Optimizing Group-By And Aggregation using GPU-CPU Co-Processing

Diego Tomé
CWI
Amsterdam, The Netherlands
diego.tome@cwi.nl

Tim Gubner
CWI
Amsterdam, The Netherlands
tim.gubner@cwi.nl

Mark Raasveldt*
CWI
Amsterdam, The Netherlands
m.raasveldt@cwi.nl

Eyal Rozenberg
CWI
Amsterdam, The Netherlands
e.rozenberg@cwi.nl

Peter Boncz
CWI
Amsterdam, The Netherlands
peter.boncz@cwi.nl

ABSTRACT

While GPU query processing is a well-studied area, real adoption is limited in practice as typically GPU execution is only significantly faster than CPU execution if the data resides in GPU memory, which limits scalability to small data scenarios where performance tends to be less critical. Another problem is that not all query code (e.g. UDFs) will realistically be able to run on GPUs. We therefore investigate CPU-GPU co-processing, where both the CPU and GPU are involved in evaluating the query in scenarios where the data does not fit in the GPU memory.

As we wish to deeply explore opportunities for optimizing execution speed, we narrow our focus further to a specific well-studied OLAP scenario, amenable to such co-processing, in the form of the TPC-H benchmark Query 1.

For this query, and at large scale factors, we are able to improve performance significantly over the state–of–the–art for GPU implementations; we present competitive performance of a GPU versus a state–of–the–art multi-core CPU baseline a novelty for data exceeding GPU memory size; and finally, we show that co-processing does provide significant additional speedup over any of the processors individually.

We achieve this performance improvement by utilizing parallelism-friendly compression to alleviate the PCIe transfer bottleneck, query-compilation-like fusion of the processing operations, and a simple yet effective scheduling mechanism. We hope that some of these features can inspire future work on GPU-focused and heterogeneous analytic DBMSes.

1. INTRODUCTION

General purpose computing on the Graphics Processing Unit (GPU) has become very popular in the past decade. Owing to the massively parallel architecture and the high memory bandwidth of both server-grade and consumer-grade GPUs, they can significantly outperform general-purpose CPUs on many tasks. GPUs have been effectively used to speed up many computation-intensive applications in the fields of scientific simulation, machine learning and data mining [21].

A significant body of work exists on exploiting GPUs to speed up analytic database query processing as well (see [8]). However, database workloads are often less compute-intensive and more data-intensive, and also require quite diverse computation. GPU database systems have mainly been limited to research and have not been adopted in practice. There are two main technical problems that stand in the way: (1) available GPU memory is very small (at most 24GB, whereas main memories can be TBs) compared to the large data sets in use today and transfer from main memory to GPU memory is slow, and (2) GPUs are limited in what operations they can perform efficiently compared to the much more flexible and generic CPU. For example, because each multiprocessor only has a single instruction decoder, all threads in a block have to execute the same instruction at the same time. As a result, algorithms that involve modestly complex control flow are not as efficient on a GPU as they are on a CPU – this includes for instance operations on variable-length strings. Given the importance of string processing in workloads [29], but also the use of UDFs and libraries written in CPU code, GPU-only database systems have little traction in practice, especially when their performance edge only surfaces when it is least needed: while processing small datasets that can be cached in GPU memory. Yet another problem are scheduling restrictions on GPUs that make concurrent execution of different queries on one CPU quite hard.

While a lot of GPU database processing research papers show order-of-magnitude performance gains when executing specific database operations inside the GPU, these results are obtained only when the input data is already resident in GPU Global memory [13]. Another issue often found in these work is that they compare a highly optimized implementation of the task at hand on the GPU to a straightforward baseline implementation on the CPU [13]. For many of these algorithms

1

the performance improvements tend to disappear when the cost of transferring the data over the PCIe bus is taken into account and when they would be compared against high-end CPU-based system such as e.g. Hyper. Finally, none of the GPU research systems supports the concurrent execution of different queries, while the ability to do so is required in most database workloads.

For these reasons, we focus on CPU-GPU **co-processing**, rather than GPU-only processing. Co-processing systems should ideally never be slower than a CPU-only system, because they can always choose to only execute on the CPU. The decision whether and how to use the GPU to perform operations, should be made automatically by such systems. However, such decisions are difficult to make a priori as these decision depends on many different factors, such as where the intermediate data resides and various characteristics of the input data, the size of intermediate structures [7], and the operations to be performed. Instead a dynamic approach that switches between which processor to use as the data is processed is desirable. The currently used processor could even be changed during the computation of an operation. For example when intermediates are too big for GPU main memory, the query engine can switch to using the CPU.

Further, we think that GPUs can only become useful in database processing if the state-of-the-art is improved in various dimensions. First is the deployment of GPU-friendly data compression methods to alleviate the PCIe transfer bottleneck, and to be able to fit more data in GPU memory. Second is compilation of (sub-)queries into custom GPU kernels that fuse together non-blocking relational operators as well as decompression. In addition more advances are needed in query processing methods, for instance for aggregation. Finally, we think co-processing systems need to develop intelligent division-of-work strategies between CPU and GPU that maximize the strength of each device and leaves work that involves its weaknesses to the other device.

The over-arching research question motivating our work is: what should CPU-GPU co-processing database architecture look like? In this paper, we make one small step into that direction by limiting the question to one well-known OLAP scenario, namely TPC-H Query 1 (Q1), which involves scan, selection, projection (expression calculation) and aggregation. We investigate how we can use a GPU to process this query, and what different implementation choices can be made along the way. We compare against our previous work on highly-optimized CPU implementations for this same scenario [15], to obtain an ambitious CPU baseline. All of the source code we have used for our experiments is publically available [25].

Contributions. The main contributions of this paper are as follows.

- We review current GPU-accelerated DBMSs and the related work that has been done on efficient group-by and aggregations on GPU.
- We explore the design space of efficient grouping and aggregation on compressed data on a GPU, and the division of work between CPU and GPU. For each aspect, we discuss the trade-offs that can be made and the performance benefits or drawbacks resulting from the decisions made.
- We perform an extensive performance evaluation which allows us to conclude that CPU-GPU co-processing can make significant performance gains, even given the relatively adverse characteristics of database workloads.

```sql
SELECT
  l_returnflag,
  l_linestatus,
  SUM(l_quantity),
  SUM(l_extendedprice),
  SUM(l_extendedprice * (1 - l_discount)),
  SUM(l_extendedprice * (1 - l_discount) *
    (1 + l_tax)),
  AVG(l_quantity),
  AVG(l_extendedprice),
  AVG(l_discount),
  COUNT(*)
FROM
  lineitem
WHERE
  l_shipdate <= date '1998-09-02'
GROUP BY
 l_returnflag, l_linestatus
ORDER BY
 l_returnflag, l_linestatus;
```

Listing 1: TPC-H Query 1 (with DELTA=90)

Outline. The remainder of this paper is organized as follows. In Section 2 we describe the architecture of GPUs and briefly discuss execution models for processing queries using them. In Section 3 we discuss the design space for processing TPC-H Q1-like group-by aggregation queries on the GPU, for using a CPU and aGPU together, co-processing the query. Experiments with points in this design space are described and discussed in Section 4. In Section 5 we discuss some related work, on the kind of GPU computation necessary for this query and the performance of more general GPU-utilizing query processors on TPC-H Q1. We conclude in Section 6, with a brief description of potential future work.

2. BACKGROUND

The architecture of a modern computer with a CPU and GPU as co-processor is depicted in Figure 1. The host system contains the CPU which is directly connected to the DRAM. The device system contains the GPU, and is connected to the host system through the PCIe bus. Any data that is transferred from the host system to the device system has to be transferred over the PCIe bus. The GPU itself is formed by a set of SMs that packages several scalar processors, an on-chip *Shared memory*, caches and a connector to the GPU *Global memory*. In the SM level, multiple threads are physically located in an execution unit called a *warp*. Every warp is accompanied by a single instruction decoder. In each cycle, the *warp* scheduler fetches instructions and executes instructions on all the threads located inside that warp, resulting in highly concurrent execution.

GPU Global Memory. The global memory has typically used DRAM modules, but with much better overall bandwidth than main system memory. Access to Global memory is paged, with a TLB being used. Access is also cached, in two levels: An L2 cache coherent across cores, and an L1 cache which is *not* kept coherent, unlike on a CPU. Also unlike a CPU, cache line lengths are 32B for L2 and 128B for L1; and the L1 cache is bypassed more often in accesses than on the CPU.

Shared Memory. Each stream multiprocessor has a dedicated region of high-bandwidth and low-latency shared mem-
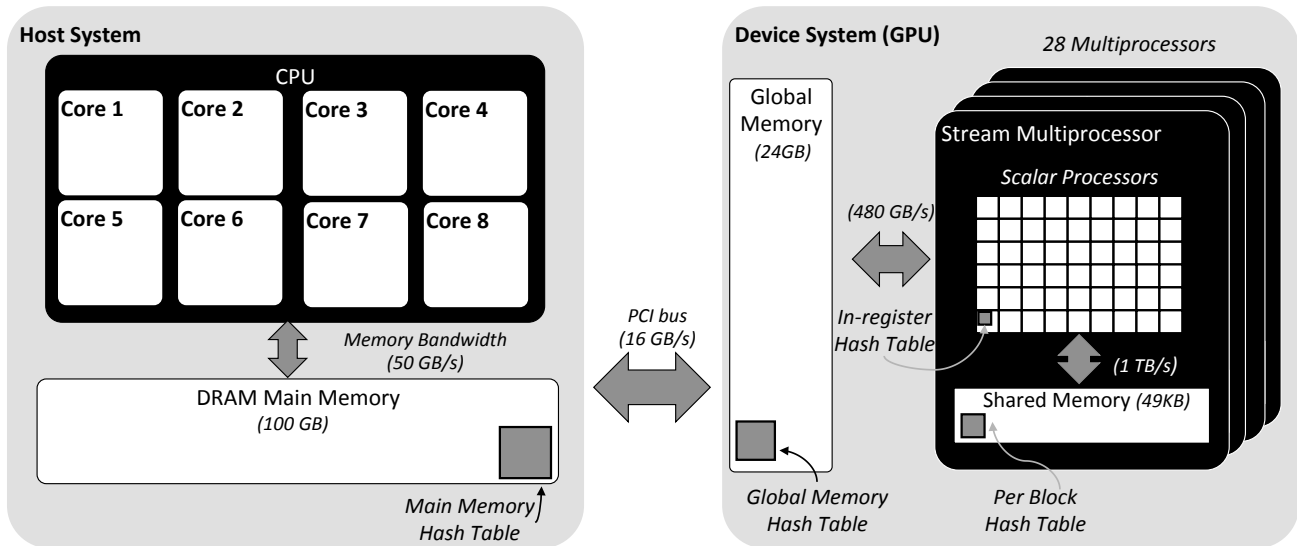
Figure 1: Architectures of a Host and a Device System with different placements for the Hash table

ory which is physically located together with the L1 cache. When running a kernel, the shared memory can be used to load and store data from global memory that will be accessed multiple times to reduce the amount of global memory accesses. The shared memory throughput is driven by two main factors: the amount of active warps per SM and the ILP [20].

3. DESIGN SPACE

In this section, we explore the design space of an optimized GPU implementation for grouping and aggregation. In particular, we focus our efforts on creating an optimized version of Query 1 of the well known TPC-H Benchmark [28]. This query is shown in Listing 1. We discuss the different implementation strategies that can be used implement group-by and aggregation on the GPU, and discuss how these strategies can be applied in the context of Q1. Our work intends to explore this design space using manually coded kernels, in order to identify opportunities and limitations. The final goal is to inform future CPU+GPU database architecture that will generate such kernels, which fuse operator pipelines and decompressing scans with Just-In-Time query compilation. Such compilation techniques are out of scope for this paper.

3.1 Compression

As mentioned earlier, a major hurdle for GPU effectiveness in query processing is its "underprivileged" access to data in main memory: It must receive all of its data at a third or quarter of the bandwidth at which data is available to the CPU. In absolute terms the figure is also underwhelming: Around 12 GB/sec in practice. Thus, the better optimized one can make the computation work on the GPU itself, the more it approaches the point at which the kernels' demand for unprocessed data saturates the bus.

Compression in analytic query processing has been relatively well studied; it is utilized in many modern columnar DBMSs to improve memory bandwidth and operate on compressed data [1]. As regarding GPU use, i.e. on-GPU decompression, only some initial exploration of this subject has been done so

far, including [10] and the more recent [24, 23]. We follow the approach of these papers, adopting a fully-columnar layout for the compressed data, rather than defining a complex block format, with internal pointers, markers and so on. As a consequence of this design choice are that:

- Fundamentally, there is *no difference between uncompressed and compressed data* layout — in both cases, these are sets of (named) plain columns, each of a uniform width and type throughput, whose interpretation is part of the execution plan.
- Instead of breaking up compressed (or uncompressed) columns into chunks or blocks from the outset, a work scheduler has the flexibility to cut up the columns as it sees fit, to be scheduled for execution, provided the cutting-up matches for the different columns. (The latter condition is trivial for TPC-H Q1, less so in general).

As for the choice of *compression schemes* and techniques — many different techniques exist both for compressing data and for operating on compressed data; and some have been adapted or developed for GPU-targeted use, in the more strictly-columnar layout discussed above [24]. However, the TPC-H benchmark data which Q1 takes as input is not only artificially-generated, but also rather uncompressible: Other than some deterministic formulae, it uses uniform independent samples for the relevant domains. Real-life data exhibits many correlations between columns and within columns, and a measure of locality (i.e. the effective domain in small consecutive sequences of elements is typically much lower than the overall column's domain) — and this motivates the more involved compression schemes. We have none of that — nor even do we have "noise", invalid data or NULLs to consider. We therefore confine ourselves to, first, potentially using Frame-of-Reference (storing offsets from a base value) or Dictionary compression, then applying Null Suppression (discarding unused bits).

An additional consideration we need to make is for the cost in terms of effective read throughput. For example, if one can compress a 32-bit value using 30 bits — this is typically not worth the effort: It requires reading more locations in memory

Column	SQL Type	Uncompressed Size (bits)	Method	Compressed Size (bits)	Optimum Compressed Size (bits)
l_quantity	INTEGER	32	Null-Suppression	8	6
l_extendedprice	DECIMAL(15,2)	64	Null-Suppression	32	21
l_discount	DECIMAL(15,2)	64	Null-Suppression	8	4
l_tax	DECIMAL(15,2)	64	Null-Suppression	8	4
l_shipdate	DATE	32	Frame-of-Reference + NS	16	12
l_returnflag	VARCHAR(1)	8	Dictionary	2	2
l_linestatus	VARCHAR(1)	8	Dictionary	1	1
Total per Tuple (as bytes)		272		75	50

Table 1: Compression schemes for the columns in `lineitem` used in TPC-H Query 1

for obtaining a single value; and shifting and combining bits from consecutive reads; this is even less convenient on GPUs as all memory accesses must be properly aligned (e.g. the 4-byte value cannot be read from an address with low bits 10).

Our choices for compression under the above considerations appear in Table 1. Note that the sub-optimality is 9% of the original tuple width, but 25% of the optimum width (the optimum assumes an integral number of bits per value).

3.2 Hashing vs. sorting

There are two common fundamental approaches to performing a grouping operation: (1) Use of hash functions on group keys and (2) Sorting data, by order of the group keys [18]. With the hash-based grouping, a hash table is constructed, the indices into which are obtained by applying a hash function to the group keys (in the SQL context: `GROUP BY` columns). This includes the special case of "non-hashing", i.e. use of an identity or something similarly trivial as the hash function (see below in Subsection 3.3). For a group-by-aggregation query, the hash table entries contain the aggregates, updated as the input table is scanned: The processing of a tuple involves computing its hash, looking up the hash table entry (within one or more tables), and updating the aggregate with the tuple's data.

Sort-based grouping, on the other hand, is executed by first sorting the data along with the keys, by order of the keys, then scanning the sorted data and reducing/aggregating the sequences with identical group keys.

A methodical comparison of these methods is performed in Karnagel et al. [17]. The hash-based implementations performs better in all circumstances for lower numbers of groups, and when the grouping is carried out in a pipelined fashion, outpeforms sort-based implementations on larger numbers of groups as well. The sort-based grouping is only worth it if (1) the entire dataset fits in the GPU memory and is processed all at once, and (2) the amount of groups is very high. The scenario of grouping smaller amounts of data into larger numbers of groups is not very common. This confirms our intuitive predisposition to focus on a hash-table-based implementation, particularly for TPC-H Q1.

3.3 Hash Table Design

Collision Resolution. The main issue when dealing with hash table design on the GPU is that of *collisions* — identical hash values for different group key inputs to the chosen hash function. When two groups share the same hash value, this collision has to be resolved in some way, as they may not be aggregated together. Common approaches to resolving collisions — chaining, probing and cuckoo hashing — suffer from the inherent problem of introducing data-dependent

branching: When a collision occurs, the collision has to be resolved by either following a chain to find a free position; by continued probing of subsequent table cells, or by re-applying a hashing function. Each of these has worst-case complexity linear in the size of the hash table, despite the amortized constant-time complexity is $O(1)$. In practical terms, even a few such data-dependent branches are highly disruptive on the GPU: When even a single thread in a warp takes a branch, all other warp threads follow its execution path, despite if themselves only waiting to take the other branch; and the branching thread then waits for the rest to take their own path. Further discussion of using collision resolution techniques in a GPU hash table implementation is discussed in [4].

Collision Avoidance. Instead of resolving collisions as they occur, one may invest effort in *avoiding* them altogether — by obtaining a hash function which is guaranteed to be an injective function for the input data [5]. This ensures no divergent branching is necessary during execution and allows us to use a simpler hash table layout.

There are several methods for obtaining a perfect hash function. The simplest is to use the identity function as the hash. In this case, the hash table size must span the entire domain of the group keys. This is only possible for small-domain data types. For larger-domain types, if the group keys are known to be losslessly compressible, one may use an enumeration of the compressed form as the hash key, or alternatively an enumeration of a superset of the comrpessed form. Specifically, if the group keys are known to fit into a small contiguous range, [min,max], the hash can be a computation of the offset from min, with a hash table size of $|\max - \min|$.

For a small set of group keys, and when preprocessing is not available to obtain an enumeration, a perfect hash might still be feasible if each extant group is sufficiently frequent in the column. In such cases, a uniform sample of group keys is likely to yield the entire set; and given this set, a perfect hash function can be obtained with high probability when sampling from a universal family of hash functions. This is first described in the context of hash tables in [11], or using the easier-to-compute family discussed in [9].

Without knowledge of the frequency distribution of group keys, one cannot realize whether the above is indeed the case. A possible approach is to assume optimistically that it is, and upon encountering a collision, to fall re-generate another, perfect, hash function or to fall back to a collision-resolution-based method.

If the set of group keys is large, searching for a perfect hash function is not worthwhile relative to the overhead of collision resolution.

Our case in focus, TPC-H Q1, admits a straightforward perfect hash: Despite grouping by two distinct columns, which

usually implies a large set of group keys (the cartesian product of two sets), the `l_returnflag` and `l_linestatus` columns of the `lineitem` table only contain 3 and 2 distinct values respectively. Dictionary compression of each of the columns can be combined to obtain this enumeration, as was done in [2]. The column ranges are `'F'..'O'` and `'A'..'R'` respectively, for a combined range size $17 \cdot 9 = 153$; a hash table of this size will therefore suffice to map pairs of values to an index without enumeration, i.e. on-the-fly, with no need for preprocessing/compression.

3.4 Hash Table Placement

Hash tables can be placed in one of several spaces of within the physical hierarchy of GPU memory: Global device memory (possibly with caching); "Shared Memory" on a single computing core (and common to all threads of a grid block executing on that core); or a core's register file (to which access is thread-specific); and they can even be placed in specially-allocated main system memory and accessed piecemeal from the GPU. Figure 1 illustrates this range of options. Shared-Memory and in-registers tables can also theoretically be split up, so that a single core only holds part of a single table, or a single table is distributed over multiple threads' associated registers.

Intuitively, one would try to simply place the tables in the lowest level of the hierarchy, the "fastest memory", into which the table fits; but this intuition can be misleading. Unlike the simpler hierarchy of CPU memory (Global memory with coherent caches of decreasing sizes, not-so-many machine registers) — the GPU's hierarchy is more complex. It is not a simple spectrum, from slowest to fastest. Specifically:

- Caches are *not* coherent, and cache lines sizes differ between levels — with L1 cache being even larger than L2.
- There is more space available on registers (64 Ki registers = 256 KiB per core on NVIDIA Kepler and later) as there is available in Shared memory (64 or 96 KiB per core on Maxwell and later).
- Registers do not support indexed (a.k.a. indirect) access — loading the index of the register to access from another register. They thus have to be scanned in larger numbers merely for a single access, offsetting the greater latency of indexed shared memory accesses.
- Then again, shared memory is organized in 32 interleaved "banks" of 4-byte elements, and if different lanes of the same warp access different elements in the same bank — these accesses are serialized, multiplying the warp's access latency.
- Excess use of registers, or indirect-indexed access to registers, in source code actually translates to the use of global device memory, but in a thread-local fashion (referred to as Local Memory in CUDA parlance). Avoiding this is not trivial without intentional under-utilization of regiers.
- Atomic access has different absolute and relative penalties in shared memory and in Global device memory.

These conflicting considerations are already sufficient to raise interest in comparing implementations using all three physical memory spaces.

3.5 Hash Table Granularity

Another design decision to make is whether to use fewer hash tables as possible, perhaps even just a single table — or rather, more tables, localized to some of the data or some of the working threads, warps or blocks. The granularity is constrained from below by the choice of physical memory space: Placing the table in shared memory or in registers requires at least one separate table for each thread block or thread, or alternatively, splitting a table between blocks or threads, and acting on those parts independently. From above, the choice of memory space together with the table design serve as a hard constraint — the capacity of shared memory and the register file are limited. A soft constraint is posed by features such as the 32-bank structure of NVIDIA GPU shared memory: With more than 32 full tables per block, it becomes difficult or impossible to prevent bank conflicts between accesses to the different tables (see also in Subsection 4.1 below); and when these conflicts occur, one might as well reduce the number of tables and have more threads coordinate access to the same table.

The constraints still leave ranges of choice at different spaces: Between 1 and 32 tables in shared memory (if the table is not too large); between one split table up to 1024 per-thread tables, theoretically; and in global memory we have the greatest range, with only the largest of tables being limited by the global memory size of several GB.

The more tables are used, and the more individual tables are split up, the more reintegration and table-aggregation work is to be done as a last phase. With large amounts of data, the overhead of this final phase is negligible, but it raise the bar for what constitutes "enough" data to benefit from the on-GPU execution.

As far as performance is concerned, a single hash table in global memory that is accessed directly by all GPU threads is not expected to perform well. Its performance on a GPU should be weaker than on a CPU, due to higher memory access latency, exacerbated by the higher number of threads performing atomic accesses. Such contention becomes even worse with small hash tables, or when the group keys exhibit some locality in the order by which tuples are processed — as more threads will race to update the same table entries.

At the other end of the spectrum is a full hash table per each thread. The table can even reside in global memory physically. This may even be practical: GPUs offer a "local memory" mechanism which is stored in per-thread areas in global memory; and while access to this memory has high latency in the worst-case, caching and latency-hiding can make it faster than one might imagine.

Alternatively, the per-thread tables must reside in shared memory or in registers. This method can only be used for a small number of groups (there is at most 1536 KiB of shared memory per thread). This constraint is even stricter for the in-register hash tables, because of the need to scan them on each access.

Another dimension of granularity we have hinted at is the conceptual partition of individual hash tables, so that an individual thread is only concerned with one part of a table at a time — and either makes another pass for another part of the table later on, or leaves it to other threads to read the same data and update the global aggregates. This means that a thread-local hash table only has to store groups that are relevant for the thread it is associated with. Such decomposition may allow placement in a lower level of the memory hierarchy even at larger sizes. It should be noted that hash table partitioning was also used in [16, §4.2], but for different purposes and under assumptions irrelevant to our case.

Larger-than-GPU-memory hash tables. In some cases, the set of group keys may be so large that even a single hash table cannot fit into GPU global memory. If the factor of excess relative to available GPU memory is very small, one

could partition the hash table across several GPUs, using the approach mentioned above: Each GPU will only consider tuples with group keys in a designated range; the details of doing so exceed the scope of this work. One could also use table partitioning to run multiple passes on a GPU, each ignoring all tuples except those which hashes into the GPU-resident part of the table. Unfortunately, if the execution of a single pass is PCIe-I/O-bound (as is often the case), this approach slows execution down and thus does not scale.

When a multi-pass approach is inapplicable, very large hash table would likely have be placed in main system memory. GPU drivers allow for allocating system memory which is accessible directly from the GPU, and recent microarchitecture advances in NVIDIA GPUs (the Pascal generation and later) have significantly improved the performance in such scenarios: GPU memory is virtualized and paged so that a page fault due to a page not being in GPU memory triggers a copy of that page from system memory, with no need for orchestration of these transfers by user code [27].

As the very large hash table is the extreme opposite case from the one we face with our choice of TPC-H Q1, we do not explore further accommodating it, nor is it part of our experimental evaluation. It can be said with near certainty, though, that GPU use would not be recommended for this kind of a query on an Intel-like platform (where it is a second-class citizen w.r.t. memory access); but a GPU may well outperform a CPU socket on a more "even-handed" platform.

3.6 Co-processing

Up to this point we have only discussed the execution of the query as a whole on the GPU. On a typical platform there is also a CPU controlling the platform. Naturally, there is no reason to leave the CPU idle while the GPU is used: System memory bandwidth is underused, and having both processing units working concurrently should allow us to squeeze extra performance from the system.

When processing a more complex query, there are innumerable possibilities of splitting work between the CPU and the GPU for different parts of the execution plan; and we would choose among them so that the each PU would be assigned work it tends to better at; and while considering memory bandwidth use, the overhead of sending intermediary results back and forth, and so on. In group-by-aggregation queries, especially simpler ones such as TPC-H Q1, these are much more limited, especially with only a single table being involved. We identify the following options for co-processing in this scenario:

1. **Filter Pre-computation.** Instead of fusing the entire query's work together into a single kernel, we can instead first compute the `WHERE` clause filter result, obtaining a column of boolean values, represented as a bit vector. This can then be used as input to the actual aggregation. By computing the filter on the CPU, we do not need to transfer the table columns involved only in filter computation to the GPU. Doing so is a trade-off: We benefit from less I/O to the GPU in exchange for more I/O from main system memory to the CPU and back and some computation time on the CPU.

    For TPC-H Q1, the I/O saved per tuple is 31 bits with no compression, 15 bits with our chosen compression schemes or 11 with the optimal compression scheme, which makes for 11%, 20% or 22% of the tuple representation size, respectively (see Table 1).

2. **Filter Pre-application.** A further step beyond filter precomputation would be to execute the filter on the CPU side and copy only the matching columns to the CPU. This approach, however, would require us to scan over the entire input data and copy the matching columns to a separate location prior to transferring it to the GPU. The question is then, if we are already scanning the entire data set in the CPU, is it not faster to directly perform the aggregation instead of creating the new columns and still transferring them to the GPU. Especially for filters with a high selectivity, this approach requires a lot of CPU processing for little gain.

3. **Aggregate Split (Vertical Partitioning).** In queries involving multiple aggregates, the aggregates involve a disjoint or nearly-disjoint set of columns. In this case, certain aggregates could be computed entirely on the CPU and other aggregates could be computed entirely on the GPU. In TPC-H Q1, for example, the aggregates involving `l_quantity` do not involve `l_extendedprice`, `l_tax` and `l_discount`. While some columns would be shared (e.g. the `GROUP BY` and `WHERE` columns in our case) Such vertical partitioning-with-overlap of the query narrows the set of columns each processing unit has to scan.

4. **Data Parallelism (Horizontal partitioning).** The other axis for partitioning the computation involves having CPU and GPU both compute the same set of aggregates, but on different subsets of each of the columns. The final results of each for each of the processing unit are then merged together, obtaining the final result.

In this work, we investigate the filter precomputation and data parallelism models of co-processing. The filter preapplication method is not useful when the selectivity is high (and we indeed focus on the default parameter values in the TPC-H spec, in which the selectivity $> 90\%$); and when selectivity is low, the CPU must scan all of the table columns to materialize the filtered data, making it less attractive to delegate the aggregation work to the GPU. Additionally, the simplicity of Q1 makes it easier to divide the work between them, both statically/apriori and dynamically, by any ratio we desire, while with an Aggregate Split there is just one, or a few, possible vertical partitions. With more complex queries, however, the vertical split could be more useful — due to further work which requires different aggregates in their entirety, which could be assigned to the different processing units.

Regardless of the choice of preprocessing option, We note that excessive memory pressure from the CPU side, as it processes its part of the workload, may potentially starve out the GPU. Suppose the main memory bandwidth is B_{main} GB/sec and the effective PCIe bandwidth is B_{PCIe}. If the CPU's activity allows it to consume more than about $B_{\text{main}} - B_{\text{PCIe}}$ GB/sec, the GPU may not get the full PCIe bus' worth of data — which will hinder its performance if it is I/O-bound rather than compute-bound. This is a realistic possibility with today's CPUs: With 8 or more cores per socket, they are indeed able them to saturate the main memory bandwidth with reads.

4. EXPERIMENTS

In this section we choose points within the design space we've mapped in Section 3 — combinations of design decisions for executing a group-by and aggregation query utilizing a

Implementation	Data-parallel Co-processing	Filter Precomp-utation	Hash table Placement	Time (sec)	Relative Standard Deviation
global	✗	✗	Global	0.754	1%
local_mem	✗	✗	Global	0.537	1%
shared_mem_per_thread	✗	✗	Shared	0.547	2%
in_registers	✗	✗	Registers	0.537	1%
in_registers_per_thread	✗	✗	Registers	0.536	1%
global	✗	✓	Global	0.848	1%
local_mem	✗	✓	Global	0.528	4%
shared_mem_per_thread	✗	✓	Shared	0.527	12%
in_registers	✗	✓	Registers	0.529	9%
in_registers_per_thread	✗	✓	Registers	0.526	4%
global	✓	✗	Global	0.473	7%
local_mem	✓	✗	Global	0.488	9%
shared_mem_per_thread	✓	✗	Shared	0.358	22%
in_registers	✓	✗	Registers	0.365	24%
in_registers_per_thread	✓	✗	Registers	0.380	19%
global	✓	✓	Global	0.476	12%
local_mem	✓	✓	Global	0.507	1%
shared_mem_per_thread	✓	✓	Shared	0.372	19%
in_registers	✓	✓	Registers	0.378	18%
in_registers_per_thread	✓	✓	Registers	0.374	23%

Table 2: Overall query execution time for TPC-H Q1 at SF100 our implementations for compressed data.

GPU — which may be evaluated in the context of TPC-H Q1. To evaluate the performance of the various points in this constellation, we use a smaller number of basic implementation, executed and timed with different run-time and "JIT-time" variations.

Experimental setup. All experiments were run on a dual-socket machine with two Xeon E5-2650 8-core CPUs (but only one socket was used) and a NVIDIA GeForce GTX 1080 Ti card; its chip is the GP104: Pascal microarchitecture, Compute capability 6.1, 28-physical-core. The system has 256 GB of DRAM memory. Software: Fedora GNU/Linux 26 with kernel version 4.15.17, CUDA version 9.1.85 and NVIDIA driver version 390.48; GCC 6.4.0 was used for compilation.

Data set. The data set used is the lineitem table from the TPC-H benchmark [28]. The TPC-H data-set is generated in a uniform distribution with no local skew, noise or outliers. We used Scale Factor 100, which results in $\approx 600M$ records in the lineitem table.

Experimental procedure. To obtain each result, we ran the query 5 times and computed the sample mean and population standard deviation. Timing began when the (un/)compressed columns were in memory, and after all relevant memory allocation had occurred (we did not allocate additional on-CPU or on-GPU memory during the query run). We did not time the 3 divisions required to provide the averages from the aggregated sums (see 1), as they are insignificant, nor the printing or transmission of the query results.

4.1 Implementation Flavors

Our implementations for TPC-H Q1 all use hash tables based aggregation; and all of them use the enumeration-based perfect hash function discussed in Subsection 3.3: The hash table has 6 entries (with 4 in actual use). This permits us to cover the spectrum of choices for placement — except for the case of exceeding GPU global memory, which cannot

occur with TPC-H Q1. We have implemented the following variations:

1. **global.** A single hash table in global GPU memory.
2. **local_mem.** A hash table for each working thread in local_memory (i.e. in thread-distinct areas of global memory).
3. **shared_mem_per_thread.** A hash table for each working thread in shared memory. The hash tables in this implementation are not placed contiguously, but are rather *interleaved* in such a way, that all of a thread's hash table lies within two consecutive shared memory banks (for 64-bit aggregates), or one bank (for 32-bit aggregates). This ensures that regardless of which table index each lane has, the warp's shared memory access into the table is guaranteed to have *no bank conflicts*. Such an interleaving is described in [30, Fig. 6 & Listing 5] (for the case of byte values).
4. **in_registers.** A separate hash table for each working thread in its registers.
5. **in_registers_per_thread.** A single table cell per working thread, in a register. In this implementation we hold several hash tables per warp, distributed across multiple threads' register, so that each thread holds (and updates) only a single table cell (for a given aggregate); this results in (warp size) / (no. table size) $= 32/6 = 5$ tables per warp.

Co-Processing. To experiment with execution on the CPU, we have adapted the implementation presented in [15], and introduced a Morsel-driven model of parallelized execution, as suggested in [19]. The implementation is guaranteed to have NUMA locality both due to the Morsel-driven model, but also apriori, as we restrict ourselves to processing on a single CPU socket. For the collaborative use of the CPU and the GPU, we have implemented two of the options for co-processing described in 3.6: Filter Precomputation and Data Parallelism.

Placement	Tuples per Thread	Threads per Grid Block					
global		32	64	128	160	256	512
	32	785	796	798	809	800	806
	64	804	820	823	821	828	820
	128	794	803	789	781	801	799
	256	779	783	785	792	786	
	512	**772**	781	784			
	1024	783	794				
in_registers		32	64	128	160	256	512
	32					116	
	64					99	
	128					**96**	
	256					98	
	512						
	1024						
in_registers_per_thread		32	64	128	160	256	512
	32	578	576	576	542	526	377
	64	552	550	550	536	524	369
	128	556	555	554	517	502	391
	256	454	447	445	447	438	
	512	**357**	360	360			
	1024	536	550				
local_mem		32	64	128	160	256	512
	32	492	489	489	462	445	321
	64	478	474	476	447	434	291
	128	478	475	474	433	417	294
	256	363	355	354	360	341	
	512	229	227	**226**			
	1024	286	297				
shared_mem_per_thread		32	64	128	160	256	512
	32	178	171	176	175		
	64	110	109	108	109		
	128	89	91	87	89		
	256	78	77	78	79		
	512	**69**	69	69			
	1024	112	110				

Table 3: Sum of kernel on-GPU execution times, in milliseconds, at SF100, for the different implementations of a TPC-H Q1 kernel, using compressed data and without filter on-CPU precomputation.

4.2 Parameter settings

When scheduling execution of each of our implementations, we can vary the following parameters without disrupting its basic mode of operation:

1. **Tuples per thread.** The number of of tuples processed by each thread, in each grid block, of each scheduled kernel.
2. **Kernel grid block size.** The number of threads assigned to the same GPU computing core ("streaming multiprocessor") with a common L1-like Shared memory region. If the number of hash tables also depends on the block size (for tables placed in Shared memory or in registers), this is indirectly a cap on that number as well.
3. **Processed batch size.** The number of tuples for which a transfer of column data to the GPU, an on-GPU kernel execution, and potentially a filter pre-computation, are scheduled once, as a whole (and using the same GPU stream).
4. **GPU streams.** The number of streams onto which we enqueue batches of work.

The batch size must be large enough to accommodate kernel launches with enough tuples processed per thread, in large enough blocks, and with enough blocks so that each GPU core ("streaming multiprocessor") has enough warps available for execution so as to hide memory access latency; in other words — it is determined last, by the other parameters. We increased

it gradually by factors of 2 after setting other parameters until, for all placements, there was no benefit or essentially no benefit in increasing it further; this occurred at $2^{22} = 4194304$ tuples per batch, which makes for nearly 40 MB of data. This affirms the common assumption that On-GPU processing requires much larger batches of data to be effective. However, it may be the case that more careful merging using less atomic accesses to global memory could reduce this value.

The number of streams used for scheduling must be at least 2, to allow for overlapped Compute and I/O. It may be the case that with more complex queries, this value is more significant; but with TPC-H Q1, however, the scheduled work is highly regular: Same-size batches and nearly-uniform kernel execution. We have chosen to fix the number of streams at 4, noticing no benefit in increasing them further, nor closely investigating performance with 2 or 3 streams.

The two remaining parameters are more complex to determine, or at least determine exactly. There is some delicate interplay between them and various specifics of the code, the microarchitecture and on the exact GPU specifications, so that their effect on performance is not even monotone, and behaves differently for each of our implementations. The specifics appear in Table 3. If one is willing to settle for close-to-optimal values, a rough rule of thumb for group-by-aggregation queries similar to TPC-H Q1 may be: As many threads per block as possible while not compromising the number of registers available to each block (which does drop after 256 on NVIDIA GPUs); and having each thread process hundreds of tuples, but keeping a decent number of grid blocks, e.g. at least 4 times the number of physical cores on the GPU.

4.3 Results

The overall query processing times and on-GPU computation times are listed in Table 2. The optimal on-GPU execution times for our different implementations, ignoring data transfers appear as the boldface cells in the heatmaps in Table 3.

We first observe that, with almost all of our implementation variants, **the execution time is very close to the PCIe transfer time** to the GPU. In the compressed case, this is 5.625 GB of data divided by 11.4 GB/sec, the typical achieved transfer rate, or about 0.493 seconds. I/O does not actually take place continuously through the execution, due to some gaps between consecutive transfers — which do not occur due to streams' being busy with compute tasks. Some of this gap can be attributed to the the fact that the computation on the last batch of data occurs begins once it has been received entirely, during which time the PCIe bus is unused; but this is not a significant part. We have not determined how the "blame" for this inefficiency is shared by our code, by the NVIDIA runtime and driver code, and by the hardware itself; our only finding on this matter is that the gaps are independent of the specific choice of implementation.

As our GPU executions (or parts therefore) are I/O bound, it is unsurprising that the use of **data compression affords the most significant performance benefit**; in fact, it is the *only* design choice with any effect, other than avoiding a single global hash table. The difference between our implementations are apparent only when ignoring the PCIe transfers and considering on-GPU execution times separately. We recall that for on-CPU execution, the challenge TPC-H Q1 poses is considered to be the high amount of raw expression arithmetic and aggregation work [6, §2.1, Table 1]. But on the GPU —

TPC-H Q1 involves only a little arithmetic and not much aggregation work — not enough to stress the GPU's raw computing power.

The single implementation which is not I/O-bound is still noteworthy: The single global hash table performance is only about 50% slower than than the I/O limit on performance. The very-few-groups case ensures the hash table resides in L2 cache (L1 residence is not useful, as many cores write to the same cache lines all the time); but it also has a lot of contention of atomic operations on the same hash table cells; and with larger group indices, the PCIe transfer time will also increase. Thus it is conceivable that performance will not degrade terribly for larger tables. On the other hand, as tables grow, less of each cache line would be used when it is fetched, and TLB misses will start affecting performance. Thus the performance here is not sufficient cause for optimism. The behavior of a global hash table with increasing size is discussed in depth in [17, §3], but in that case an actual hash function is used, and collisions are sometimes an issue.

If we do ignore the transfer time, the on-GPU computation shines: TPC-H Q1 can be processed on a single GPU in 68 milliseconds, about an order of magnitude faster than a state-of-the-art CPU implementation on our test machine.

CPU use and co-processing. In single-processor execution, The GPU implementations, despite being PCIe-bottlenecked, perform much better than the optimized CPU implementation; however, our test machine does not use the latest-generation CPU available. Preliminary testing on an Intel Kaby lake (7XXX) CPU suggests this advantage is lost, and CPU execution is about as fast as the PCIe transfer speed of the compressed data.

When applying filter precomputation, we gain slightly in speed, about 2% — but this is within 2 standard deviations of the result for most placements, making it statistically questionable. When we add filter precomputation to data-parallel co-processing, this yields little or no benefit, and in fact slows computation down by up to 5% with non-global placement. This suggests that our test machine CPU is about as fast in computing the filter as it is to transfer 15 extra bits per tuple to the GPU (assuming naively no other overhead). On a more modern CPU, the precomputation will be faster, and benefit will likely be pronounced.

The more conclusive and significant result, though, is the marked improvement with our second preprocessing option: **Data parallel coprocessing yields a 30% speedup over the best single-processor implementation** on our test system, arbitrarily assigning half of the workload to the CPU; and this figure does seem to carry over to newer CPUs.

Finally, we note that in our experiments involving CPU use, execution times varied much more significantly. While there might also be some variance with on-GPU computational work, it is is mostly hidden by the PCIe transfer times, which are more stable. We specifically noticed the first execution out of every repeated sequence of executions with the same parameters performs significantly worse, and the rest have less variance.

5. RELATED WORK

Karnagel et al. [17] studied the execution of `SELECT aggregate FROM tbl GROUP BY columns` queries. Unlike in this work, they assume a perfect hash function is not available, and focus on hash table design with an imperfect function. Similarly to this work, they also consider placing small hash tables in shared memory, but do not go "lower" than that, into the register file; nor do they consider local memory placement.

The study assumes the table placement is easily determined solely by the number of groups, while we have found that there is a more complex interplay with additional parameters. Karnagel et al. [17] characterizes regions of behavior of hash tables by the number of groups (and fill factor), on NVIDIA Kepler microarchitectural GPU. Unfortunately, the microarchitectural features changes over time. This work used a second microarchitecture after Kepler (Pascal), while a third one (Volta) was in production and another one named but not yet released. By the time the work was published, performance of shared-memory atomic operations had already greatly improved with the Maxwell microarchitecture [26], significantly impacting the results. It's also important to note that hardware features and behavior does not always correspond to the features in NVIDIA's PTX intermediary representation, as implied in [17, Table 1].

The focus in Karnagel et al. [17] on tables with larger groups also led its authors to dedicate a separate kernel to merging multiple hash tables in case those were placed in global memory — a choice we did not need to consider for TPC-H Q1. The GPU-CPU co-processing is not considered in the work, and neither do they discuss the use of compression for reading in the data.

With regard to the "CPU side": group-by-aggregation query processing purely on a CPU is treated in the recent [15]: It surveys earlier work; maps out the design space similarly to our Section 3, but for a modern CPU; and demonstrates the effectiveness of aggressively optimizing using SIMD instructions — specifically for TPC-H Q1. As mentioned earlier, the code used in [15] is used in large part in this work.

TPC-H Q1 Performance

It has been observed [3] that much of the published work on query processing using GPUs uses problematic baselines: If one speeds up a slower CPU-targeted DBMS using GPU — this does not imply that GPUs are useful for query processing, nor that one's GPU implementation is of good quality relative to the device's potential. This is true specifically for TPC-H results, typically published as performance benchmarks for GPU-utilizing query processing systems. This is illustrated in Table 4, comparing prominent recently-published results for TPC-H Q1. The results have been scaled to SF 100, under an assumption of linear scaling. Results involving CPUs are *not* normalized for different CPU strengths, as a choice of a normalization scheme is difficult (if not even a source of bias); the reader is encouraged to consult the cited publications to better understand the numbers.

As noted above, a reasonable implementation of Q1 on the GPU, reasonably scheduled, should be strongly PCIe-bottlenecked, with actual kernels taking much less time than that; however, existing systems are typically an order of magnitude slower in overall performance, and considering also tbl:grid-params-heatmap-and-kernel-perf — apparently much slower still with resplect to on-GPU work only.

That is not to discount these contributions: In this work, we've had the luxury of hand-optimizing — even if such optimization could theoretically be the result of JIT compilation; and our custom code need only accommodate one use case, while these systems are designed to be general. Still, a GPU-utilizing system, which uses group-by-aggregation queries as

System	Reported in	CPU / GPU	Time (msec)	Published SF	Scaled to SF 100
Red Fox	[31, Table 3]	GPU	330	1	(33000)
Ocelot	[22, Fig. 12]	GPU	347	10	3470
Voodoo	[22, Fig. 12]	GPU	294	10	2940
AXE/GPU	[2, Table 3]	GPU	25.8	1	2580
CoGaDB/HorseQC	[12, Fig. 22]	GPU	∼350	10	3500
(This work)	Table 2	GPU	536	100	536
(Net PCIe I/O, compressed)		GPU	493	100	493
Red Fox	[31]	CPU	2760	1	2.7e6
Voodoo	[22, Fig. 13]	CPU	162	10	1620
HyPeR DB	[22, Fig. 13]	CPU	120	10	1200
AXE/CPU	[2, Table 3]	CPU	(83.8)	1	(8380)
(This work)	Table 2	CPU	791	100	791
(This work)		CPU+GPU	373	100	385

Table 4: TPC-H Q1 execution times in previously published work.

key benchmarks, is fundamentally flawed unless it either over-performs a relevant CPU, or at least is PCIe-I/O-bounded. The first option is highly unlikely with TPC-H Q1 on an Intel platform, but the second one is not exceedingly hard to achieve. When hand-coded kernels are faster than a system's execution plan by over 100x on a prominent benchmark query, the system's design is called into question; specifically, it makes it difficult to draw conclusions regarding such systems choices of points within the design space for the relevant category of queries.

A final point regarding to most systems mentioned above regards co-processing. Ocelot, Red Fox, Voodoo and "AXE/GPU" of [2] all have published results both for processing on a CPU and for processing on a GPU; but their authors have not published results nor claimed the capability of *co-processing*, with both PUs at once, as we have done in this work. Perhaps this is due to the general nature of these experimental systems: Since they've not tackled the challenge of preprocessing generally, no SQL-fragment-specific solution is available even for simple queries. On the other hand, it should not be exceedingly difficult for a query processor's developers to implement at least some opportunistic data-parallel coprocessing. Unfortunately, at least three (if not all four) of the above systems are essentially abandoned, so this situation is unlikely to change for them.

6. CONCLUSION

In this paper, we have explored part of the design space of processing grouping and aggregation operations on GPU+CPU co-processing, in the admittedly narrow context of TPC-H Query 1. Overall, we are pleased to have achieved significantly improved performance with respect to an aggressively optimized CPU-only baseline.

Future Work

We plan to extend our work to Group-By/aggregations involving larger numbers of groups. This would involve both variations on our implemented placement choices as well as work with in-system-memory tables (working on table segments and/or via unified memory access). It is particularly interesting to compare the competitive and collaborative behaviors of CPUs and GPUs, as the performance degradation with table size increase behaves differently for each kind of processor.

We believe it is also necessary to experiment with group-by aggregation queries on non-uniform data distributions — even still in the limited context of TPC-H Q1-like queries — as these distributions heavily influence the processing of a query. We expect that other compression schemes will be required, likely with significantly better ratios. The need to decompress these may shift the balance more in the favor of GPUs relative to CPUs.

There is also a significant work to be done to integrate design choices such as those we have described into a proper DBMS. This work includes, among other matters, a a kernel JIT compilation, with more complex operator fusion than in existing (CPU and GPU) systems and a richer optimization framework recognizing some of the concepts and considerations we have discussed. We also believe that such a system will lend itself towards multi-faceted co-processing, marshalling dependent and independents fragments on an execution plan on and between CPUs, GPUs and other processing devices, dynamically learning to exploit the "best of both (or multiple) worlds" [14].

7. REFERENCES

[1] D. Abadi, P. Boncz, and S. Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers, 2013.

[2] A. Agbaria, D. Minor, N. Peterfruend, O. Rosenberg, and E. Rozenberg. Overtaking cpu DBMSes with a GPU in whole-query analytic processing. In *Proc. ADMS*, 2016.

[3] A. Agbaria, D. Minor, N. Peterfruend, O. Rosenberg, E. Rozenberg, and R. Talyansky. Towards a real GPU-speedup in SQL query processing with a new heterogeneous execution framework. In *GTC*. NVIDIA, 2015.

[4] D. A. F. Alcantara. *Efficient Hash Tables on the GPU*. PhD thesis, UC Davis, Davis, CA, USA, 2011.

[5] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, Displace, and Compress. In A. Fiat and P. Sanders, editors, *Proc. ESA*, pages 682–693, Berlin, Heidelberg, 2009. Springer.

[6] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking*, pages 61–76. Springer, 2014.

[7] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084 – 1096, 2013.

[8] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. *GPU-Accelerated Database Systems: Survey and Open Challenges*, pages 1–35. Springer, 2014.

[9] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[10] W. Fang, B. He, and Q. Luo. Database Compression on Graphics Processors. *PVLDB*, 3(1-2):670–680, 2010.

[11] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[12] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proc. ICDE*, pages 1603–1618. ACM, 2018.

[13] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proc. ISPASS*, pages 134–144, Apr 2011.

[14] T. Gubner. Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware. In *Proc. ICDE*, 2018.

[15] T. Gubner and P. A. Boncz. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. In *ADMS@VLDB*, 2017.

[16] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient Gather and Scatter operations on graphics processors. In *Proc. SC*, page 46. ACM, 2007.

[17] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing GPU-accelerated group-by and aggregation. In *Proc. ADMS*, 2015.

[18] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, Aug. 2009.

[19] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proc. SIGMOD*, pages 743–754. ACM, 2014.

[20] X. Mei and X. Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):72–86, Jan. 2017.

[21] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26:80–113, 03 2007.

[22] 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, Oct 2016.

[23] E. Rozenberg. Decomposing and re-composing lightweight compression schemes and why it matters. In *Proc. ICDE*, 2018.

[24] E. Rozenberg and P. A. Boncz. Faster across the PCIe bus: A GPU library for lightweight decompression. In *Proc. DaMoN*, pages 8:1–8:5, 2017.

[25] E. Rozenberg, D. Tomé, T. Gubner, M. Raasveldt, and P. Boncz. TPC-H Q1 optimization experiments code. https://github.com/diegomestre2/tpchQ01_GPU.

[26] N. Sakharnykh. GPU pro tip: Fast histograms using shared atomics on maxwell, Mar 2015. https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/.

[27] N. Sakharnykh. Beyond GPU memory limits with unified memory on Pascal. NVIDIA Parallel4All blog, Dec 2016.

[28] TPC Council. TPC benchmark H v2.17.3. http://www.tpc.org/tpch.

[29] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *DBTEST 2018*, 2018.

[30] N. Wilt. Histograms in CUDA: Privatized for fast, level performance. http://informit.com/articles/article.aspx?p=2143393.

[31] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on GPUs. In *Proc. CGO*, pages 44–54. ACM, 2014.