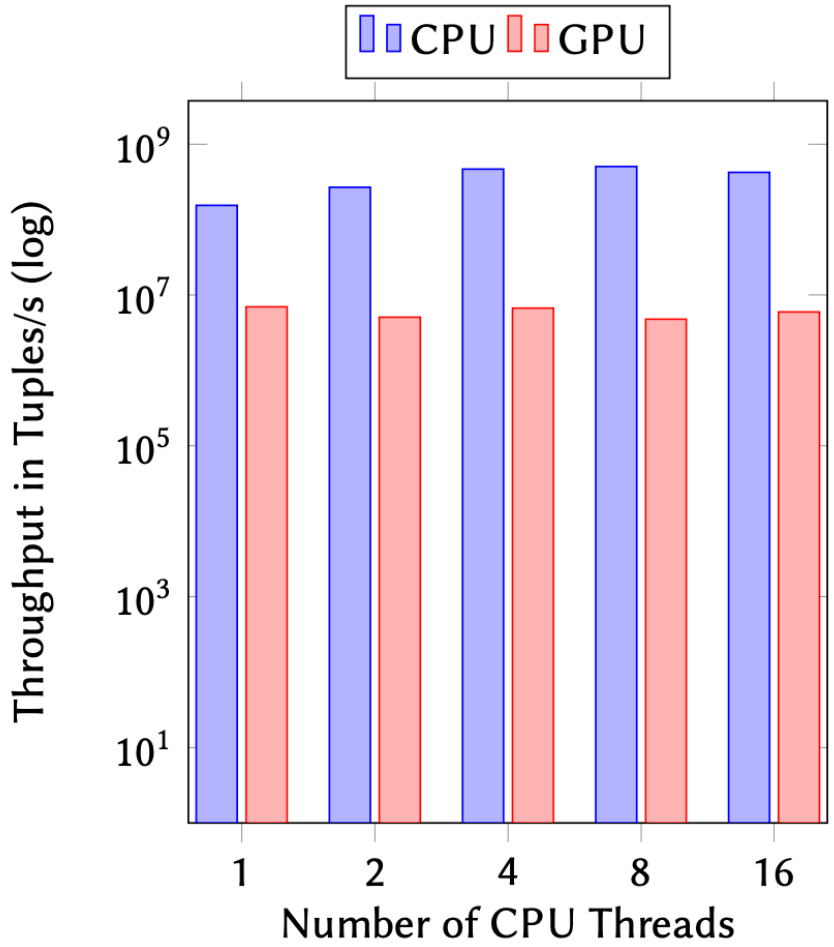
Evaluation

Experimental Setup

Component	Specification
CPU	Intel Xeon Gold 5115 @ 2.40 GHz
GPU	NVIDIA Tesla V100 (PCIe 3.0 16 GB/s)
System Memory	192 GB
Operating System	Ubuntu 22.04 LTS
Compiler	Clang 16.0.1
GPU Framework	CUDA Toolkit 11.7
Experiment Platform	NebulaStream v0.5
Workload	NEXMark queries Q1, Q2

Evaluation

End-to-End Query Processing Throughput



(a) NEXMark query Q1.

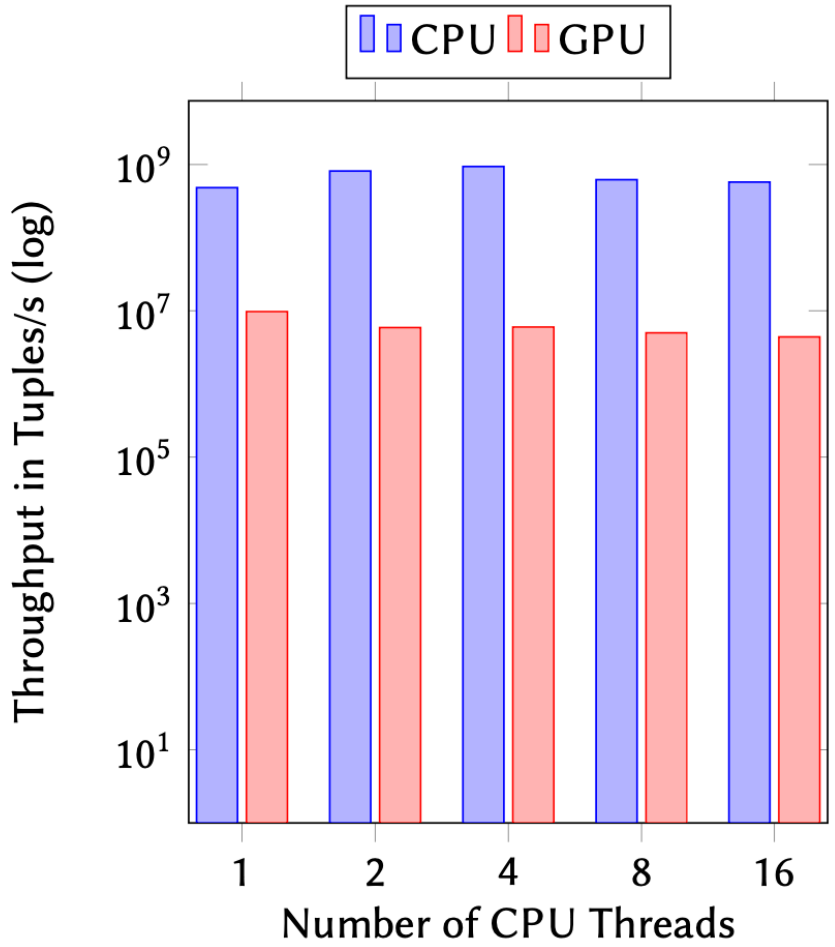
```
Query::from("Bid")
        .map(Attribute("price") = Attribute("price") * 0.89)
        .sink(NullSink());
```

Throughput in CPU-based execution (blue) is higher than in GPU-based execution (red)

Evaluation

End-to-End Query Processing Throughput

```
Query::from("Bid")
        .filter(Attribute("auction") == 1007
            || Attribute("auction") == 1020
            || Attribute("auction") == 2001
            || Attribute("auction") == 2019
            || Attribute("auction") == 2087)
        .sink(NullSink());
```

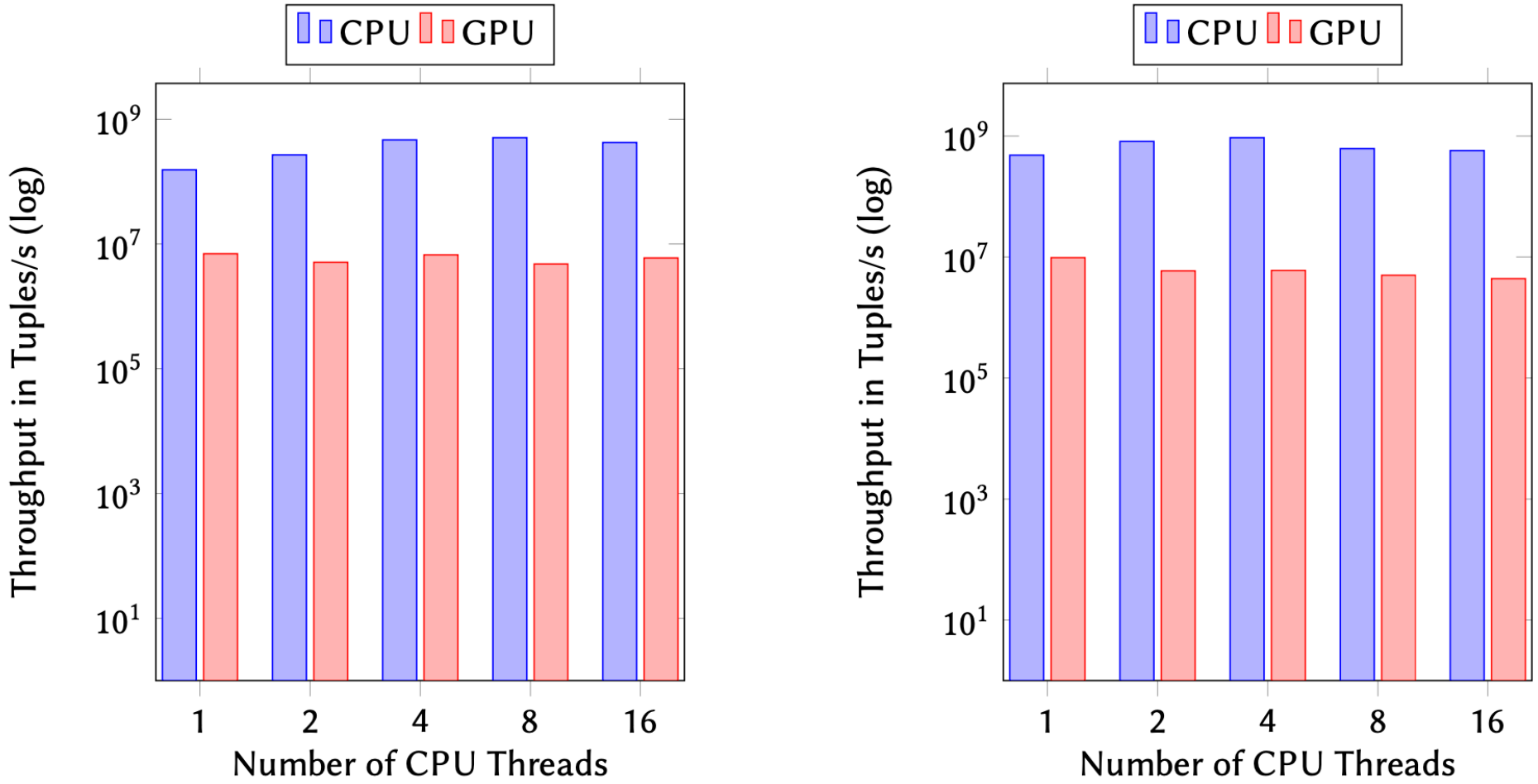


(b) NEXMark query Q2.

Throughput in CPU-based execution (blue) is higher than in GPU-based execution (red)

Evaluation

End-to-End Query Processing Throughput



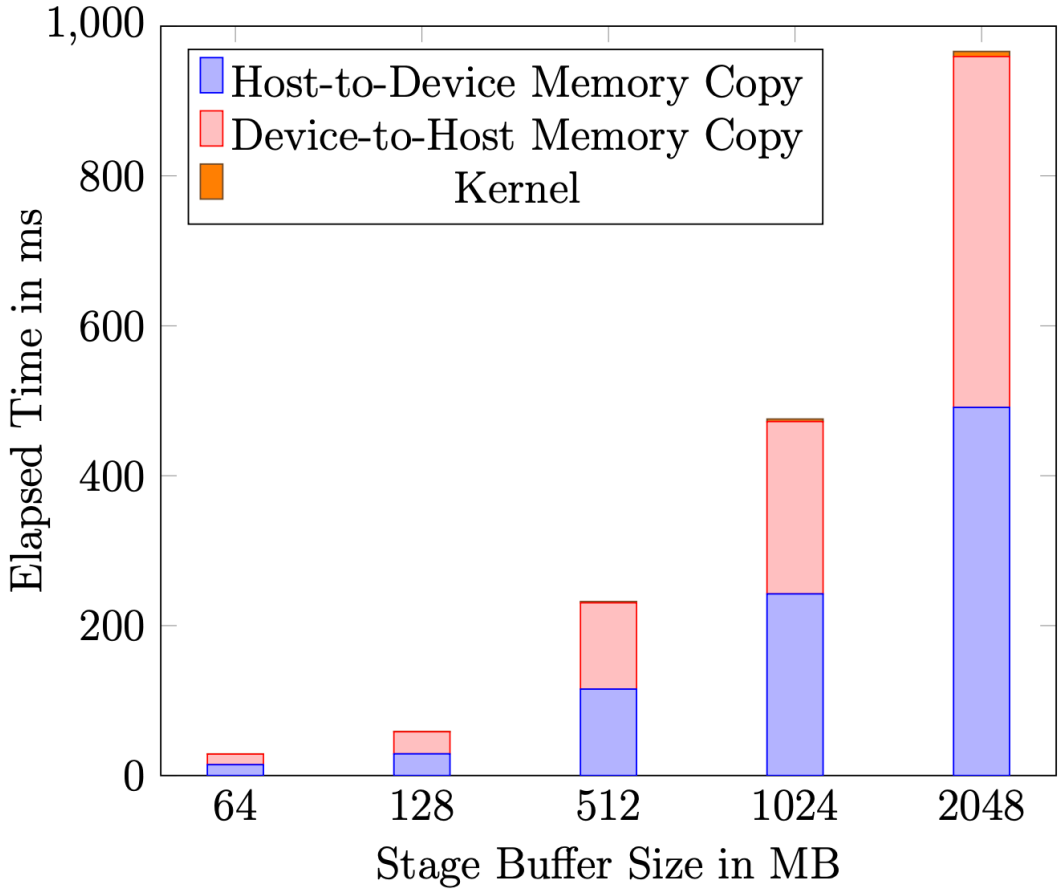
(a) NEXMark query Q1.

(b) NEXMark query Q2.

Throughput in CPU-based execution (blue) is higher than in GPU-based execution (red)

Evaluation

Breakdown of NEXMark Q1



Memory transfer time dominates execution time

Evaluation

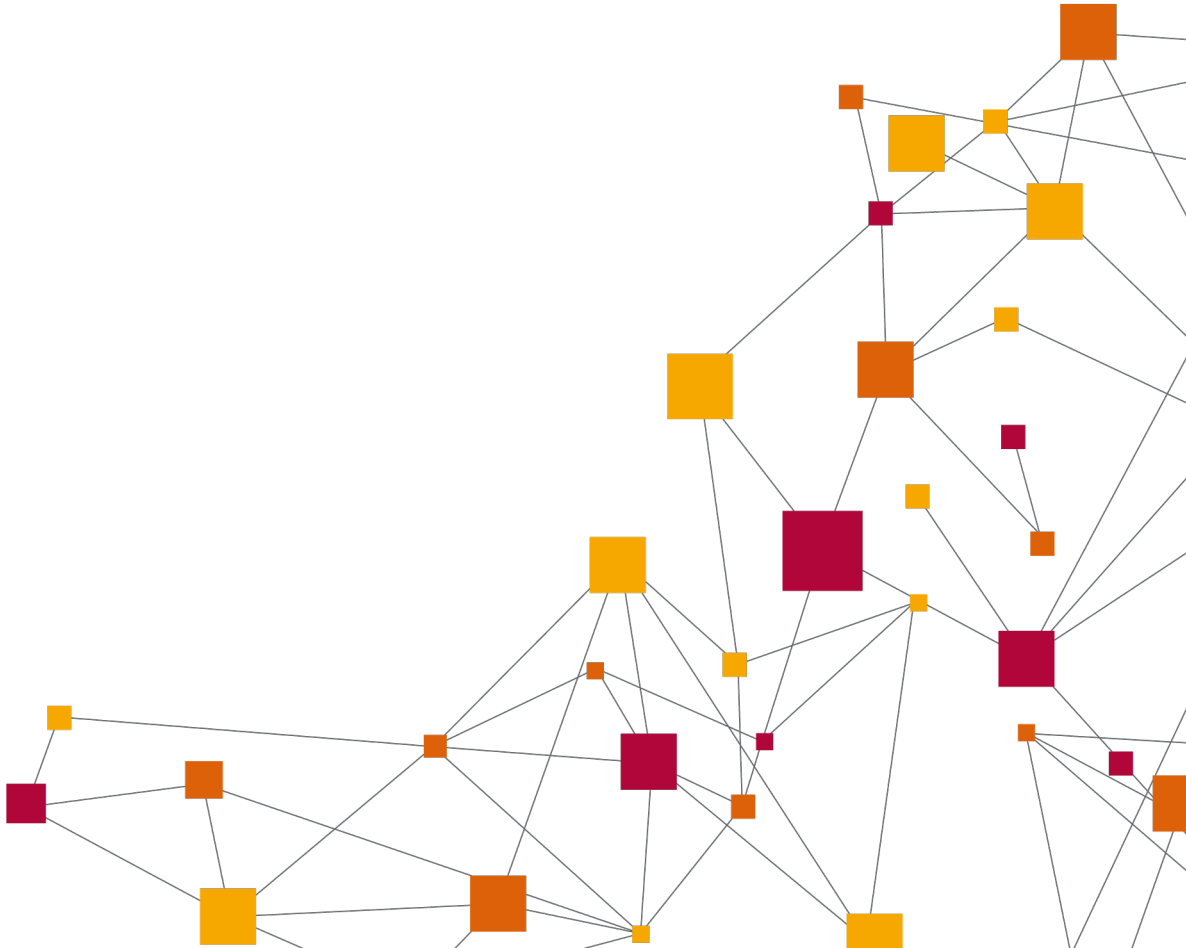
Discussion & Main Takeaways

- CPU-GPU memory bandwidth negatively impacts throughput
  - Hide latency using complex operator like join

- Bandwidth increase improves tuple throughput
  - CPU-GPU bandwidth much lower than CPU-memory bandwidth in evaluation
  - Recent systems use NVLink to support faster data transmission

- Code generation benefits from knowledge of workload characteristics

Conclusion



Design IT.
Create Knowledge.

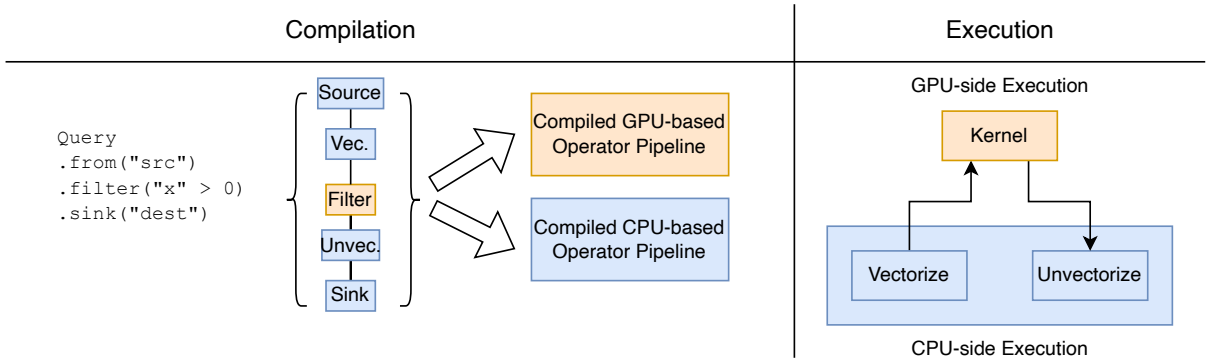
www.hpi.de

Conclusion

Thank you

- Query compilation capitalizes on recent hardware trends

- Framework supports data parallelism in Nautilus-style operators
  - High abstraction level for operator developers (target architectures, GPU frameworks)
  - Other vectorization methods like SIMD possible



- Evaluation shows importance of memory bandwidth in GPU-based stream processing

Evaluation

Map Operator Benchmark

	Pageable Memory			Pinned Memory		
Buffer Size [MB]	64	512	1024	64	512	1024
Throughput [M tup/s]	6.9	7.0	7.4	7.5	7.5	7.1
Bandwidth [GB/s]						
CPU-GPU	4.6	4.6	4.7	12.3	12.4	12.4
GPU Global	520.5	502.2	503.4	518.2	501.7	501.9
GPU Requested	130.2	125.6	125.9	129.6	125.4	125.5

Improvement in CPU-GPU bandwidth leads to better throughput

No coalescing of memory access leads to worse requested GPU bandwidth