Pipelined Query Processing in Coprocessor Environments

Henning Funke
TU Dortmund University
henning.funke@udo.edu

Sebastian Breß
DFKI GmbH
sebastian.bress@dfki.de

Stefan Noll
TU Dortmund University
stefan.noll@udo.edu

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

Jens Teubner
TU Dortmund University
jens.teubner@udo.edu

ABSTRACT

Query processing on GPU-style coprocessors is severely limited by the movement of data. With teraflops of compute throughput in one device, even high-bandwidth memory cannot provision enough data for a reasonable utilization.

Query compilation is a proven technique to improve memory efficiency. However, its inherent tuple-at-a-time processing style does not suit the massively parallel execution model of GPU-style coprocessors. This compromises the improvements in efficiency offered by query compilation. In this paper, we show how query compilation and GPU-style parallelism can be made to play in unison nevertheless. We describe a compiler strategy that merges multiple operations into a single GPU kernel, thereby significantly reducing bandwidth demand. Compared to operator-at-a-time, we show reductions of memory access volumes by factors of up to 7.5x resulting in shorter kernel execution times by factors of up to 9.5x.

1 INTRODUCTION

GPUs are frequently used as powerful accelerators for query processing. As the arithmetic throughput of the coprocessor peaks in the teraflop range, it becomes a challenge to provision enough data. For this reason, hardware vendors equip graphics cards with high bandwidth memory that has read and write rates of hundreds of GB/s. Still, memory intensive applications such as query processing fall behind regarding the cost of data movement for different reasons. Figure 1 shows the path of relational data through the hierarchical memory levels in a typical coprocessor system. Along the path, several bandwidth
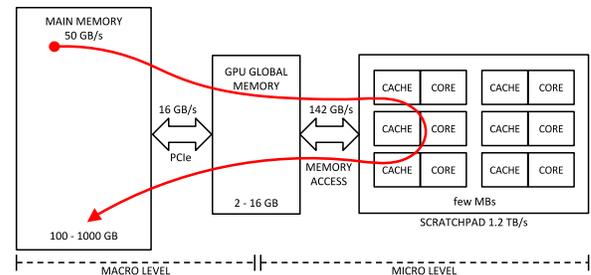
Figure 1: The path of a tuple through the memory levels of a coprocessor environment.

and capacity constraints need to be considered to achieve scalability and performance:

PCIe / OpenCAPI / NVLink. A widely-acknowledged problem is the data transfer bottleneck between the host system and the coprocessor [9], typically via PCIe. Due to the coprocessor's limited memory capacity, data transfers are necessary *during* computations. With an order of magnitude between internal and external memory bandwidth, database developers are challenged with data locality-aware algorithms that efficiently use inter-processor communication. Recent technologies, i.e., OpenCAPI and NVLink, increase the bandwidth over PCIe, shifting the bottleneck towards GPU global memory.

GPU Global Memory. The fine-grained data parallelism of a GPU typically requires that kernels perform additional passes over the data. Performing multiple passes, however, can significantly inflate memory loads and can cause a bandwidth bottleneck especially for random memory accesses.

Main-Memory. A recent development are integrated GPU-style coprocessors that can directly access the memory of the host CPU. Such an *Accelerated Processing Unit* (APU) allows to use massively parallel processing without additional data transfers. However, the available memory bandwidth is lower than that of a dedicated GPU (30 GB/s vs. hundreds of GB/s).

Scratchpad Memory[1]. Scratchpad memory is located on-chip and placed next to each compute unit of a GPU. It can be controlled as an explicit cache for low-level computations and offers a very high bandwidth. However, the capacity is

---

[1]We use the term *scratchpad memory* to disambiguate *shared memory* for CUDA and *local memory* for OpenCL.

RUN-TO-FINISH – **input:** R, **output:** P

```
move R Host → GPU
tmp ← op1(R)        /* invoke first GPU kernel */
P ← op2(tmp)        /* invoke second GPU kernel */
move P GPU → Host
```

Figure 2: Run-to-finish execution of two successive kernels.

limited to 16 KB – 96 KB per core which makes it challenging to use it for large-scale computations.

Contributions

In this work, we present our new query compiler HORSEQC. We designed HORSEQC to account for the hierarchical memory structure of coprocessor environments and for the inherent bandwidth limitations. Our main contribution is to show how various existing techniques can be combined and extended to build an efficient query processing engine on coprocessors.

(1) We *analyze the bandwidth limitations* in several execution models (cf. Section 2).
(2) We show a way to *integrate query compilation* into a coprocessor-accelerated DBMS (cf. Sections 3 and 4).
(3) We present solutions to efficiently process all processing steps of a pipeline in a *single pass over the data* (cf. Sections 5 and 6).
(4) We describe how these parts play together in an overall system (Section 7) and evaluate our proposed concepts (cf. Section 8).
(5) We discuss our results (cf. Section 9) and related work (cf. Section 10), and conclude in Section 11.

GPU-accelerated database systems have used different *macro execution models* in the past. Orthogonally, our work describes a *micro execution model* that can be integrated with different existing macro execution models.

2 MACRO EXECUTION MODEL

We first analyze macro execution models that various systems have used in the past. To evaluate a relational query operator, state-of-the-art systems will select a number of primitives and execute the corresponding kernel sequence on the GPU. To feed the kernels with data, the macro execution model defines how data transfers will be interleaved with kernel executions. Here, the data movement from kernel to kernel may result in additional bandwidth demand as compared to conventional systems. To understand the effect, we study the implications that existing macro execution models have on the use of bandwidth at multiple levels (PCIe, GPU global memory, etc.). As a poster child, we profiled the execution of Query 3.1 from the star schema benchmark (SSB) [24]. The query was executed at scale factor 10 with CoGaDB [6] on a NVIDIA GTX970 GPU[2] (details are given in Appendix Section A). In the following, we discuss three macro execution models: *run-to-finish*, *kernel-at-a-time* and *batch processing*.

---

[2]We measured 146.1 GB/s GPU global memory bandwidth in a host system with 16 GB/s bidirectional PCIe bandwidth.

KERNEL-AT-A-TIME – **input:** R, **output:** P

```
foreach rᵢ in R=r₁ ∪ ··· ∪ rₘ do
    move rᵢ Host → GPU
    mᵢ ← op1(rᵢ)      /* invoke first GPU kernel */
    move mᵢ GPU → Host (assemble into M)
foreach mⱼ in M=m₁ ∪ ··· ∪ mₙ do
    move mⱼ Host → GPU
    pⱼ ← op2(mⱼ)      /* invoke second GPU kernel */
    move pⱼ GPU → Host (assemble into P)
```

Figure 3: Kernel-at-a-time achieves scalability by transferring I/O for each kernel through PCIe.

2.1 Run-To-Finish (Not Scalable)

A straightforward way to execute a sequence of kernels is to first transfer all input, execute the kernels, and finally transfer all output. The approach, illustrated in Figure 2, has the advantage that intermediate data remains in GPU global memory in-between kernel executions and no significant PCIe transfers are necessary. However, run-to-finish has the disadvantage that it only works if *all* input, output, and intermediate data is small enough to fit in GPU memory. Run-to-finish macro execution models are used, e.g., by Ocelot [12] and CoGaDB [6]. The *lack of scalability* leads us to evaluate the following execution models.

2.2 Kernel-At-A-Time

To process large data on coprocessors, we can execute each kernel on blocks of data. The pseudocode of this approach is shown in Figure 3. Processing blocks of data requires algorithm choices that can deal with partitioned inputs. Joins or aggregations, for instance, can only be processed in this mode if their internal state (e.g. a hash table) can fit in GPU global memory.

We analyze the data movement of kernel-at-a-time for SSB Query 3.1. Blocks are first moved via PCIe from the host to the coprocessor and then read by the kernel from GPU global memory (output passes both levels vice-versa). In this way, the data volumes for GPU global memory accesses equal the data volume transferred via PCIe, plus the cost to build up the hash tables in GPU global memory (0.4 GB here). Figure 5a shows the resulting data movement.

In the figure, the arrows annotated with data volumes represent PCIe transfers and GPU global memory accesses. We aggregated the data volumes by kernel types (e.g. scan, gather) and show only the most important kernels responsible for 88.2% of the memory traffic. Given a PCIe bandwidth of 16 GB/s, all PCIe transfers together require at least 350 ms to complete. This exceeds the aggregate time for GPU global memory access by a factor of **5.8x**. For kernel-at-a-time processing *the PCIe link is clearly the bottleneck.*

Kernel-at-a-time processing is used to scale out individual operators [16]. Unified virtual addressing (UVA) produces the same low-level access pattern, albeit transparent to the system developer.

BATCH PROCESSING – **input:** R, **output:** P

```
foreach rᵢ in R=r₁ ∪ ··· ∪ rₘ do
    move rᵢ Host → GPU
    tmpᵢ ← op1(rᵢ)    /* invoke first GPU kernel */
    pᵢ ← op2(tmpᵢ)    /* invoke second GPU kernel */
    move pᵢ GPU → Host (assemble into P)
```

Figure 4: Batch processing executes multiple kernels for each block that is transferred via PCIe.

2.3 Batch Processing

We can alleviate PCIe bandwidth limitations by rearranging the operations of kernel-at-a-time. Instead of running kernels until a column is processed, we can short-circuit the transfer of intermediate results to the host. Batch processing achieves this by reusing the output of the previous operation (op1) as input for the next operation (op2) instead of transferring to the host. This is applicable whenever intermediate batch results can be kept within GPU global memory. The corresponding pseudocode is shown in Figure 4.

We analyze the data movement cost with the example of SSB Query 3.1. The GPU global memory load is the same as for kernel-at-a-time processing, because each kernel reads and writes I/O to GPU global memory. We obtain the PCIe transfer cost using the transfer volumes of input columns of the query and output of the final result. Figure 5b shows the resulting data movement cost. Batch processing reduces the amount of PCIe transfers by a factor of **8.8x**. This shows that transferring data in blocks *and* performing multiple operators per block allows scalability and increases the efficiency compared to kernel-at-a-time.

Batch processing macro execution models have been used for coprocessing by GPUDB [36] and Hetero-DB [37]. Wu et al. [33] describe the concept as *kernel fission* and detect opportunities to omit PCIe transfers automatically.

Limitations. The lower amount of PCIe traffic can expose GPU global memory bandwidth as the next limitation. Batch processing reduces the PCIe transfer cost, but the amount of GPU global memory access remains unaffected. The memory access volume inside the device is now an order of magnitude larger which, despite the high bandwidth, takes longer to process than the PCIe bus transfers (Figure 5b). For this reason, batch processing SSB Query 3.1 is *not* limited by PCIe transfers, but by accesses to the (high-speed) GPU global memory. Since in typical query plans, I/O and hashing operations both address the same GPU global memory, the situation, in fact, may arise frequently in real-world workloads.

Other Queries. A limiting amount of global memory access can easily occur when many kernels are executed one after another. Karnagel et al. [15] show that a simple query with one selection and one aggregation operator already uses 13 kernels for processing. To determine the prevalence of GPU global memory bandwidth limitations, we profiled several queries
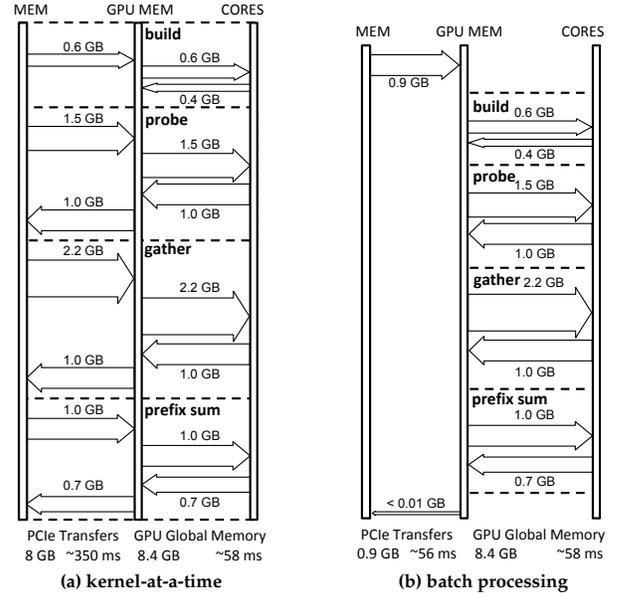

Figure 5: Data movement for processing SSB Query 3.1. While the throughput of (a) is limited by PCIe transfers, (b) exposes GPU global memory access as the next bottleneck.

Query	Passes	Query	Passes	Query	Passes
ssb11	7.5	ssb34	2.2	tpch5	7.2
ssb12	6.9	ssb41	7.4	tpch6	6.2
ssb13	6.7	ssb42	3.9	tpch7	**9.0**
ssb21	**9.6**	ssb43	3.5	tpch9	**9.0**
ssb22	**9.2**	tpch1	15.5	tpch10	5.8
ssb23	**9.1**	tpch2	14.5	tpch15	6.3
ssb31	**11.0**	tpch3	5.2	tpch18	**38.5**
ssb32	7.9	tpch4	6.6	tpch20	**10.5**
ssb33	7.5				

Table 1: Number of passes for benchmark queries. Out of 25 queries, 9 are definitely limited by GPU global memory.

from the TPC-H and SSB benchmark sets[3]. We look at the ratio of memory access to PCIe traffic as number of passes to assess the load on memory and bus links. Table 1 shows the number of passes for queries from the TPC-H and SSB benchmarks. With a symmetric memory load, we can afford $\frac{146\,\text{GB/s}}{2 \cdot 16\,\text{GB/s}} \sim 4$ to 5 passes before being limited by GPU global memory. While memory can adapt to asymmetric read and write loads, PCIe can service each direction with at most 16 GB/s. This changes the number of affordable passes for asymmetric workloads to $\frac{146\,\text{GB/s}}{16\,\text{GB/s}} \sim 9$ in the worst case. Queries that require more than 9 passes are always limited by memory bandwidth before being affected by the PCIe bottleneck. In Table 1 this is the case for 9 out of 24 queries, which indicates that it is crucial to reduce the GPU global memory load.

---

[3]Note that CoGaDB does not support all TPC-H queries yet.

3 MICRO EXECUTION MODEL

Tuning the macro level helps to remove the main bottleneck for scalability: data transfers over PCIe. However, the macro level change exposes a new bottleneck: the memory bandwidth of GPU global memory (cf. Section 2.3). To utilize the GPU global memory bandwidth more efficiently, we need to apply additional micro-level optimizations using *micro execution models* and combine them with the macro execution model (batch processing) to achieve scalability *and* performance.

Existing micro-level optimizations such as *vector-at-a-time* processing [38] and *query compilation* [23] utilize memory bandwidth more efficiently by leveraging pipelining in on-chip processor caches. Therefore, both techniques are promising candidates for opening up the bottleneck of limited GPU global memory bandwidth. However, vector-at-a-time processing and query compilation are designed in the context of CPUs. While it is highly desirable to apply both techniques in the context of GPUs, mapping the techniques from CPU to GPU is challenging, which we discuss in the following.

Vector-At-A-Time. To mediate the interpretation overhead of Volcano and the materialization overhead of operator-at-a-time, vector-at-a-time uses batches that fit in the processor caches. First, this reduces the number of `getNext()` calls from one per tuple to one per batch. Second, this makes materialization cheap because operators pick up the cached results of previous operators. On CPUs, vector-at-a-time benefits from batch sizes that are large enough to limit the function call overhead and small enough to fit in the CPU caches.

On GPUs, the compromise between tuple-at-a-time and full materialization strategies is not a sweet spot, however. Kernel invocations are an order of magnitude more expensive than CPU function calls. Furthermore, GPUs need much larger batch sizes to facilitate over-subscription and out-of-order execution. This leads to the problem that batches, which fit in the GPU caches, are too small to be processed efficiently. Alternatively, more recent GPUs support *pipes* to move a local execution context from one kernel to another. This has been used by GPL [25] for query processing. However, this technique still introduces an overhead for switching the execution context. In addition, it is limited to a depth of 2–32 kernels depending on the microarchitecture.

Query Compilation. Query compilation is a commonplace tool for avoiding excessive memory transfers during query processing. Compiling code for incoming queries becomes feasible with low-level code generation and achieves performance close to hand-written code. The compilation strategy of Neumann [23] keeps intermediate results in CPU registers and passes data between operators without accessing memory at all. The generated code processes full relations or blocks of tuples using a sequential tight loop.

To use query compilation on GPUs, we must integrate fine-grained data-parallelism into compiled queries. The parallelization strategy of HyPer [18], however, uses a coarse-grained approach, which allows it not to break with the concept of tight loops. In fact, HyPer does not use SIMD instructions [23] and thus omits fine-grained data-parallelism. Even

on CPUs with a moderate degree of parallelism in SIMD instructions, database researches are challenged with integrating query compilation and SIMD instructions [20, 30].

In summary, using a micro-level technique for efficient on-chip pipelining on GPUs remains a challenge. Applying any of the commonplace techniques makes it necessary to combine at least three things that are hardly compatible: fine-grained data-parallel processing, extensive out-of-order execution, and deep operator pipelines. To achieve our goal of mitigating the GPU global memory bottleneck, we need to develop a new micro execution model which we build up step by step in the following sections.

4 DATA-PARALLEL QUERY COMPILATION

In the following, we show a micro-level execution strategy that reduces GPU global memory access volumes by means of pipelining in on-chip memory. To this end, we show the approach of our query compiler HorseQC and its integration with the operator-at-a-time execution engine of CoGaDB [6].

4.1 Fusion Operators

HorseQC extends the operator-at-a-time approach with the concept of *fusion operators*, operators that embrace multiple relational operations. A fusion operator replaces a sequence of conventional operators in the physical execution plan with a micro-level-optimized pipeline. The data movement within a fusion operator can be improved by applying different micro level execution models.

4.2 Micro-Level Pipeline Layout

To keep matters simple, we first apply query compilation with the operator-at-a-time primitives described by He et al. [11]. This choice is not limiting as other data-parallel primitives may be used instead. However, a commonality of different primitive sets is that they use *relational primitives* with relational functionality (e.g. select) and *threading primitives* with thread coordination functionality (e.g. map, prefix sum, gather).

State-Of-The-Art. We look at a query with two input tables and a total of four relational operators $op_1, \cdots, op_4$. Operator-at-a-time runs three primitives per operator (cf. Figure 6): The first pass executes the relational primitive (e.g. select, project) and counts the number of outputs of each thread. The second pass computes a *prefix sum* to obtain unique per-thread write positions. The third pass performs an *aligned write*. This means that the output values are written into a dense array and may include executing the relational primitive for a second time to
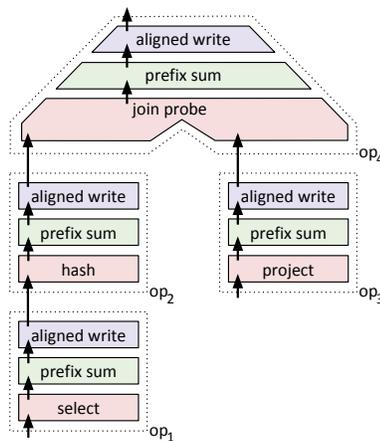

Figure 6: Operator-at-a-time

Figure 8: Transforming data-parallel operator-at-a-time into compiled execution. The functionality of each operator maps to designated positions in the generated kernels.

Figure 9: Data movement for data-parallel query compilation with three phases.

produce the output values. Thus, the query is processed in twelve operations with separate GPU global memory I/O.

Multi-Pass Query Compilation. By grouping operations that are applied to the same input table, the query may be processed with two fusion operators. Within each fusion operator,

Figure 7: Multi-pass QC

we apply the following query compilation strategy (cf. Figure 7): We extract the prefix sum from the operators and execute it only once between all relational primitives and all aligned writes. The relational primitives are then compiled into one kernel called count, which is executed before the prefix sum. The aligned writes are compiled into one kernel called write, which is executed after the prefix sum. In this way, we apply *kernel fusion* [31] to the four relational primitives and to the four aligned writes. The same query is processed with six operations and the operations in compiled kernels communicate through on-chip memory instead of GPU global memory.

4.3 Instancing Relational Code Templates

We briefly describe the process used by HorseQC to compile OpenCL code for the count and write kernels by an example of the projection operation (similar to [5]). Each primitive, except for prefix sum, is mapped to a designated position in

the count kernel or in the write kernel (cf. Figure 8). The query compiler receives a C++ object that describes the primitive's functionality (e.g. a tree for an arithmetic expression) and maps the semantics to fragments of OpenCL. To illustrate, $\pi_{\text{revenue}\leftarrow\text{price*discount}}$ would compile to

```
revenue[wp] = price[tid] * discount[tid];.
```

The global index tid is used to access the input columns and the write position wp is used for the output columns.

The instantiated code is placed in a code frame, which has several invariant features, e.g., thread offset computations, a surrounding loop, as well as managed features such as a parameter list. Projection is positioned in a conditional clause of the write kernel that is entered by all threads with a positive is_selected flag (cf. Figure 8). Other operations may include function calls for reductions or hash table operations.

4.4 Memory Access and Limitations

In Figure 9, we illustrate the bandwidth characteristics of our example query when using code generation with three phases. The figure shows the behavior of the three-phase micro execution model described above with the batch processing macro execution model. To analyze the implications of forwarding intermediate results in the generated kernels through registers and scratchpad memory, we extended the illustration with an additional GPU-internal layer of memory.

GPU global memory access has previously been the bottleneck for query execution. Here the count kernel accesses 1.7 GB in GPU global memory, the prefix sum computation 0.8 GB, and the write kernel 1.9 GB respectively. This is a reduction by factor **1.9x** compared to batch processing. In the generated kernels, a substantial amount of memory traffic

has moved to on-chip memory. In on-chip memory, the access volume of 14.4 GB is not a limiting factor due to the extremely high bandwidth of 1.2 TB/s of scratchpad memory.

Although the reduced GPU global memory traffic may suggest that the approach eliminates the bottleneck, real world queries still experience limitations. In fact, Section 8.6 shows that compilation with three phases can still not saturate PCIe for 9 out of 12 SSB queries. This is because the query complexity prevents the strategy from utilizing the full GPU global memory bandwidth. Therefore, we investigate ways to further increase the processing efficiency in the next section.

5 PROCESSING PIPELINES IN ONE PASS

The previous execution model relied on a typical programming concept of GPUs that executes operations with multiple kernels. The kernels that execute the actual work for the operation are interleaved with kernels that execute prefix sum computations. To further improve the processing efficiency, we have to break with this concept. With a new micro execution model, we avoid round trips to GPU global memory, which are caused by multi-pass implementations. This enables us to radically reduce GPU global memory traffic and lift the bandwidth bottleneck.

Compound Kernel. Kernel fusion brought reduction operations (e.g. prefix sum) as boundaries into the spotlight.
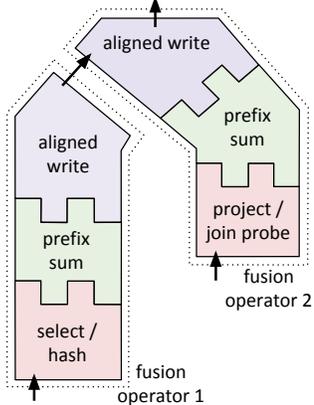
Figure 10: Compound kernel

Previously, we computed the prefix sum *between* two generated kernels to obtain write positions. Instead of two separate kernels, we now generate only one *compound kernel* that integrates the prefix sum computation (cf. Figure 10) and this eliminates multiple passes. Computing write positions *within* a generated kernel makes it possible to process pipelines in one pass without intermediate materialization. In this way, each fusion operator is executed by a single compound kernel. In the following, we look at implementation strategies for reduction operations that enable fully pipelined processing.

5.1 Pipelining Data-Parallel Reductions

Reductions are a poster child for data-parallel algorithms [14] and have been investigated in detail regarding complexity, efficient implementations, and their applications. In the context of database systems, they are especially relevant in the context of prefix sums [2, 8] and aggregations. The latter involves two techniques: Simple reductions aggregate to a single tuple and segmented reductions compute grouped aggregates on sorted data [29]. As reductions have inherent parallel dependencies, they are typically implemented in a hierarchical structure that involves running multiple kernels in sequence. This approach is applied in state-of-the-art coprocessor database systems such
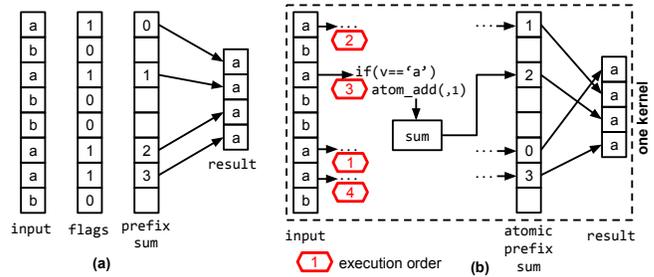

Figure 11: The computation of a prefix sum for writing selected elements to a dense array (a) can be parallelized using atomic operations (b).

as Ocelot [12], CoGaDB [6], GPUDB [36], Kernel Weaver [32] and Voodoo [26].

Atomic Prefix Sum. The separation into multiple reduction kernels with intermediate materialization is an obstruction for pipelining. To introduce a pipelined implementation, let us look at a very simple sequential prefix sum at first:

```
for(i=0; i<n; i++)
    if(flags[i]) prefix_sum[i] = sum++;
```

The sequential prefix sum loops through the array `flags` while writing *and* incrementing `sum` for every valid entry. Figure 11a illustrates the use of the prefix sum for a dense write of selected input elements. When parallelizing the for-loop, this implementation runs into the issue of many threads trying to increment `sum` at the same time. To resolve this parallel dependency, atomic operations can be used to isolate parallel modifications of the same memory address. Atomic operations ensure a consistent state, yet are executed in an undefined order. The following code executes an *atomic prefix sum* to compute unordered, dense write positions:

```
if(is_selected) wp = atom_add(&sum, 1);
```

Threads contribute an offset of 1 to the sum at address `&sum` by executing the expression conditionally. Each `atomic_add(..)` returns the previous state of `sum`. Thus, threads immediately obtain a unique global write offset as `wp` in register. This is illustrated in Figure 11b.

The use of atomic operations causes a break with the semantic of the prefix sum because the result has *no defined order*. For the relational semantic, however, only the *uniqueness* of output positions is critical. Output permutations lead to non-aligned GPU global memory access where adjacent threads do not write to adjacent memory addresses. The impact on write throughput, however, is limited, because the filter semantics lead to non-aligned access for separate prefix sums too.

5.2 Code Generation for Compound Kernels

Computing write positions within a generated kernel allows us to contract the three phases within a fusion operator into one *compound kernel*. This simplifies code generation for two reasons (cf. Figures 8 and 12): First, selection flags and write offsets remain in registers and do not have to be passed

```
compound_kernel( ... ) {
  int tid = get_thread_offset();
  // select
  ...
  // join probe
  ...

  //atomic prefix sum
  if(is_selected)
    wp = atom_add(&sum, 1);

  if(is_selected) {
    // project/write
    ...
  }
}
```

Figure 12: The compound kernel integrates all three pipeline phases into one kernel.

Figure 13: Data movement for query compilation with one pass. The compound kernel reduces data movement by 4.7x.

between kernels through materialization. Second, relational primitives that occur both in the `count` and in the `write` kernel are executed only once in the compound kernel, e.g., we probe the hash table to check the number of matches and keep the payload in registers for projection. This becomes possible in the compound kernel as the register content remains valid until projection. In Appendix E, we show the full code for an exemplary query.

To instantiate relational primitives, we follow a similar procedure as previously described, but now we use only one kernel code frame: All relational primitives that affect the number of outputs are placed before the atomic prefix sum and all relational primitives that produce output after it. The atomic prefix sum is instantiated from an invariant code template that takes the `is_selected` flag as input and assigns the write position `wp` as output. Both the input flag and the write position are available in registers.

5.3 Memory Access and Limitations

The compound kernel micro execution model further reduces GPU global memory access by a factor of 2.4x to 1.8 GB (see Figures 9 and 13). Compared to operator-at-a-time, this is a reduction by a factor of **4.7x**. Pipelining the prefix sum avoids round trips to GPU global memory that are necessary in the three-phase micro execution model. The compound kernel has only a minimal GPU global memory access volume for input, output and hash table access. Now the on-chip traffic is balanced with the GPU global memory traffic when relating each memory volume to the available bandwidth.

The described approach heavily relies on atomic operations. This has the disadvantage to cause limitations for parallelism. Although the execution order is undefined, the operations *are* sequentialized and reducing n values takes O(n) parallel steps. However, Egielski et al. [7] show that recent hardware support makes atomic operations competitive to parallel algorithms. Still, the integrated prefix sum puts a significant pressure on the atomic functional units, which prevents pipeline kernels
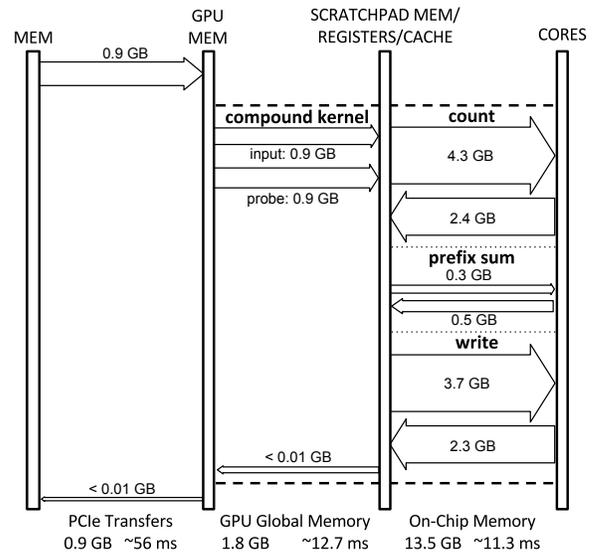
from utilizing full GPU global memory bandwidth. In the following, we address this issue and show how the efficiency of parallel reductions in compound kernels can be increased.

6 EFFICIENT PIPELINED REDUCTIONS

Previously, we showed a way to pipeline reductions in generated kernels using atomic operations. This benefits the memory efficiency, but at the same time exposes the atomic functional units of a GPU as the bottleneck. This is especially critical because several operations that are combined in the compound kernel rely on atomic isolation as well, i.e., state-of-the-art implementations of hash joins and hash aggregations [16] use atomic operations to isolate hash table inserts.

This section addresses performance bottlenecks that occur when utilizing atomic reductions to pipeline relational operators. We show a new technique *local resolution, global propagation*, that is used by HorseQC to pipeline prefix sums, single tuple aggregation and grouped aggregation efficiently. The approach reduces the pressure on atomic functional units and offers tunability regarding hardware and thread group granularity. We describe the approach in the following.

6.1 Local Resolution, Global Propagation

Similar to other efficient GPU implementations such as in CUB [21], local resolution with global propagation consists of two levels of reductions. In contrast to other techniques, local resolution, global propagation always uses pipelined techniques on *both* levels. Local resolution is an additional pre-reduction step, computed by a local thread group, whereas global propagation is the same atomic reduction as described in Section 5. We use the term *Collaborative Thread Array* (CTA) for the thread groups in local resolution. CTAs can either
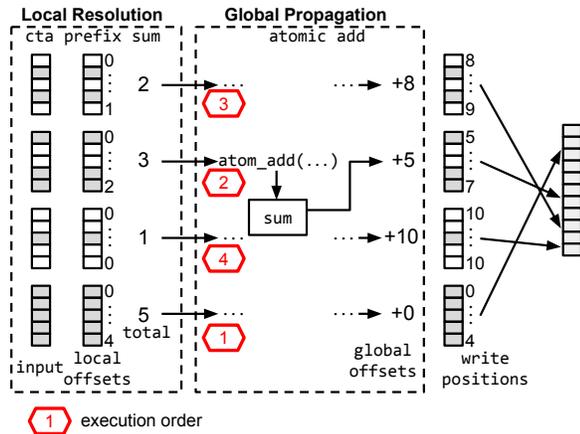
Figure 14: Computing write positions with local resolution (local offset), global propagation (global offset).

Figure 15: Local resolution mechanisms: (a) Work-efficient reduction (b) SIMD reduction (c) segmented reduction.

match the workgroup (AMD) or thread-block (NVIDIA) size of the GPU kernel or work on a finer granularity.

The following code, illustrated by Figure 14, executes an atomic prefix sum using local resolution, global propagation:

```
l_os = cta_prfx(flags, &cta_total);  //local res.
if(cta_thread_idx == 0)
  g_os = atom_add(&sum, cta_total);  //global prop.
wp = l_os + g_os;
```

First, each CTA executes `cta_prfx` to compute a local prefix sum on `flags`. This is the local resolution step. We implement `cta_prfx` with SIMD reductions (cf. Intra-Warp Scan Algorithm by Sengupta et al. [28]). The function returns the local offset `l_os` and the sum of all flags assigned to the CTA `cta_total`. Second, one thread of each CTA adds `cta_total` atomically to a global counter `sum`. This is the global propagation step. The call to `atom_add` returns the global offsets `g_os`. Finally, the write position `wp` is the sum of `l_os` and `g_os`.

Compared to the simple atomic prefix sum, we now add pre-aggregates instead of 1/0 flags to `sum`. Therefore, each atomic add obtains ranges of output indices instead of a single index. We make the analogy of *allocating* segments of output memory to CTAs. The order of the allocations however is undefined (see execution order in Figure 14). This leads to output that is ordered *within segments* and permuted *between segments*. Further investigation revealed that, due to the GPUs stream processing engine, the permutations exhibit locality, leading to semi-ordered output data.

Local Resolution Mechanisms. The mechanisms used for local resolution are interchangeable. This makes it possible to tune pipelined reductions and to apply them in different operations (cf. Appendix Section C for more details). Figure 15a and 15b show the integration of work-efficient reductions [3] and SIMD reductions [28]. Both techniques have different thread group granularities and we can choose between them to adapt to the hardware parallelism of different processors. Figure 15c shows the use of pipelined segmented reductions for grouping. First, segmented reductions compute pre-aggregates
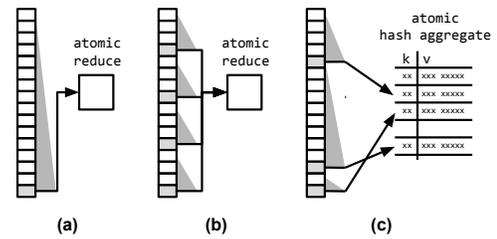
in scratchpad memory. Second, global propagation inserts the pre-aggregates into a hash table with an atomic operation. The ability to control scratchpad memory opens up a new design space for grouping algorithms in pipelined computations (e.g. handling frequent items). A similar approach PLAT [35] aggregates frequent grouping keys in a table local to each CPU core.

7 DBMS INTEGRATION

We integrated our query compiler HORSEQC into the open source DBMS CoGaDB, leveraging the built-in code generator *Hawk* [5]. The DBMS uses a columnar data layout and processes full columns operator-at-a-time on GPUs and CPUs. We use the front-end and the storage layer of CoGaDB, and HORSEQC adds a compiler-based execution engine.

We added two components to the DBMS: 1. a query compiler that compiles fusion operators to GPU code (cf. Section 4) and 2. a *translation layer* that identifies fusion operators and drives the query compiler. Currently, there are two different workflows for the translation layer:

(1) CoGaDB parses the SQL code for a query and generates a query plan. The translation layer applies the produce/consume model [23] to the query plan to determine fusion operators. We use this approach for the SSB queries and TPC-H Q6.

(2) The translation layer parses a JSON file that describes the query plan including the fusion operators. This enables us to process queries when (1) cannot handle the queries via SQL (e.g. correlated subqueries or automatic unnesting). This is used for the other TPC-H queries.

When the fusion operators are defined, the translation layer drives the query compiler to compile and execute. Finally, decompression of dictionary compressed columns and sorting are executed by CoGaDB's original execution engine.

8 EVALUATION

Section 2.1 showed that query coprocessing in existing macro execution models is sensitive to memory bandwidth bottlenecks on various hierarchical levels. We proposed several micro execution models that allow to remove memory indirections to achieve a more efficient use of bandwidth. In this section, we evaluate our approaches and carefully assess bandwidth and throughput to show several benefits.

The experimental study is structured as follows: First, we evaluate the *micro execution models*. Therefore, we execute specific queries to analyze the *reduction performance* of the proposed techniques in Experiments 1 and 2. Then, we evaluate the micro execution models for the SSB and TPC-H benchmarks in Experiments 3 and 4. Second, we analyze the *integration* of our micro execution model with the batch processing *macro execution model*. Therefore, we analyze the *real-world benefits* of our approach with a comparison of end-to-end performance in Experiment 5 and a scalability analysis in Experiment 6. Note that all experiments, except for Experiment 6, were executed with scale factor 10.

8.1 Processing Techniques

There are three micro execution models in HᴏʀꜱᴇQC that result from the paper. Table 4 in the appendix provides implementation details. The goal of our micro execution models is to use them within macro execution models to improve performance. Therefore, it is crucial to achieve a higher throughput than PCIe when executing queries. We show the benefit of our approaches by comparing them to an operator-at-a-time micro execution model. In this way, we analyze the benefit of moving data transfers between relational operators to the on-chip level.

Multi-pass The first approach separates reductions from the generated kernels, which leads to an execution in multiple passes (Section 4). Each reduction is executed on materialized data using the `boost::compute` library.

Pipelined The second approach integrates reductions into a fully pipelined kernel using atomic operations (Section 5). By using atomic operations for each reduction input, the approach is an instance of local resolution, global propagation that has no local resolution step.

Resolution The third approach increases the efficiency of pipelined reductions with local resolution methods like pre-aggregation (Section 6). We differentiate between local resolution implementations using Resolution:SIMD for SIMD reductions and Resolution:WE for work-efficient reductions.

Operator-at-a-time We use CoGaDB 0.4.1, which processes full columns of data in each operator with CUDA kernels. It features a run-to-finish macro execution model and an operator-at-a-time micro execution model.

8.2 Baselines

PCIe transfer The *PCIe transfer time* for transferring input and output data between the host's main-memory and GPU global memory. It is the target time for micro execution models for balancing throughput and PCIe bandwidth. The PCIe transfer time is shown in each graph with a dashed line (- - -).

Memory bound The *GPU global memory bound execution time* is the time for accessing the data. As each approach has to read the input columns and write the output columns, the baseline is a lower bound on the kernel execution time. We indicate it with a solid line (——) in each graph.

Model	Type	Archi-tecture	Cores	Scratch pad (KB)	B/W (GB/s)
GTX970 (NV)	GPU	Maxwell	13	96	146.1
GTX770 (NV)	GPU	Kepler	8	48	167.6
RX480 (AMD)	GPU	Ellesmere	32	32	104.9
A10 (AMD)	APU	Godavari	8	32	18.7

Table 2: Coprocessors used in the evaluation.

```
select lo_extprice * lo_discount + lo_tax as revenue
from   lineorder
where  lo_quantity between 25 - x and 25 + x
```

Figure 16: Query 1 is a simple selection and projection query inspired by the star schema benchmark.

8.3 System Configuration

For the experiments, we use three dedicated GPUs with PCIe gen 3.0 links and one APU that accesses main-memory directly. Table 2 specifies the GPU models and shows hardware properties. The amount of scratchpad is available *per core*. The reported bandwidth refers to GPU global memory for the GPUs and to main-memory for the APU. It was measured using on-GPU `memcpy` of 1 GB data. We measured bidirectional PCIe transfers between CPU and GPU as 12.1 GB/s.

Both NVIDIA GPUs GTX770 and GTX970 run in a system with an Intel Xeon E5-1607 CPU. We use the NVIDIA 364.19 driver and CUDA Toolkit 7.5 with OpenCL drivers. The AMD RX480 GPU is placed in a separate system with the A10-7890K APU. We use the AMDGPU-Pro 16.40 driver for the GPU and the fglrx 15.201 driver for the APU. Each system is running Ubuntu 14.04 and uses the boost library 1.61.

We used the profiling tools `nvprof 2.0.28` for NVIDIA hardware and `CodeXLGpuProfiler V4.0.511` for AMD hardware to measure kernel execution times, PCIe transfers, and GPU global memory access. For the measurements of kernel execution times, we used both tools to profile individual kernels and sum up the kernel execution times if multiple kernels are involved.

8.4 Experiment 1: Pipelined Prefix Sum

We compare several pipelined prefix sum techniques to one non-pipelined technique for a query that filters and projects one table. This allows us to analyze the benefit of integrating prefix sum computations into single-pass kernels. We execute Query 1, shown in Figure 16, and vary the selectivity in the range $[0, 1]$ using `x`. By running the experiment on four GPUs, we aim to assess the best local resolution mechanisms for a given hardware. Figure 17 shows the results.

Observations. Pipelined techniques perform better than Multi-pass in most cases. Integrating the prefix sum computation into single-pass kernels reduces the kernel execution times by factors up to **6.3x**. While processing with Multi-pass takes up to 328.6% of the PCIe time, Resolution:SIMD uses only 101.3% of the PCIe time in the worst case (selectivity
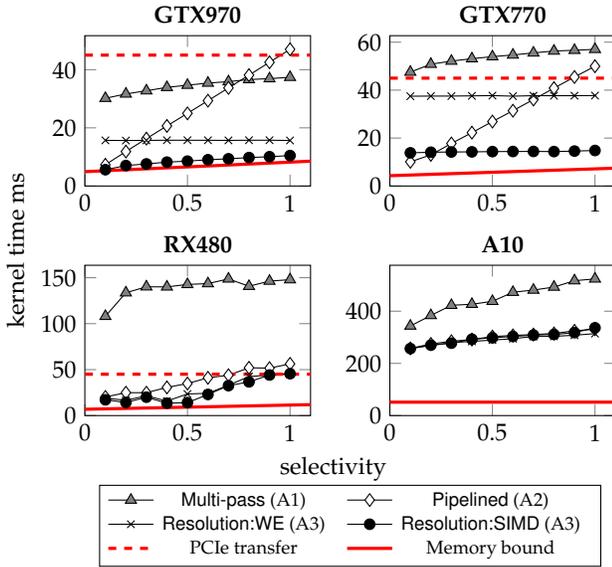
Figure 17: Projection query executed with different approaches. Integrating prefix sums into kernels allows fastest execution.

1.0, RX480). This shows that the approach can saturate the bus bandwidth for a variety of configurations. On the A10 there are no PCIe transfers and Resolution:SIMD increases the overall throughput by factors of up to **1.6x** over Multi-pass.

The results show that the local resolution step reduces the performance impact of atomic operations. This becomes visible for higher selectivity factors: Pipelined has higher executions times because the strategy executes one atomic addition per output. Resolution:SIMD and Resolution:WE however show good performance across all selectivities due to local resolution.

Resolution:SIMD achieves the shortest kernel execution times in most cases and allows memory bound processing on the GTX970. On the GTX770, lowering the output size down to 0 does not affect the execution time. We conclude that the GTX770 is compute-bound earlier than the GTX970. The higher memory bandwidth of the GTX770 leads to an increased throughput for atomic operations and Pipelined can outperform Resolution:SIMD for selectivities below 10%. On the RX480 and on the A10 there is no definite advantage for one of the reduction techniques. In the following, we only use Resolution:SIMD and skip the other techniques for a clear presentation.

8.5 Experiment 2: Pipelined Group By

We evaluate the effect of pipelined `GROUP BY` aggregations using different techniques. We execute Query 3 (shown in Figure 26) with Operator-at-a-time, Pipelined and Resolution. The query groups all tuples of `lineorder` according to the computed attribute `lo_orderkey%x` into sums. We vary the number of groups by increasing `x` from 2 to 16384. We show the results of the experiment on a GTX970 GPU in Figure 18.
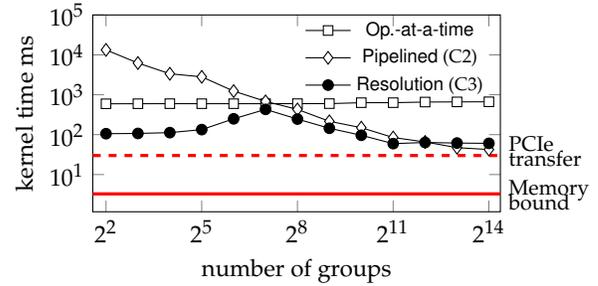

Figure 18: Performance of grouped aggregations.

Observations. The execution times of Operator-at-a-time do not depend on the group size. The main cost factor is sorting the input columns. Pipelined shows up to **11.1x** lower execution times but only for larger group sizes. For group sizes below 64, we observe high execution times. This is caused by heavy contention of parallel aggregation hash table inserts.

The bottleneck is resolved by Resolution which uses pre-aggregations to reduce the contention. The results show that execution times reduce by factors of up to **126x**. However, the local pre-aggregations have a limited effect on larger group numbers. This explains the spike at 128 groups, where both pre-aggregation and contention have an effect. While the approaches cannot saturate PCIe when aggregating a full table, filters reduce the cost of grouping for real-world queries.

8.6 Experiment 3: Star Schema Benchmark

The previous experiments showed that pipelining specific reduction operations helps to increase the throughput of query processing. In this experiment, we analyze whether this behavior carries over to real-world situations. To this end, we execute the SSB Queries[4] on the GTX970 GPU (other coprocessors in Appendix Section G.2).

We use Operator-at-a-time and two variants of HorseQC. HorseQC: Multi-pass uses pipeline breaking implementations for reductions (A1, B1 and C1). HorseQC: Fully pipelined integrates all pipeline operations in one kernel (using A3, B3 and C2). We show the results of the experiment in Figure 19.

Observations. The bandwidth analysis in Section 2.1 showed that 4 out of 12 queries are limited by GPU global memory access in operator-at-a-time processing.

- The kernel execution times of Operator-at-a-time show that compute and latencies further increase the problem. While PCIe would allow execution times between 60.6*ms* to 90.9*ms*, the kernel execution times take longer for 10 out 12 queries with up to 295.5%.
- HorseQC: Multi-pass improves over Operator-at-a-time and uses only 50.5% of the PCIe bandwidth transfer time in the best case and 215.5% in the worst case. This shows that without efficient pipelining of reduction operations, the benefit of query compilation is limited.

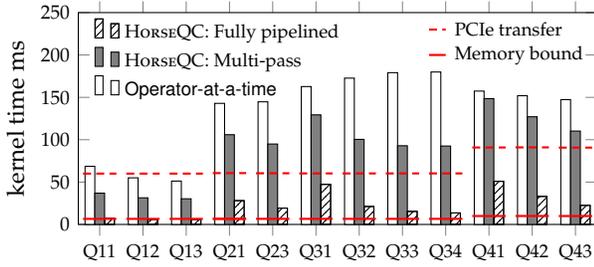[4]We could not process SSB Query 2.2 as we do not support range predicates on dictionary compressed columns yet.

Figure 19: Performance of SSB queries.

Figure 20: Performance of TPC-H queries.

Figure 21: End-to-end performance of star join computation for different scale factors.

- HORSEQC: Fully pipelined lowers all kernel execution times to a level that is consistently lower than PCIe transfer times. This shows that compiling pipelines into one kernel with local resolution, global propagation provides an execution approach with sufficient throughput. Processing takes 9.7% of the PCIe transfer time in the best case and 78.1% in the worst case. For Queries 1.1, 1.2 and 1.3 kernel execution is memory bound by GPU global memory access.

8.7 Experiment 4: TPC-H Queries

We execute and profile queries from the TPC-H benchmark [1] to show the effect when relaxing the specific assumptions of the star schema benchmark (e.g. using one centralized table). We select a subset of queries based on the work by Boncz et al. [4] to capture challenging aspects of the TPC-H benchmark, i.e., Q1, Q4, Q13, and Q21 contain heavy aggregation, Q9, and Q18 contain heavy joins, and Q4, Q19, and Q21 contain parallelism bottlenecks. We modified 4 queries, because HORSEQC currently does not support all operations, e.g., `like` expressions (cf. Appendix Section F for details). The results of the experiment are shown in Figure 20. For Q1, there is no result for HORSEQC: Multi-pass, because the strategy ran out of GPU memory. The results shown for Operator-at-a-time are for all TPC-H queries supported by the DBMS.

Observations. The PCIe and memory bound baselines show larger variations than for the SSB benchmark. This is mainly caused by the join structure, e.g., Q13 joins three small tables, while Q17, Q18, and Q21 join multiple instances of the largest `lineitem` table.

The kernel execution times show that HORSEQC can improve over operator-at-a-time by factors of up to **8.6x**. For Q1, Q4, and Q9, there are cases where Operator-at-a-time has shorter kernel execution times than compiled strategies. Further investigation showed that in these cases Operator-at-a-time moves some operators to the CPU, therefore the measurements cover a limited amount of operations.

Comparing the variants of the query compiler, we observe that HORSEQC: Fully pipelined consistently improves over HORSEQC: Multi-pass by factors of up to **5.4x**. HORSEQC: Fully pipelined achieves lower execution times than PCIe transfer times for 8 out of 11 queries. For Q1, Q13, and Q18 the PCIe bandwidth cannot be fully saturated. This is because the queries contain grouped aggregations of unfiltered columns
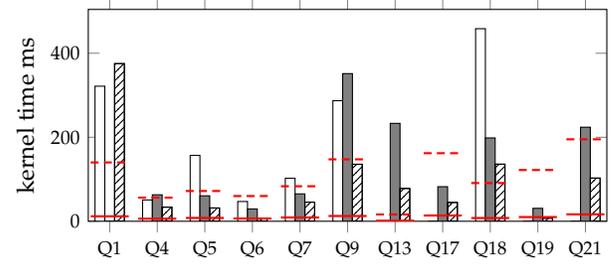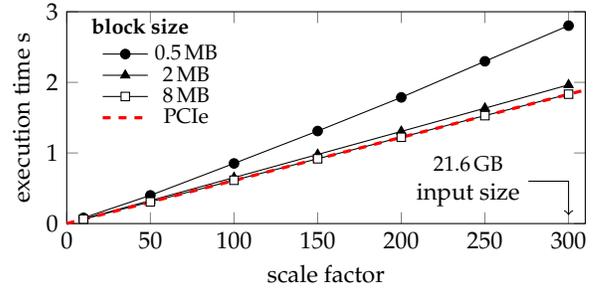
(cf. Experiment 2). The execution times of HORSEQC: Fully pipelined take 5.6% of the PCIe transfer time in the best case and 268.1% in the worst case.

8.8 Experiment 5: Scalability

Due to the deeply integrated storage layer implementations of the host DBMS CoGaDB, we were not able to build a fully scalable version of HORSEQC. For this reason, we perform a separate experiment that integrates the Resolution micro execution model with the batch processing macro execution model for the star join from SSB Query 3.1. Decoupling this experiment allows us to apply the rules for coprocessor data management by Yuan et al. [36] and to measure end-to-end performance for larger datasets.

The star join recombines three dimension tables and one fact table with an overall selectivity of 3.4%. We build hash tables for the dimension tables in GPU global memory. The fact table resides in pinned host memory and each column is partitioned into blocks of 0.5 MB, 2 MB or 8 MB. The blocks are transferred asynchronously via PCIe into an inner kernel that computes the star join by probing each dimension hash table.

Figure 21 shows the end-to-end execution times for each block size when executing the experiment. We observe that execution times grow linearly with increasing scale factors and that block sizes larger than 2 MB can saturate the PCIe bandwidth. The computation does not become a bottleneck for the examined scale factors. With a block size of 4 MB and scale factor 300, the size of intermediate data in GPU global memory is only 473 MB. Therefore, we expect the approach to scale to even larger databases with linear performance.

8.9 Experiment 6: End-to-End Performance

To make a comparison to other database systems, we execute the TPC-H queries with different database systems and measure end-to-end performance. We compare MonetDB5 Dec2016-SP3 executed on CPUs, and CoGaDB 0.41 and HorseQC executed on GPUs. Both competitors feature an operator-at-a-time approach. We perform the measurements with warm caches. MonetDB runs on a workstation-class system with an Intel Xeon E5-1607 CPU and 32 GB RAM. CoGaDB and HorseQC run on the GTX970. The results are shown in Figure 22.

Observations. For the supported queries, HorseQC is up to **5.8x** faster than CoGaDB. While CoGaDB uses GPU global memory as a cache for frequently used columns, HorseQC does not cache data between queries. This shows that HorseQC uses memory and interconnects more efficiently. For Q6 there is no improvement, because query execution is PCIe bound.

HorseQC has lower execution times than MonetDB by factors of up to **26.9x**. Despite moving data through the PCIe bottleneck, the additional bandwidth resources of GPU global memory offer an acceleration. For Q19 MonetDB has a lower execution time than HorseQC. This shows that for queries with a low complexity, it is more effective to process data directly than moving it over PCIe.

9 DISCUSSION

In the previous experiments, we evaluated our new approaches to query compilation on coprocessors. Across all experiments, we were able to show improvements of query compilation over operator-at-a-time processing. Operator-at-a-time has a low memory efficiency due to large materialization volumes and repetitive operations. The approach therefore cannot utilize the memory systems surrounding the coprocessor efficiently.

While naive compilation techniques increase the memory efficiency, reductions and prefix sums split operator pipelines into multiple passes. In this way, the approach inherits the drawbacks of operator-at-a-time. This becomes visible because kernel execution times frequently exceed PCIe transfer times.

The paper shows a query compilation technique that merges the operators of a pipeline into one compound kernel. When combined with efficient reduction techniques, the compound kernel achieves substantial advantages over other processing approaches. With upcoming OpenCAPI and NVLink interconnects, these improvements to GPU-local processing are essential to benefit from increased bandwidth of the new hardware. In the evaluation setting, the PCIe bandwidth can be saturated for all SSB queries. For the TPC-H benchmark, the approach improves over operator-at-a-time and naive compilation, but saturates PCIe only 8 out of 11 queries. We conclude that the compound kernel works particularly well with star join queries.

10 MORE RELATED WORK

Combining multiple kernels for query processing on GPUs has been used in related work. Wu et al. [32] analyze query
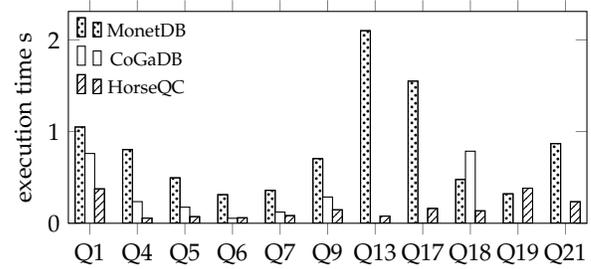

Figure 22: End-to-end performance of TPC-H queries.

plans to automatically fuse kernels with matching I/O data. Li et al. [19] use pre-fabricated kernels that recombine several operators.

Our approach to pipeline the computation of write positions produces data this is not strictly ordered but still contains locality. Such partially ordered data has been examined in the context of the Diag-Join by Helmer et al. [13].

Query compilation can be applied in higher-level languages for programmability [17] or in lower-level languages for low compilation times [23]. Similarly, on GPUs lower-level PTX or SPIR code may be used or higher-level languages may help to abstract hardware details.

With the end of frequency scaling, it has become increasingly important to exploit hardware parallelism. Power et al. [27] show that especially integrated GPUs can achieve better processing efficiency than CPUs.

In related work, two ways to compute single-pass prefix scans have been proposed. They are similar to local resolution, global propagation with different approaches to pipeline global propagation. First, in [34], Yan et al. serialize the computation of local prefix sums with memory barriers. Second, Merrill et al. [22] propose a dynamic look-back mechanism that recomputes unavailable partial sums. In contrast, we use atomic operations to avoid re-computations of long pipelines and to facilitate out-of-order execution.

11 SUMMARY

In this paper, we show query processing techniques that help to balance the data movement cost and the compute throughput on GPU-style coprocessors. We measure the data transfer volumes in different scalable processing approaches to assess bandwidth bottlenecks. While naive scalable execution techniques are limited by PCIe bandwidth, batch processing is limited by GPU-local throughput. To address the bottleneck, we propose micro execution models that benefit from on-chip pipelining. Naive query compilation techniques allow simple code generation but inherit the memory-intensity of operator-at-a-time. We introduce compound kernels that merge several pipeline phases into one efficient kernel.

REFERENCES

[1] Transaction Processing Performance Council. TPC Benchmark H. 2012.
[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.
[3] G. E. Blelloch. Prefix Sums and Their Applications. Technical report, Carnegie Mellon University, 1990.
[4] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.
[5] S. Breß et al. Generating custom code for efficient query execution on heterogeneous processors. *CoRR*, abs/1709.00700, 2017.
[6] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*, 2016.
[7] I. J. Egielski, J. Huang, and E. Z. Zhang. Massive Atomics for Massive Parallelism on GPUs. *ACM SIGPLAN Notices*, 49(11):93–103, 2015.
[8] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 328–335. IEEE, 1999.
[9] C. Gregg and K. Hazelwood. Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *ISPASS*, pages 134–144. IEEE, 2011.
[10] D. Harris. A Taxonomy of Parallel Prefix Networks. In *Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 2213–2217. IEEE, 2003.
[11] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *Transactions on Database Systems*, 34(4):21, 2009.
[12] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *PVLDB*, 6(9):709–720, 2013.
[13] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: An Opportunistic Join Algorithm for 1: N Relationships. 2004.
[14] W. D. Hillis and G. L. Steele Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.
[15] T. Karnagel, D. Habich, and W. Lehner. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB*, 10(7):733–744, 2017.
[16] T. Karnagel, R. Mueller, and G. M. Lohman. Optimizing GPU-Accelerated Group-By and Aggregation. In *ADMS@ VLDB*, pages 13–24, 2015.
[17] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.
[18] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*, pages 743–754. ACM, 2014.
[19] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *PVLDB*, 9(14):1647–1658, 2016.
[20] P. Menon, T. C. Mowry, A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, T. Mowry, M. Perron, et al. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1), 2017.
[21] D. Merrill. CUB v1.7.0: CUDA Unbound, a Library of Warp-Wide, Block-Wide, and Device-Wide GPU Parallel Primitives, 2017.
[22] D. Merrill and M. Garland. Single-Pass Parallel Prefix Scan with Decoupled Look-Back. Technical report, NVIDIA Corporation, 2016.
[23] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.
[24] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.
[25] J. Paul, J. He, and B. He. GPL: A GPU-Based Pipelined Query Processing Engine. In *SIGMOD*, pages 1935–1950. ACM, 2016.
[26] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.
[27] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood. Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-Aggregate Queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 11. ACM, 2015.
[28] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, (1):1–17, 2008.
[29] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Graphics hardware*, volume 2007, pages 97–106, 2007.
[30] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. Compilation in Query Execution. In *DaMoN*, pages 33–40. ACM, 2011.
[31] M. Wahib and N. Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE Press, 2014.
[32] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel Weaver: AUtomatically Fusing Database Primitives for Efficient GPU Computation. In *International Symposium on Microarchitecture*, pages 107–118. IEEE, 2012.
[33] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing Data Warehousing Applications for GPUS Using Kernel Fusion/Fission. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2433–2442. IEEE, 2012.
[34] S. Yan, G. Long, and Y. Zhang. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *SIGPLAN Notices*, volume 48, pages 229–238. ACM, 2013.
[35] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable Aggregation on Multicore Processors. In *DaMoN*, pages 1–9. ACM, 2011.
[36] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *PVLDB*, 6(10):817–828, 2013.
[37] K. Zhang, F. Chen, X. Ding, Y. Huai, R. Lee, T. Luo, K. Wang, Y. Yuan, and X. Zhang. Hetero-DB: Next Generation High-Performance Database Systems by Best Utilizing Heterogeneous Computing and Storage Resources. *Journal of Computer Science and Technology*, 30(4):657–678, 2015.
[38] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

A BASELINE EXPERIMENT

We executed Query 3.1 from the star schema benchmark (SSB) [24] and profiled several metrics with the `nvprof 2.0.28` tool. As hardware, we used an NVIDIA GTX970 with 146 GB/s memory bandwidth (measured using device to device `memcpy`) in a host system with 16 GB/s bidirectional PCIe bandwidth. As execution environment, we used the operator-at-a-time engine CoGaDB. CoGaDB works on full columns, therefore we chose a database with scale factor 10. This is a favorable case for a PCIe-attached system like CoGaDB: all intermediate results remain small enough to be kept in GPU global memory (4GB). Here and in all later configurations, we assume that input data resides in main-memory before query execution; results have to be moved back to main-memory afterwards. Between operators, CoGaDB keeps intermediate data in GPU global memory. We profiled the following metrics for the kernels and data transfer operations used to execute the query.

`dram_read_transactions`: Number of 32 byte read transactions between DRAM and L2 cache. The read metric indicates data volumes moved for kernel input and indirect reads (e.g. accessing input columns and probing).

`dram_write_transactions`: Number of 32 byte write transactions between DRAM and L2 cache. Write transactions include kernel output and indirect writes respectively (e.g. writing columns and scatter).

`PCIe Transfers`: Data transfer volumes between host and coprocessor through the PCIe bus. Transfers are profiled for each direction individually.

B SSB QUERY 3.1

We used Query 3.1 from the star schema benchmark several times to analyze the data movement performed by different processing approaches. We show the SQL code for the query in Figure 23.

```
select c_nation, s_nation, d_year, sum(lo_revenue)
as revenue from customer, lineorder, supplier, date
    where lo_custkey = c_custkey
    and lo_suppkey = s_suppkey
    and lo_orderdate = d_datekey
    and c_region = 'ASIA'  and s_region = 'ASIA'
    and d_year >= 1992 and d_year <= 1997
        group by c_nation, s_nation, d_year
        order by d_year asc, revenue desc
```
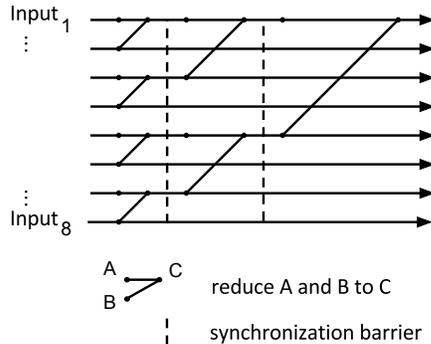
Figure 23: SSB Query 3.1.

Figure 24: Work-efficient reduction of 8 values.

C LOCAL RESOLUTION MECHANISMS

In this section, we show how different local resolution algorithms can be integrated with the atomic reduction of global propagation. Flexibly choosing local resolution algorithms enables us to tune reductions to the hardware and to realize regular as well as irregular data-parallel reductions. By embedding all local resolution techniques in one kernel with atomic global propagation all described operations remain fully pipelined. In the following, we first describe two regular data-parallel reductions with different thread group granularities for hardware tunability. Then we show how segmented reductions are embedded as pre-aggregation step for grouping.

Work-Efficient Reduction. Blelloch introduced an algorithm for work-efficient parallel reductions [3] that was adapted by Sengupta et al. [29] to GPUs with scratchpad memory. We illustrate a reduction of 8 values by 4 threads in Figure 24, which is inspired by the illustrations in [10]. The algorithm works in a tree-like structure that reduces 8 values to 4 in the first step, 4 values to 2 in the second and 2 to 1 respectively. Between each step, a synchronization barrier ensures that all results are ready. This produces one pre-aggregate per workgroup, which is passed to global propagation for atomic reduction. This is shown Figure 15 (a).

SIMD Reduction. The approach of Sengupta et al. [28] exploits the scheduling granularity of a GPU to reduce synchronization overhead. On GPU hardware, instructions are issued

in groups of 32/64 threads[5] in a SIMD fashion. By matching the reduction size with the instruction width, synchronization barriers become unnecessary within the reduction. This introduces an additional level of thread groups: E.g. a workgroup of 128 threads executes 4 reductions of 32 values each. Local resolution, global propagation adapts to this, by entering global propagation for each of the 4 pre-aggregates. This is illustrated in Figure 15 (b).

Segmented Reduction. Segmented reductions compute reductions of continuous segments within a sequence. They can be implemented as data-parallel algorithms on scratchpad memory with work-efficient reductions and SIMD reductions. This makes it possible to execute grouped pre-reductions in scratchpad memory. A similar approach PLAT [35] aggregates frequent grouping keys in a table local to each CPU core. For query compilation on GPUs the explicit control over scratchpad memory opens up a new design space for pipelined grouped aggregation algorithms. We implement a simple sort-merge-like approach that operates within CTAs on scratchpad memory:

```
sort_by_keys(keys, values);
reduce_segments(keys, values, head_flags);
if(head_flags[thread_idx])
    atomic_hash_reduce(ht, keys, values);
```

Additional to the reduced sequence, the segmented reduction outputs head flags. The head flags indicate the positions with finished pre-aggregates of a segment. Triggered by the head flags, threads enter global propagation and insert the pre-aggregates into a global hash table. This may happen in an irregular pattern as illustrated in Figure 15 (c).

D REDUCTION IMPLEMENTATIONS

We characterize three micro execution models according to the way they implement parallel reductions. Pipelined and Resolution implement reductions in the generated code; Multi-pass makes calls to library functions. While Pipelined implements reductions only with global atomic operations, Resolution uses an additional local resolution step. E.g. for technique C3, Resolution first performs a local sort, followed by a segmented reduce, and then aggregates the resulting pre-aggregates in a global hash table. We show the respective reduction implementations for each technique in Table 4. The implementations for global reduction and sorting operations use the `boost::compute` 1.61 library.

E KERNEL CODE

In Figure 25, we show the simplified code of a kernel that processes a simple query (Figure 16). The kernel includes a prefix sum computation, which allows to integrate both predicate evaluation and projection into one kernel. The code performs four steps:

(1) Evaluate predicates to count the number of results.
(2) Resolve local dependencies using `CTA_prefix_sum`.

---

[5]These thread groups are called wavefronts (64 threads) on AMD hardware and warps (32 threads) on NVIDIA hardware.

(3) Resolve global dependencies using `atomic_add` and share the global offset to each CTA thread to compute a write position.

(4) Compute projection and write the result.

```
void pipeline_kernel(__global int glob_sum, ...) {
  //1. predicate evaluation
  int num_out = 0;
  num_out += (lo_quantity[tid] >= 25-x
          && lo_quantity[tid] <= 25+x);
  //2. local resolution
  int local_offset = cta_prfx(num_out);
  //3. global propagation
  __shared int glob_offset[num_CTA];
  if(lid == CTA_limit) {
    glob_offset[CTA_idx] =
      atomic_add(&glob_sum, loc_offset);
  }
  int write_pos = loc_offset + glob_offset[CTA_idx]);
  //4. projection
  revenue[write_pos] =
    lo_extprice[tid]*lo_discount[tid]+lo_tax[tid];
}
```

Figure 25: Generated pipeline kernel for selection projection query containing the computation of global write positions.

F MODIFICATIONS TO TPC-H QUERIES

As our prototype implementation has a limited scope, we need to change some of the TPC-H queries. In particular, we currently do not support `"like"` expressions and anti joins. Note that there is no inherent limitation of supporting these features in future versions. We kept seven TPC-H queries (1, 4, 5, 6, 7, 18, 19) unchanged and modified four TPC-H queries (9, 13, 17, 21). Our modifications only marginally impact the performance of the queries. The changes are as follows:

Q9: We replaced the `"like"` expression `"p_name like '%green%'"` with a filter on the primary key p_partkey.

Q13: We removed the `"like"` expression `"o_comment not like '%special%requests%'"`.

Q17: We manually unnested the query.

Q21: We replaced the `"not exists"` expression on the second subquery with an `"exists"` expression because we do not support anti joins at the moment.

```
select sum(lo_extendedprice), lo_orderkey % x
from lineorder
group by lo_orderkey % x
```

Figure 26: Query 2 is a grouped aggregation of all `lineorder` tuples with `x` different groups.

G ADDITIONAL EXPERIMENTS

G.1 Single Tuple Aggregation

In this experiment, we evaluate the effect of pipelining single tuple aggregations. We modify Query 1 (shown in Figure 16) by adding an aggregation of the projected attribute `revenue` to a single sum. We compare Operator-at-a-time, Multi-pass and the pipelined technique Resolution. Figure 27 shows the results of Experiment 2 on all coprocessors. The experiment results confirm the basic observation that Resolution increases throughput over Multi-pass. The approach allows to saturate the PCIe bandwidth for all selectivities. Comparing the results of the GTX970 and the GTX770, we observe that atomic operations have gained performance on the newer Maxwell hardware architecture of the GTX970. However, compared to the results from Experiment 1, there is a clear difference in the performance of atomic operations. We attribute this to the reuse of the atomic aggregate value which is necessary for prefix sums but not required for aggregations.
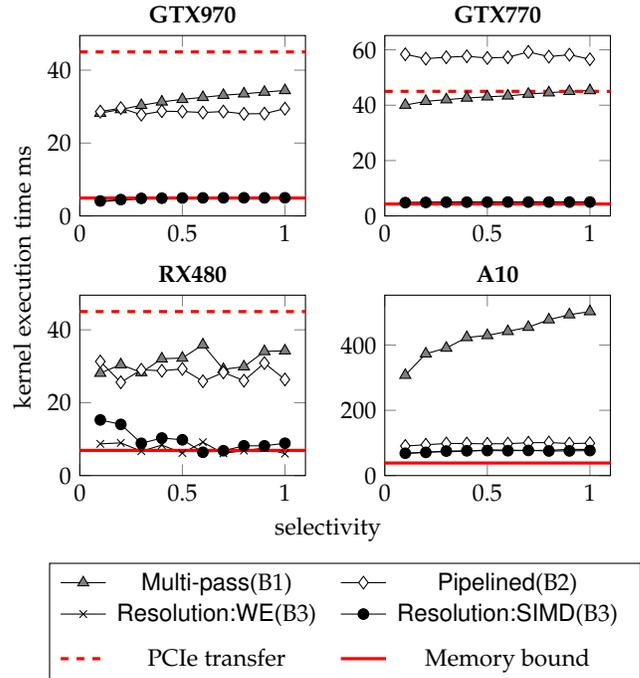

Figure 27: Performance of single tuple aggregation Query 2 across all coprocessors.

G.2 Star Schema Benchmark

In Table 3, we show the performance of our micro execution model Resolution:WE for the star schema benchmark queries across all coprocessors. We executed the queries on a database with scale factor 10 for the GTX970, GTX770, and RX480 and with scale factor 5 on the A10 due to the limited memory capacity of 2 GB. We add the measured throughput and memory bandwidth usage for each query. The results confirm the observation from the previous experiment. On the

Query	GTX970			GTX770			RX480			A10 APU (SF5)		
	time	thr.put	memory	time	thr.put	memory	time	thr.put	memory	time	thr.put	memory
	ms	GB/s	GB/s	ms	GB/s	GB/s	ms	GB/s	GB/s	ms	GB/s	GB/s
ssb11	7.20	133.33	116.58	8.14	117.97	103.15	31.79	30.19	26.40	61.94	7.75	7.61
ssb12	5.96	161.00	106.41	6.97	137.66	90.98	26.13	36.73	24.27	50.58	9.49	7.01
ssb13	5.93	161.95	106.67	6.67	143.82	94.73	27.81	34.52	22.74	51.75	9.27	6.85
ssb21	28.10	34.50	71.71	56.48	17.17	35.69	78.65	12.33	25.63	142.75	3.40	7.66
ssb23	19.31	50.06	77.04	37.62	25.68	39.53	60.80	15.89	24.46	112.94	4.28	7.11
ssb31	47.30	20.37	72.69	111.18	8.67	30.93	136.02	7.08	25.28	195.95	2.46	9.26
ssb32	21.20	45.45	38.52	39.89	24.16	20.48	84.29	11.43	9.69	101.53	4.75	4.17
ssb33	15.41	62.53	34.62	31.36	30.73	17.01	37.41	25.76	14.26	88.79	5.43	3.11
ssb34	13.56	71.06	38.31	29.26	32.93	17.75	30.67	31.42	16.94	82.35	5.85	3.32
ssb41	50.92	28.54	59.21	81.03	17.93	37.21	73.51	19.77	41.02	223.49	3.25	7.34
ssb42	33.10	43.97	61.01	59.26	24.55	34.07	97.69	14.90	20.67	151.85	4.79	7.30
ssb43	22.53	64.38	52.53	41.32	35.11	28.64	78.63	18.45	15.05	110.01	6.59	6.01

Table 3: Performance metrics of star schema benchmark queries across all coprocessors (scale factor 10, except for A10).

Operation	Local Resolution scratchpad (32-1024 elements)	Global Propagation global memory (all elements)	Pipeline Breaker	Materialization Volume		ID
Aligned	[global prefix sum]		yes	full	🐎	A1
Write	none	atomic prefix sum	no	none	⌂	A2
	prefix sum	atomic prefix sum	no	none	🖮	A3
Single Tuple	[global reduce]		yes	filtered	🐎	B1
Aggregation	none	atomic reduce	no	none	⌂	B2
	reduce	atomic reduce	no	none	🖮	B3
Grouped	[glob. sort, glob. reduce segments]		yes	filtered	🐎	C1
Aggregation	none	atomic hash reduce	no	none	⌂	C2
	sort, reduce seg.	atomic hash reduce	no	none	🖮	C3

🐎 Multi-pass	⌂ Pipelined	🖮 Resolution
Accepting pipeline breakers	Easiest way to fully pipeline	Best scratchpad usage

Table 4: Reduction techniques, that were introduced in this paper. HorseQC uses local resolution, global propagation to integrate reductions into fully pipelined kernels.

GTX970, the compute throughput consistently exceeds PCIe bandwidth, which allows a full utilization. On the other coprocessors, however, the throughput for some queries falls behind PCIe. Deeper investigation revealed that less-selective queries spend a substantial part of the time for computing the global propagation step of grouped aggregation. Improving the local resolution algorithm to use the full scratchpad memory space may benefit these cases.