A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics (Extended Version)

Anil Shanbhag
MIT
anil@csail.mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

ABSTRACT

There has been significant amount of excitement and recent work on GPU-based database systems. Previous work has claimed that these systems can perform orders of magnitude better than CPU-based database systems on analytical workloads such as those found in decision support and business intelligence applications. A hardware expert would view these claims with suspicion. Given the general notion that database operators are memory-bandwidth bound, one would expect the maximum gain to be roughly equal to the ratio of the memory bandwidth of GPU to that of CPU. In this paper, we adopt a model-based approach to understand when and why the performance gains of running queries on GPUs vs on CPUs vary from the bandwidth ratio (which is roughly 16× on modern hardware). We propose Crystal, a library of parallel routines that can be combined together to run full SQL queries on a GPU with minimal materialization overhead. We implement individual query operators to show that while the speedups for selection, projection, and sorts are near the bandwidth ratio, joins achieve less speedup due to differences in hardware capabilities. Interestingly, we show on a popular analytical workload that full query performance gain from running on GPU exceeds the bandwidth ratio despite individual operators having speedup less than bandwidth ratio, as a result of limitations of vectorizing chained operators on CPUs, resulting in a 25× speedup for GPUs over CPUs on the benchmark.

1 INTRODUCTION

In the past decade, special-purpose graphics processing units (GPUs) have evolved into general purpose computing devices, with the advent of general purpose parallel programming models, such as CUDA [3] and OpenCL [7]. Because of GPU's high compute power, they have seen significant adoption in deep learning and in high performance computing [4]. GPUs also have significant potential to accelerate memory-bound applications such as database systems. GPUs utilize High-Bandwidth Memory (HBM), a new class of RAM that has significantly higher throughput compared to traditional DDR RAM used with CPUs. A single modern GPU can have up to 32 GB of HBM capable of delivering up to 1.2 TBps of memory bandwidth and 14 Tflops of compute. In contrast, a single CPU can have hundreds of GB of memory with up to 100 GBps memory bandwidth and 1 TFlop of compute.

This rise in memory capacity, coupled with the ability to equip a modern server with several GPUs (up to 20), means that it's possible to have hundreds of gigabytes of GPU memory on a modern server. This is sufficient for many analytical tasks; for example, one machine could host several weeks of a large online retailer's (with say 100M sales per day) sales data (with 100 bytes of data per sale) in GPU memory, the on-time flight performance of all commercial airline flights in the last few decades, or the time, location, and (dictionary encoded) hash tags used in every of the several billion tweets sent over the past few days.

In-memory analytics is typically memory bandwidth bound. The improved memory bandwidth of GPUs has led some researchers to use GPUs as coprocessors for analytic query processing [15, 24, 44, 48]. However, previous work leaves several unanswered questions:

- GPU-based database systems have reported a wide range of performance improvement compared to CPU-based database systems, ranging from 2× to 100×. There is a lack of consensus on how much performance improvement can be obtained from using GPUs. Past work frequently compares against inefficient baselines, e.g., MonetDB [24, 44, 48] which is known to be inefficient [29]. The empirical nature of past work makes it hard to generalize results across hardware platforms.
- Past work generally views GPUs strictly as an coprocessor. Every query ends up shipping data from CPU to GPU over PCIe. Data transfer over PCIe is an order of magnitude slower than GPU memory bandwidth, and typically less than the CPU memory bandwidth. As a result, the PCIe transfer time becomes the bottleneck and limits gains. To the extent that past work shows performance improvements using GPUs as an coprocessor, much of those gains may be due to evaluation against inefficient baselines.
- There has been significant improvement in GPU hardware in recent years. Most recent work on GPU-based database [15] evaluates on GPUs which have memory capacity and bandwidth of 4 GB and 150 GBps respectively,

while latest generation of GPUs have 8× higher capacity and bandwidth. These gains significantly improve the attractiveness of GPUs for query processing.

In this paper, we set out to understand the true nature of performance difference between CPUs and GPUs, by performing rigorous model-based and performance-based analysis of database analytics workloads after applying optimizations for both CPUs and GPUs. To ensure that our implementations are state-of-the-art, we use theoretical minimums derived assuming memory bandwidth is saturated as a baseline, and show that our implementations can typically saturate the memory bus, or when they cannot, describe in detail why they fall short. Hence, although we offer some insights into the best implementations of different operators on CPUs and GPUs, the primary contribution of this paper is to serve as a guide to implementors as to what sorts of performance differences one should expect to observe in database implementations on modern versions of these different architectures.

Past work has used GPUs mainly as coprocessors. By comparing an efficient CPU implementation of a query processor versus an implementation that uses the GPU as a coprocessor, we show that GPU-as-coprocessor offers little to no gain over a pure CPU implementation, performing worse than the CPU version for some queries. We argue that the right setting is having the working set stored directly on GPU(s).

We developed models and implementations of basic operators: Select, Project, and Join on both CPU and GPU to understand when the ratio of operator runtime on CPUs to runtime on GPUs deviates from the ratio of memory bandwidth of GPU to memory bandwidth of CPU. In the process, we noticed that the large degree of parallelism of GPUs leads to additional materialization. We propose a novel execution model for query processing on GPUs called the *Tile-based execution model*. Instead of looking at GPU threads in isolation, we treat a block of threads ("thread block") as a single execution unit, with each thread block processing a tile of items. The benefit of this tile-based execution model is that thread blocks can now cache tiles in shared memory and collectively process them. This helps avoid additional materialization. This model can be expressed using a set of primitives where each primitive is a function which takes as input of set of tiles and outputs a set of tiles. We call these primitives *block-wide functions*. We present `Crystal`, a library of block-wide functions that can be used to implement the common SQL operators as well as full SQL queries. Furthermore, we use `Crystal` to implement the query operators on the GPU and compare their performance against equivalent state-of-the-art implementations on the CPU. We use `Crystal` to implement the Star-Schema Benchmark (SSB) [30] on the GPU and compare it's performance against our own CPU implementation, a state-of-the-art CPU-based OLAP DBMS and a state-of-the-art GPU-based OLAP DBMS. In both cases, we develop models assuming memory bandwidth is saturated and reason about the performance based on it.

In summary, we make the following contributions:

- We show that previous designs which use the GPU as a coprocessor show no performance gain when compared against a state-of-the-art CPU baseline. Instead, using modern GPU's increased memory capacity to store working set directly on the GPU is a better design.
- We present `Crystal`, a library of data processing primitives that can be composed together to generate efficient query code that can full advantage of GPU resources.
- We present efficient implementations of individual operators for both GPU and CPU. For each operator, we provide cost models that can accurately predict their performance.
- We describe our implementation of SSB and evaluate both GPU and CPU implementations of it. We present cost models that can accurately predict query runtimes on the GPU and discuss why such models fall short on the CPU.

2 BACKGROUND

In this section, we review the basics of the GPU architecture and describe relevant aspects of past approaches to running database analytics workloads on CPU and GPU.

2.1 GPU Architecture

Many database operations executed on the GPU are performance bound by the memory subsystem (either shared or global memory) [48]. In order to characterize the performance of different algorithms on the GPU, it is, thus, critical to properly understand its memory hierarchy.

Figure 1 shows a simplified hierarchy of a modern GPU. The lowest and largest memory in the hierarchy is the *global memory*. A modern GPU can have global memory capacity of up to 32 GB with memory bandwidth of up to 1200 GBps. Each GPU has a number of compute units called *Streaming Multiprocessors (SMs)*. Each SM has a number of cores and a fixed set of registers. Each SM also has a *shared memory* which serves as a scratchpad that is controlled by the programmer and can be accessed by all the cores in the SM. Accesses to global memory from a SM are cached in the L2 cache (L2 cache is shared across all SMs) and optionally also in the L1 cache (L1 cache is local to each SM).

Processing on the GPU is done by a large number of threads organized into *thread blocks* (each run by one SM). Thread blocks are further divided into groups of threads called warps (usually consisting of 32 threads). The threads of a warp execute in a *Single Instruction Multiple Threads (SIMT)* model, where each thread executes the same instruction stream on different data. The device groups global memory loads and stores from threads in a single warp such that multiple loads/stores to the same cache line are combined into a single

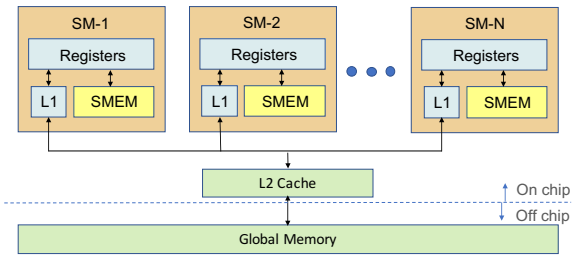


Figure 1: GPU Memory Hierarchy

request. Maximum bandwidth can be achieved when a warp's access to global memory results in neighboring locations being accessed.

The programming model allows users to explicitly allocate global memory and shared memory in each thread block. Shared memory has an order of magnitude higher bandwidth than global memory but has much smaller capacity (a few MB vs. multiple GB).

Finally, registers are the fastest layer of the memory hierarchy. If a thread block needs more registers than available, register values spill over to global memory.

2.2 Query Execution on GPU

With the slowing of Moore's Law, CPU performance has stagnated. In recent years, researchers have started exploring heterogeneous computing to overcome the scaling problems of CPUs and to continue to deliver interactive performance for database applications. In such a hybrid CPU-GPU system, the two processors are connected via PCIe. The PCIe bandwidth of a modern machine is up to 16 GBps, which is much lower than the memory bandwidth of either CPU or GPU. Therefore, data transfer between CPU and GPU is a serious performance bottleneck.

Past work in the database community has focused on using the GPU as a coprocessor, which we call the *coprocessor model*. In this model, data primarily resides in CPU's main memory. For query execution, data is shipped from the CPU to the GPU over PCIe, so that (some) query processing can happen on the GPU. Results are then shipped back to the CPU. Researchers have worked on optimizing various database operations under the co-processor model: selection [40], join [18, 19, 22, 35, 38, 39, 47], and sort [16, 42]. Several full-fledged GPU-as-coprocessor database query engines have been proposed in recent years. Ocelot [20] provides a hybrid analytical engine as an extension to MonetDB. YDB [48] is a GPU-based data warehousing engine. Both systems used an operator-at-a-time model, where an operator library containing GPU kernel implementations of common database operators such as scans and joins is invoked on batches of tuples, running each operator to completion before moving on to the next operator. Kernel fusion [46] attempted to hide in-efficiency associated with running multiple kernels for each query like in the operator-at-a-time model. Kernel fusion fused operator kernels with producer-consumer dependency when possible to eliminate redundant data movement. As kernel fusion is applied as a post-processing step, it will miss opportunities where kernel configurations are incompatible (like the one in described in Section 3.2). HippogriffDB [24] used GPUs for large scale data warehousing where data resides on SSDs. HippogriffDB claims to achieve 100× speedup over MonetDB when the ratio of memory bandwidth of GPU to CPU is roughly 5×. We have not been able to get the source code to compare against the system. More recently, HorseQC [15] proposes pipelined data transfer between CPU and GPU to improve query runtime. As we show in the next section, using HorseQC ends up being slower than running the query efficiently directly on the CPU.

Commercial systems like Omnisci [6], Kinetica [5], and BlazingDB [2] aim to provide real-time analytical capabilities by using GPUs to store large parts (or all) of the working set. The setting used in this paper is similar to ones used by these systems. Although these systems use a design similar to what we advocate, some have claimed 1000× performance improvement by using GPUs [1] but have not published rigorous benchmarks against state-of-the art CPU or GPU databases, which is the primary aim of this paper.

2.3 Query Execution on CPU

Database operators have been extensively optimized for modern processors. For joins, researchers have proposed using cache-conscious partitioning to improve hash join performance [9–11, 25]. Schuh et al. summarized the approaches [37]. For sort, Satish et al. [36] and Wassenberg et al. [45] introduced buffered partitioning for radix sort. Polychroniou et al. [32] presented faster variants of radix sort that use SIMD instructions. Sompolski et al. [41] showed that combination of vectorization and compilation can improve performance of project, selection, and hash join operators. Polychroniou et al. [31] presented efficient vectorized designs for selections, hash tables, and partitioning using SIMD gathers and scatters. Prashanth et al. [27] extended the idea to generate machine code for full queries with SIMD operators. We use ideas from these works, mainly the works of Polychroniou et al. [31, 32] for our CPU implementations.

C-Store [43] and MonetDB [12] were among the first column-oriented engines, which formed the basis for analytical query processing. MonetDB X100 [13] introduced the idea of vectorized execution that was cache aware and reduced memory traffic. Hyper [29] introduced the push-based iteration and compiling queries into machine code using LLVM. Hyper was significantly faster than MonetDB and brought query performance close to that of handwritten C code. We compare the performance of our CPU query implementations against MonetDB [12] and Hyper [29].

```
SELECT SUM(lo_extendedprice * lo_discount) AS revenue
FROM lineorder
WHERE lo_quantity < 25
AND lo_orderdate >= 19930101 AND lo_orderdate <= 19940101
AND lo_discount >= 1 AND lo_discount <= 3;
```

Figure 2: Star Schema Benchmark Q1.1

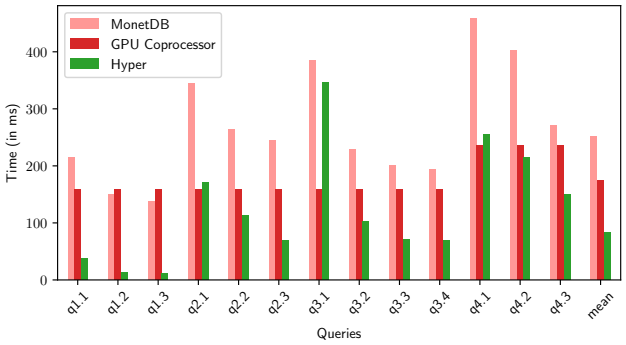


Figure 3: Evaluation on the Star Schema Benchmark

3 OUR APPROACH

In this section, we describe the tile-based execution model we use to execute queries on GPU efficiently. We begin by showing why the coprocessor model used by past works is a suboptimal design and motivate why storing the working set directly on the GPU in a heterogeneous system (as done by all commercial systems) is a better approach. Through an example, we illustrate the unique challenges associated with running queries in a massively-parallel manner on the GPU. We show how by treating the thread block as the basic execution unit with each thread block processing a tile of items (similar to vector-based processing on the CPU where each thread processes a vector of items a time) leads to good performance on the GPU. We call this approach the tile-based execution model. Finally, we show how this model can be expressed using a set of primitives where each primitive is a function which takes as input of set of tiles and outputs a set of tiles. We call these primitives block-wide functions. We present `Crystal`, a library of block-wide functions that can be composed to create a full SQL query.

3.1 Failure of the Coprocessor Model

While past work has claimed speedups from using GPUs in the coprocessor model, there is no consensus among past work about the performance improvement obtained from using GPUs, with reported improvements varying from 2× to 100×.

Consider Q1.1 from the Star Schema Benchmark (SSB) shown in Figure 2. For simplicity, assume all column entries are 4-byte integers and L is the number of entries in `lineorder`. An efficient implementation on a CPU will be able to generate the result using a single pass over the 4 data columns. The optimal CPU runtime (R_C) is upper bounded by $16L/B_c$ where B_c is the CPU memory bandwidth. This is an upper bound because, if the predicates are selective, then we may be able to skip entire cache lines while accessing the `lo_extendedprice` column. In the coprocessor model, we have to ship 4 columns of data to GPU. Thus, the query runtime on the GPU (R_G) is lower bounded by $16L/B_p$ where B_p is the PCIe bandwidth. The bound is hit if we are able to perfectly overlap the data transfer and query execution on GPU. However, since $B_c > B_p$ in modern CPU-GPU setups, $R_C < R_G$, i.e., running the query on CPU yields a lower runtime than the running query with a GPU coprocessor.

To show this empirically, we ran the entire SSB with scale factor 20 on an instance where CPU memory bandwidth is 54 GBps, GPU memory bandwidth is 880 GBps, and PCIe bandwidth is 12.8 GBps. The full workload details can be found in Section 5.1 and the full system details can be found in Table 2. We compare the performance of the `GPU Coprocessor` with two OLAP DBMSs: `MonetDB` and `Hyper`. Past work on using GPUs as a coprocessor mostly compared their performance against MonetDB [24, 44, 48] which is known to be inefficient [29]. Figure 3 shows the results. On an average, `GPU Coprocessor` performs 1.5× faster than `MonetDB` but it is 1.4× slower than `Hyper`. For all queries, the query runtime in `GPU coprocessor` is bound by the PCIe transfer time. We conclude the reason past work was able to show performance improvement with a GPU coprocessor is because their optimized implementations were compared against inefficient baselines (e.g., MonetDB) on the CPU.

With the significant increase in GPU memory capacity, a natural question is how much faster a system that treats the GPU as the primary execution engine, rather than as an accelerator, can be. We describe our architecture for such a system in the rest of this section.

3.2 Tile-based Execution Model

While a modern CPU can have dozens of cores, a modern GPU like Nvidia V100 can have 5000 cores. The vast increase in parallelism introduces some unique challenges for data processing. To illustrate this, consider running the following simple selection query as a micro-benchmark on both a CPU and a GPU:

```
Q0: SELECT y FROM R WHERE y > v;
```

On the CPU, the query can be efficiently executed as follows. The data is partitioned equally among the cores. The goal is to write the results in parallel into a contiguous output array. The system maintains a global atomic counter that acts as a cursor that tells each thread where to write the next result. Each core processes its partition by iterating over the entries in the partition one vector of entries at a time, where a vector is about 1000 entries (small enough to fit in the L1 cache). Each core makes a first pass over the first vector of entries to count

the number of entries that match the predicate d. The thread increments the global counter by d to allocate output space for the matching records, and then does a second pass over the vector to copy the matched entries into the output array in the allocated range of the output. Since the second pass reads data from L1 cache, the read is essentially free. The global atomic counter is a potential point of contention. However, note that each thread updates the counter once for every 1000+ entries and there are only around 32 threads running in parallel at any point. The counter ends up not being the bottleneck and the total runtime is approximately $\frac{D}{B_C} + \frac{D\sigma}{B_C}$ where D is the size of the column, and B_C is the memory bandwidth on the CPU.

We could run the same plan on the GPU, partitioning the data up among the thousands of threads. However, GPU threads have significantly fewer resources per thread. On the Nvidia V100, each GPU thread can only store roughly 24 4-byte entries in shared memory at full occupancy, with 5000 threads running in parallel. Here, the global atomic counter ends up becoming the bottleneck as all the threads will attempt to increment the counter to find the offset into the output array. To work around this, existing GPU-based database systems would execute this query in 3 steps as shown in Figure 4(a). The first kernel K_1 would be launched across a large number of threads. In it, each thread would read in column entries in a strided fashion (interleaved by thread number) and evaluate the predicate to count the number of entries matched. After processing all elements, the total number of entries matched per thread would be recorded in an array `count`, where `count[t]` is number of entries matched by thread `t`. The second kernel K_2 would use the `count` array to compute the prefix sum of the count and store this in another array `pf`. Recall that for an array A of k elements, the prefix sum p_A is a k element array where $p_A[j] = \sum_{i=0}^{j-1} A_j$. Thus, the i^{th} entry in `pf` indicates the offset at which the ith thread should write its matched results to in the output array `o`. Databases used an optimized routine from a CUDA library like Thrust [8] to run it efficiently in parallel. The third kernel K_3 would then read in the input column again; here the i^{th} thread again scans the i^{th} stride of the input, using `pf`$[i]$ to determine where to write the satisfying records. Each thread also maintains a local counter c_i, initially set to 0. Specifically for each satisfying entry, thread i writes it to `pf`$[i]+c_i$ and then increments c_i. In the end, `o[pf[t]] ... o[pf[t+1] - 1]` will contain the matched entries of thread `t`.

The above approach shifts the task of finding offsets into the output array to an optimized prefix sum kernel whose runtime is a function of T (where T is the number of threads ($T << n$)), instead of finding it inline using atomic updates to a counter. As a result, the approach ends up being significantly faster than the naive translation of the CPU approach to the GPU. However, there are a number of issues with this approach. First, it reads the input column from global memory

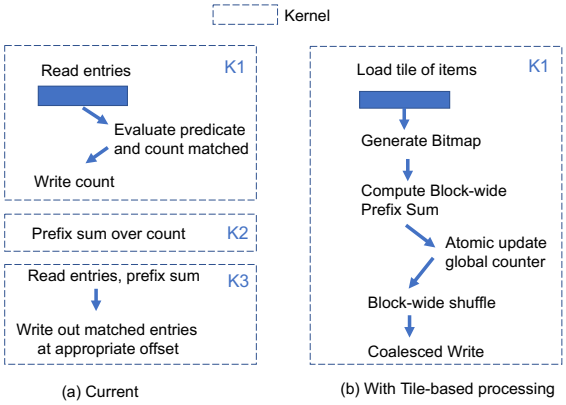


Figure 4: Running selection on GPU

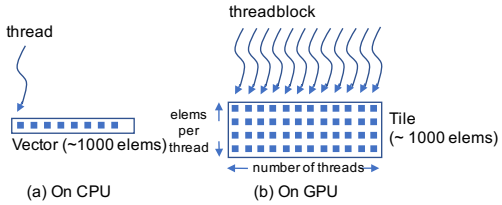


Figure 5: Vector-based to Tile-based execution models.

twice, compared to doing it just once with the CPU approach. It also reads/writes to intermediate structures `count` and `pf`. Finally, each thread writes to a different location in the output array resulting in random writes. To address these issues, we introduce the **Tile-based execution model**.

Tile-based processing extends the vector-based processing on CPU where each thread processes a vector at a time to the GPU. Figure 5 illustrates the model. As discussed earlier in Section 2.1, threads on the GPU are grouped into thread blocks. Threads within a thread block can communicate through shared memory and can synchronize through barriers. Hence, even though a single thread on the GPU at full occupancy can hold only up to 24 integers in shared memory, a single thread block can hold a significantly larger group of elements collectively between them in shared memory. We call this unit a **Tile**. In the Tile-based execution model, instead of viewing each thread as an independent execution unit, we view a thread block as the basic execution unit with each thread block processing a tile of entries at a time. One key advantage of this approach is that after a tile is loaded into shared memory, subsequent passes over the tile will be read directly from shared memory and not from global memory, avoiding the second pass through global memory described in the implementation above.

Figure 4(b) shows how selection is implemented using the tile-based model. The entire query is implemented as a single kernel instead of three. Figure 6 shows a sample execution with a tile of size 16 and a thread block of 4 threads for the predicate $y > 5$. Note that this is just for illustration, as most

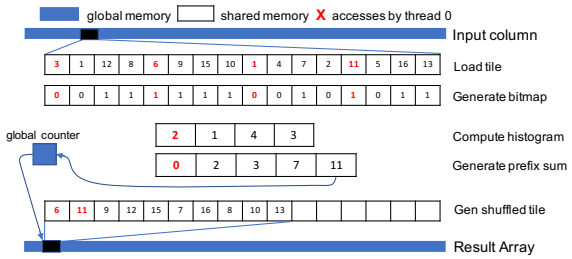


Figure 6: Query Q0 Kernel running y > 5 with tile size 16 and thread block size 4

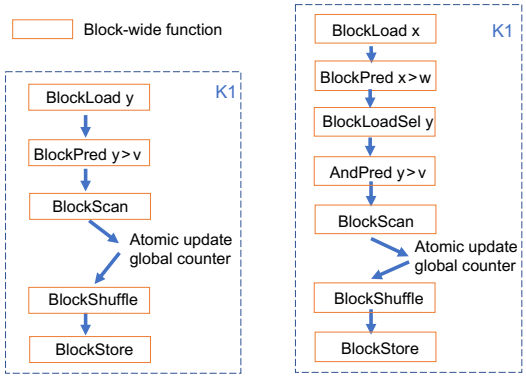
modern GPUs would use a thread block size that is a multiple of 32 (the warp size) and the number of elements loaded would be 4–16 times the size of the thread block. We start by initializing the global counter to 0. The kernel loads a tile of items from global memory into the shared memory. The threads then apply the predicate on all the items in parallel to generate a bitmap. For example, thread 0 evaluates the predicate for elements 0,4,8,12 (shown in red). Each thread then counts the number of entries matched per thread to generate a histogram. The thread block co-operatively computes the prefix sum over the histogram to find the offset each thread writes to in shared memory. In the example, threads 0,1,2,3 match 2,1,4,3 entries respectively. The prefix sum entries 0,2,3,7 tell us thread 0 should write its matched entries to output at index 0, thread 1 should write starting at index 2, etc. We increment a global counter atomically by total number of matched entires to find the offset at which the thread block should write in the output array. The shuffle step uses the bitmap and the prefix sum to create a contiguous array of matched entries in shared memory. The final write step copies the contiguous entries from shared memory to global memory at the right offset.

By treating the thread block as an execution unit, we reduce the number atomic updates of the global counter by a factor of size of tile T. The kernel also makes a single pass over the input column with the `Gen Shuffled Tile` ensuring that the final write to the output array is coalesced, solving both problems associated with approach used in previous GPU databases.

The general concept of the tile-based executing model i.e., dividing data into tiles and mapping threadblocks to tiles has been used in other domains like image processing [21] and high performance computing [8]. However, to the best of our knowledge this is the first work that uses it for database operations. In the next section, we present `Crystal`, a library of data processing primitives that can be composed together to implement SQL queries on the GPU.

3.3 Crystal Library

The kernel structure in Figure 6 contains a series of steps where each is a function that takes as input a set of tiles, and



(a) SELECT y FROM R WHERE y > v(b) SELECT y FROM R WHERE x > w AND y > v

Figure 7: Implementing queries using Crystal

```
// Implements SELECT y FROM R WHERE y > v
// NT => NUM_THREADS
// IPT => ITEMS_PER_THREAD
template<int NT, int IPT>
__global__ void Q(int* y, int* out, int v, int* counter) {
int tile_size = get_tile_size();
int offset = get_tile_offset();
__shared__ struct buffer {
  int col[NT * IPT];
  int out[NT * IPT];
};
int items[IPT];
int bitmap[IPT];
int indices[IPT];

BlockLoadInt<NT, IPT>(col+offset,items,buffer.col,tile_size);
BlockPredIntGT<NT, IPT>(items,buffer.col,cutoff,bitmap);
BlockScan<NT, IPT>(bitmap,indices,buffer.col,
  num_selections,tile_size);

if(threadIdx.x == 0)
  o_off = atomic_update(counter,num_selections);

BlockShuffleInt<NT, IPT>(items,indices,buffer.out);
BlockStoreInt<NT, IPT>(buffer.out,out + o_off,num_selections);
}
```

Figure 8: Query Q0 Kernel Implemented with `Crystal`

outputs a set of tiles. We call these primitives *block-wide functions*. A block-wide function is a device function[1] that takes in a set of tiles as input, performs a specific task, and outputs a set of tiles. Instead of reimplementing these block-wide functions for each query, which would involve repetition of non-trivial functions, we developed a library called **`Crystal`**.

`Crystal`[2] is a library of templated CUDA device functions that implement the full set of primitives necessary for executing typical analytic SQL SPJA analytical queries. Figure 7(a) shows an sketch of the simple selection query implemented using block-wide functions. Figure 8 shows the query kernel of the same query implemented with `Crystal`. We use this

[1]Device functions are functions that can be called from kernels on the GPU
[2]The source code of the Crystal library is available at `https://github.com/anilshanbhag/crystal`

Primitive	Description
BlockLoad	Copies a tile of items from global memory to shared memory. Uses vector instructions to load full tiles.
BlockLoadSel	Selectively load a tile of items from global memory to shared memory based on a bitmap.
BlockStore	Copies a tile of items in shared memory to device memory.
BlockPred	Applies a predicate to a tile of items and stores the result in a bitmap array.
BlockScan	Co-operatively computes prefix sum across the block. Also returns sum of all entries.
BlockShuffle	Uses the thread offsets along with a bitmap to locally rearrange a tile to create a contiguous array of matched entries.
BlockLookup	Returns matching entries from a hash table for a tile of keys.
BlockAggregate	Uses hierarchical reduction to compute local aggregate for a tile of items.

Table 1: List of block-wide functions

example to illustrate the key features of Crystal. The input tile is loaded from the global memory into the thread block using BlockLoad. BlockLoad internally uses vector instructions when loading a full tile and for the tail of the input array that may not form a perfect tile, it is loaded in a striped fashion element-at-a-time. BlockPred applies the predicate to generate the bitmap. A key optimization that we do in Crystal is instead of storing the tile in shared memory, in cases where the array indices are statically known before hand, we choose to use registers to store the values. In this case, items (which contains entries loaded from the column) and bitmap are stored in registers. Hence, in addition to 24 4-byte values that a thread can store in shared memory, this technique allows us to use roughly equal amount of registers available to store data items. Next we use BlockScan to compute the prefix sum. BlockScan internally implements a hierarchical block-wide parallel prefix-sum approach [17]. This involves threads accessing bitmap entries of other threads — for this we load bitmap into shared memory, reusing buffer.col shared memory buffer used for loading the input column. Shared memory is order of magnitude faster than global memory, hence loads and stores to shared memory in this case do not impact performance. After atomic update to find offset in output array, BlockShuffle is used to reorder the array and finally we use BlockStore to write to output array. The code skips some minor details like when the atomic update happens, since it is executed on thread 0, the global offset needs to be communicated back to other threads through shared memory.

In addition to allowing users to write high performance kernel code that as we show later can saturate memory bandwidth, there are two usability advantages of using Crystal:

- **Modularity:** Block-wide functions in Crystal make it easy to use non-trivial functions and reduce boilerplate code. For example, BlockScan, BlockLoad, BlockAggregate each encapsulate 10's to 100's of lines of code. For the selection query example, Crystal reduces lines of code from more than 300 to less than 30.
- **Extensibility:** Block-wide functions makes it is fairly easy to implement query kernels of larger queries. Figure 7(b) shows the implementation of a selection query with two predicates. Ordinary CUDA code can be used along with Crystal functions.

Crystal supports loading partial tiles like in Figure 7(b). If a selection or join filters entries, we use BlockLoadSel to load items that matched the previous selections based on a bitmap. In this case, the thread block internally allocate space for the entire tile, however, only matched entries are loaded from global memory. Table 1 briefly describes the block-wide functions currently implemented in the library.

To evaluate Crystal, we look at two microbenchmarks:
1) We evaluate the selection query $Q0$ with size of input array as 2^{29} and selectivity is 0.5. We vary the tile sizes. We vary the thread block sizes from 32 to 1024 in multiples of 2. We have three choices for the number of items per thread: 1,2,4. Figure 9 shows the results. As we increase the thread block size, the number of global atomic updates done reduces and hence the runtime improves until the thread block size approaches 512 after which it deteriorates. Each streaming multiprocessor on the GPU holds maximum of 2048 threads, hence, having large thread blocks reduces number of independent thread blocks. This affects utilization particularly when thread blocks are using synchronization heavily. Having 4 items per thread allows to effectively load the entire block using vector instructions. With 2 items per thread, there is reduced benefit for vectorization as half the threads are empty. With 1 item per thread there is no benefit. The best performance is seen with thread block size of 128/256 and items per thread equal to 4. In these cases, as we show later in Section 4.2 saturate memory bandwidth and hence achieve optimal performance.
2) We evaluated the selection query $Q0$ using two approaches: independent threads approach (Figure 4(a)) and using Crystal (Figure 4(b)). The number of entries in the input array is 2^{29} and selectivity is 0.5. The runtime with the independent threads approach is 19ms compared to just 2.1ms when using Crystal. Almost all of the performance improvement is from avoiding atomic contention and being able to reorder matched entries to write in a coalesced manner.

Across all of the workloads we evaluated, we found that using thread block size 128 with items per thread equal to 4 is indeed the best performing tile configuration. In the rest of the paper, we use this configuration for all implementations using Crystal. All the implementations with Crystal are implemented in CUDA C++. Since Crystal's block-wide

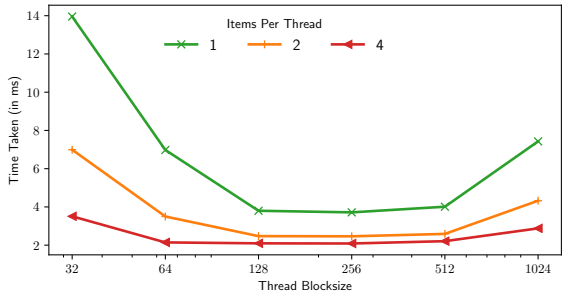


Figure 9: Q0 performance with varying tile sizes

functions are standard device functions, they can also called directly from LLVM IR.

In the next section, we show how to use these block-wide functions to build efficient operators on a GPU and compare their performance to equivalent CPU implementations.

4 OPERATORS ON GPU VS CPU

In order to understand the true nature of performance difference of queries on GPU vs. CPU, it is important to understand the performance difference of individual query operators. In this section, we compare the performance of basic operators: project, select, and hash join on GPU and CPU with the goal of understanding how the ratio of runtime on GPU to runtime on CPU compares to the bandwidth ratio of the two devices. We use block-wide functions from `Crystal` to implement the operators on GPU and use equivalent state-of-the-art implementations on CPU. We also present a model for each of the operators assuming the operator saturates memory bandwidth and show that in most cases the operators indeed achieve these limits. We use the model to explain the performance difference between CPU and GPU. For the micro-benchmarks, we use a setup where GPU memory bandwidth is 880GBps and CPU memory bandwidth is 54GBps, resulting in a bandwidth ratio of 16.2 (see Section 5 for system details). In all cases, we assume that the data is already in the respective device's memory.

4.1 Project

We consider two forms of projection queries: one that computes a linear combination of columns (Q1) and one involving user defined function (Q2) as shown below:

```
Q1: SELECT ax1 + bx2 FROM R;
Q2: SELECT σ(ax1 + bx2) FROM R;
```

where x_1 and x_2 are 4-byte floating point values. The number of entries in the input array is 2^{29}. σ is the sigmoid function (i.e., $\sigma(x) = \frac{1}{1+e^{-x}}$) which can represent the output of a logistic regression model. Note that $Q1$ consists of basic arithmetic and will certainly be bandwidth bound. $Q2$ is representative of the most complicated projection we will likely see in any SQL query.

On the CPU side, we implement two variants: `CPU` and `CPU-Opt`. `CPU` uses a multi-threaded projection where each thread works on a partition of the input data. `CPU-Opt` extends `CPU` with two extra optimizations: (1) non-temporal writes and (2) SIMD instructions. Non-temporal writes are write instructions that bypass higher cache levels and write out an entire cache line to main memory without first loading it to caches. SIMD instructions can further improve performance. With a single AVX2 instruction, for example, a modern x86 system can add, subtract, multiply, or divide a group of 8 4-byte floating point numbers, thereby improving the computation power and memory bandwidth utilization.

On the GPU side, we implement a single kernel that does two `BlockLoad`'s to load the tiles of the respective columns, computes the projection and does a `BlockStore` to store it in the result array.

Model: Assuming the queries can saturate the memory bandwidth, the expected runtime of Q1 and Q2 is

$$runtime = \frac{2 \times 4 \times N}{B_r} + \frac{4 \times N}{B_w}$$

where N is the number of entries in the input array and B_r and B_w are the read and write memory bandwidth, respectively. The first term of the formula models the runtime for loading columns x_1 and x_2, each containing 4-byte floating point numbers. The second term models the runtime for writing the result column back to memory, which also contains 4-byte floating point numbers. Note that this formula works for both CPU and GPU, by plugging in the corresponding memory bandwidth numbers.

Performance Evaluation: Figure 10 shows the runtime of queries Q1 and Q2 on both CPU and GPU (shown as bars) as well as the predicted runtime based on the model (shown as dashed lines). The performance of Q1 on both CPU and GPU is memory-bandwidth bound. `CPU-Opt` performs better than `CPU` due to the increased memory bandwidth efficiency. `GPU` performs substantially better than both CPU implementations due to its much higher memory bandwidth. The ratio of runtime of `CPU-Opt` to `GPU` is 16.56 which is close to the bandwidth ratio of 16.2. The minor difference is because read bandwidth is slightly lower than write bandwidth on the CPU and the workload has a read:write ratio of 2:1.

A simple multi-threaded implementation of Q2 (i.e., `CPU`) does not saturate memory bandwidth and is compute bound. After using the SIMD instructions (i.e, `CPU-Opt`), performance improves significantly and the system is close to memory bandwidth bound. The ratio of runtime of `CPU-Opt` to `GPU` for Q2 is 17.95. This shows that even for fairly complex projections, good implementations on modern CPUs are able to saturate memory bandwidth. GPUs do significantly better than CPUs due to their high memory bandwidth, with the performance gain equal to the bandwidth ratio.

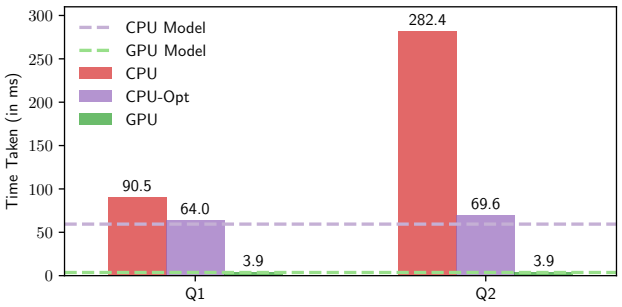


Figure 10: Project microbenchmark

```
for each y in R:
  if y > v:
    output[i++] = v
```

(a) With branching

```
for each y in R:
  output[i] = y
  i += (y > v)
```

(b) With predication

Figure 11: Implementing selection scan

4.2 Select

We now turn our attention to evaluating selections, also called selection scans. Selection scans have re-emerged for main-memory query execution and are replacing tradition unclustered indexes in modern OLAP DBMS [33]. We use the following micro-benchmark to evaluate selections:

```
Q3: SELECT y FROM R WHERE y < v;
```

where y and v are both 4-byte floating point values. The size of input array is 2^{29}. We vary the selectivity of the predicate from 0 to 1 in steps of 0.1.

To evaluate the above query on a multi-core CPU, we use the CPU implementation described earlier in Section 3.2. We evaluate three variants. The "naive" branching implementation (`CPU If`) implements the selection using an if-statement, as shown in Figure 15(a). The main problem with the branching implementation is the penalty for branch mispredictions. If the selectivity of the condition is neither too high nor too low, the CPU branch predictor is unable to predict the branch outcome. This leads to pipeline stalls that hinder performance. Previous work has shown that the branch misprediction penalty can be avoided by using branch-free *predication* technique [34]. Figure 15(b) illustrates the predication approach. Predication transforms the branch (control dependency) into a data dependency. `CPU Pred` implements selection scan with predication. More recently, vectorized selection scans have been shown to improve on `CPU Pred` by using selective stores to buffer entries that satisfy selection predicates and writing out entries using streaming stores [31]. `CPU SIMDPred` implements this approach.

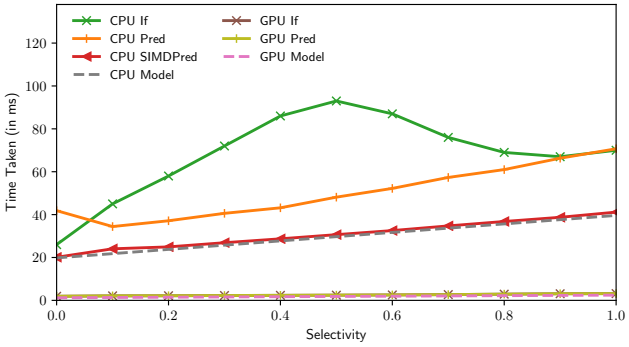


Figure 12: Select Microbenchmark

On the GPU, the query is implemented as a single kernel as described in Section 3.2 and as shown in Figure 4(b). We implement two variants: `GPU If` implements the selection using an if-statement and `GPU Pred` implements it using predication.

Model: The entire input array is read and only the matched entries are written to the output array. Assuming the implementations can write out the matched entries efficiently and saturate memory bandwidth, the expected runtime is:

$$runtime = \frac{4 \times N}{B_r} + \frac{4 \times \sigma \times N}{B_w}$$

where N is the number of entries in the input array, B_r and B_w are the read and write bandwidth of the respective device, and σ is the predicate selectivity.

Performance Evaluation: Figure 12 shows the runtime of the three algorithms on CPU, two algorithms on GPU, and the performance models. `CPU Pred` does better than `CPU If` at all selectivities except 0 (at 0, `CPU If` does no writes). Across the range, `CPU SIMDPred` does better than the two scalar implementations. On GPU, there is no performance difference between `GPU Pred` and `GPU If` — A single branch misprediction does not impact performance on the GPU. Both `CPU SIMDPred` and `GPU If/Pred` closely track their respective theoretical models which assume saturation of memory bandwidth. The average runtime ratio of CPU-to-GPU is 15.8 which is close to the bandwidth ratio 16.2. This shows that with efficient implementations, CPU implementations saturate memory bandwidth for selections and the gain of GPU over CPU is equal to the bandwidth ratio.

4.3 Hash Join

Hash join is the most popular algorithm used for executing joins in a database. Hash joins have been extensively studied in the database literature, with many different hash join algorithms proposed for both CPUs and GPUs [9–11, 14, 18, 23]. The most commonly used hash join algorithm is the no partitioning join, which uses a non-partitioned global hash table. The algorithm consists of two phases: in the *build phase*, the tuples in one relation (typically the smaller relation) are

used to populate the hash table in parallel; in the *probe phase*, the tuples in the other relation are used to probe the hash table for matches in parallel. For our microbenchmark, we focus on the following join query:

```
Q4: SELECT SUM(A.v + B.v) AS checksum
    FROM A,B WHERE A.k = B.k
```

where each table A and B consists of two 4-byte integer columns k,v. The two tables are joined on key k. We keep the size of the probe table fixed at 256 million tuples, totaling 2 GB of raw data. We use a hash table with 50% fill rate. We vary the size of the build table such that it produces a hash table of the desired size in the experiment. We vary the size of the hash table from 8KB to 1GB. The microbenchmark is the same as what past works use [9–11, 37].

In this section, we mainly focus on the probe phase which forms the majority of the total runtime. We discuss briefly the difference in execution with respect to build time at the end of the section. There are many hash table variants, in this section we focus on linear probing due to its simplicity and regular memory access pattern; our conclusions, however, should apply equally well to other probing approaches. Linear probing is an open addressing scheme that, to either insert an entry or terminate the search, traverses the table linearly until an empty bucket is found. The hash table is simply an array of slots with each slot containing a key and a payload but no pointers.

On the CPU side, we implemented three variants of linear probing. (1) `CPU Scalar` implements a scalar tuple-at-a-time join. The probing table is partitioned equally among the threads. Each thread iterates over its entries and for each entry probes the hash table to find a matching entry. On finding a match, it adds $A.v+B.v$ to its local sum. At the end, we add the local sum to the global sum using atomic instructions. (2) `CPU SIMD` implements vertical vectorized probing in a hash table [31]. The key idea in vertical vectorization is to process a different key per SIMD lane and use gathers to access the hash table. Assuming W vector lanes, we process W different input keys on each loop iteration. In every round, for the set of keys that have found their matches, we calculate their sum, add it to a local sum, and reload those SIMD lanes with new keys. (3) Finally, `CPU Prefetch` adds group prefetching to `CPU Scalar` [14]. For each loop iteration, software prefetching instructions are inserted to load the hash table entry that will be accessed a certain number of loop iterations ahead. The goal is to better hide memory latency at the cost of increased number of instructions.

On the GPU side, we implemented the join as follows. We load in a tile of keys and payloads from the probe side using `BlockLoad`; the threads iterate over each tile independently to find matching entries from the hash table. Each thread maintains a local sum of entries processed. After processing all entries in a tile, we use `BlockAgg` to aggregate the local sums within a thread block into a single value and increment a global sum with it.

Model: The probe phase involves making random accesses to the hash table to find the matching tuple from the build side. Every random access to memory ends up reading an entire cache line. However, if the size of hash table is small enough such that it can be cached, then random accesses no longer hit main memory and performance improves significantly. We model the runtime as follows:
1) If the hash table size is smaller than the size of the K^{th} level cache, we expect the runtime to be:

$$runtime = max(\frac{4\times2\times|P|}{B_r},(1-\pi_{K-1})(\frac{|P|\times C}{B_K}))$$

where $|P|$ is the cardinality of the probe table, B_r is the read bandwidth from device memory, C is the cache line size accessed on probe, B_K is the bandwidth of level K cache in which hash table fits and π_{K-1} is the probability of an access hitting a $K-1$ level cache. The first term is the time taken to scan the probe table from device memory. The second term is the time for probing the hash table. Note that each probe accesses an entire cache line. If the size of level K cache is S_K and size of the hash table is H, we define cache hit ratio $\pi_K = min(S_K/H, 1)$. The total runtime will be bounded by either the device memory bandwidth or the cache bandwidth. Hence, the runtime is the maximum of the two terms.
2) If the hash table size is larger than the size of the last level cache, we expect the runtime to be:

$$runtime = \frac{4\times2\times|P|}{B_r} + (1-\pi)(\frac{|P|\times C}{B_r})$$

where π is the probability that the accessed cache line is the last level cache.

Performance Evaluation: Figure 13 shows the performance evaluation of different implementations of Join. Both CPU and GPU variants exhibit step increase in runtime when the hash table size exceeds the cache size of a particular level. On the CPU, the step increases happen when the hash table size exceeds 256KB (L2 cache size) and 20MB (L3 cache size). On the GPU, the step increase happens when the hash table size exceeds 6MB (L2 cache size).

We see that `CPU SIMD` performs worse than `CPU Scalar`, even when the hash table is cache-resident. `CPU-SIMD` uses AVX2 instructions with 256-bit registers which represent 8 lanes of 32-bit integers. With 8 lanes, we process 8 keys at a time. However, a single SIMD gather used to fetch matching entries from the hash table can only fetch 4 entries at a time (as each hash table lookup returns an 8 byte slot. i.e., 4-byte key and 4-byte value, with 4 lookups filling the entire register). As a result, for each set of 8 keys, we do 2 SIMD gathers and then de-interleave the columns into to 8 keys and 8 values. This added overhead of extra instructions does not exist in the

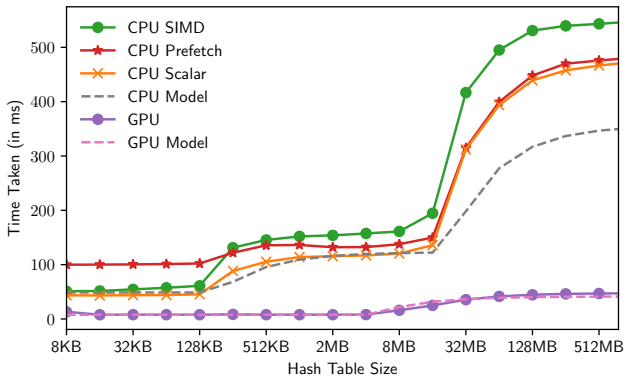


Figure 13: Join Performance.

scalar version. `CPU SIMD` is also brittle and not easy to extend to cases where hash table slot size is larger than 8 bytes. Note that past work has evaluated vertical vectorization with key-only build relations which do not exhibit this issue [27, 31]. Comparing `CPU Prefetch` to `CPU Scalar` shows that there is limited improvement from prefetching when data size is larger than the L3 cache size. When the hash table fits in cache, prefetching degrades the performs due to added overhead of the prefetching instructions.

Due the step change nature of the performance curves, the ratio of the runtimes varies based on hash table size. When the hash table size is between 32KB and 128KB, the hash table fits in L2 on both CPU and GPU. In this segment, we observe that the runtime is bound by DRAM memory bandwidth on CPU and L2 cache bandwidth on the GPU. The average gains are roughly 5.5× which is in line with the model. When the hash table size is between 1MB and 4MB, the hash table fits in the L2 on the GPU and in the L3 cache on the CPU. The ratio of runtimes in this segment is 14.5× which is the ratio of L2 cache bandwidth on GPU to the L3 cache bandwidth on the CPU. Finally when the hash table size is larger than 128MB, the hash table does not fit in cache on either GPU or CPU. The granularity of reads from global memory is 128B on GPU while on CPU it is 64B. Hence, random accesses into the hash table read twice the data on GPU compared to CPU. Given the bandwidth ratio is 16.2x, we would expect it as roughly 8.1x, however it is 10.5x due to memory stalls. The fact that actual CPU results are slower than `CPU Model` is because the model assumes maximum main memory bandwidth, which is not achievable as the hash table causes random memory access patterns.

Discussion: The runtime of the build phase in the microbenchmark shows a linear increase with size of the build relation. The build phase runtimes are less affected by caches as writes to hash table end up going to memory.

In this section, we modeled and evaluated the no partitioning join. Another variant of hash join is the partitioned hash join. Partitioned hash joins use a partitioning routine like radix partitioning to partition the input relations into cache-sized chunks and in the second step run the join on the corresponding partitions. Efficient radix-based hash join algorithms (*radix join*) have been proposed for CPUs [9–11, 14] and for the GPUs [35, 38]. Radix join requires the entire input to be available before the join starts and as a result intermediate join results cannot be pipelined. Hence, while radix join is faster for a single join, radix joins are not used for queries with multiple joins. While we do not explicitly model/evaluate radix joins, in the next section we discuss the radix partitioning routine that is the key component of such joins. That discussion shows that a careful radix partition implementation on both GPU and CPU are memory bandwidth bound, and hence the performance difference is roughly equal to the bandwidth ratio.

4.4 Sort

In this section, we evaluate the performance of sorting 32-bit key and 32-bit value arrays based on the key. According to literature, the fastest sort algorithm for this workload is the radix sort algorithm. We start by describing the Least-Significant-Bit (LSB) radix sort on the CPU [32] and on the GPU [28]. LSB radix sort is the fastest for the workload on the CPU. We describe why the LSB radix sort does poorly in comparison on the GPU and why an alternate version called Most-Significant-Bit (MSB) radix sort does better on the GPU [42]. We present a model for the runtime of the radix sort algorithm and then analyze the performance characteristics of radix partitioning on CPU vs GPU. Our implementations are primarily based on previous work but this is first time that these algorithm are compared to each other.

The LSB radix sort internally comprises a sequence of radix partition passes. Given an array A, radix r, and start bit a, a radix partition pass partitions the elements of the input array A into a contiguous array of 2^r output partitions based on value of r-bits $e[a:a+r)$ (i.e., radix) of the key e. Both on the CPU and GPU, radix partitioning involves two phases. In the first phase (*histogram phase*), each thread (CPU) / thread block (GPU) computes a histogram over its entries to find the number of entries in each partition of the 2^r partitions. In the second phase (*data shuffling phase*), each thread (CPU) / thread block (GPU) maintains an array of pointers initialized by the prefix sum over the histogram and writes entries to the right partition based on these offsets. The entire sorting algorithm contains multiple radix partition passes, with each pass looking at a disjoint sets of bits of the key e starting from the lowest bits $e[0:r)$ to highest bits $e[k-r:k)$ (where k is the bit-length of the key).

On the CPU, we use the implementation of Polychroniou et al. [32]. In the histogram phase, each thread makes one pass over its partition, for each entry calculating its radix value and incrementing the count in the histogram (stored in the L1 cache). For the shuffle phase, we first compute a

prefix sum over the histograms of all the threads (a 2D array of dimension $2^r \times t$ where t is the number of threads) to find the partition offsets for each of the threads. Next, each thread makes a pass over its partition using gathers and scatters to increment the counts in its offset array and writing to right offsets in output array. The implementation makes a number of optimizations to achieve good performance. Interested reader can refer to [32] for more details.

On the GPU, we implemented LSB radix sort based on the work of Merrill et al. [28]. In the histogram phase, each thread block loads a tile, computes a histogram that counts the number of entries in each partition, and writes it out to global memory. Prefix sum is used to find the partition offsets for each thread block in the output array. Next, in the shuffling phase each thread block reads in its offset array. The radix partitioning pass described above need to do stable partitioning i.e., ensures that for two entries with the same radix, the one occurring earlier in the input array also occurs earlier in the output array. Now on the GPU, in order to ensure stable partitioning for LSB radix sort we need to internally generate an offsets array for each thread from the the thread block offset array. For an r-bit radix partitioning, we need 2^r size histogram per thread. A number of optimizations have been proposed to store the histogram efficiently in registers, details of which are described in [28]. Due to restriction on number of registers available per thread, stable radix partitioning pass can only process 7-bits at a time.

Recently, Stehle et al. [42] presented an MSB radix sorting algorithm for the GPU. The MSB radix sort does not require stable partitioning. As a result, in the shuffle phase, we can just maintain a single offset array of size 2^r for the entire thread block. This allows MSB radix sort to process up to 8-bits at a time. Hence, the MSB radix sort to sort array of 32-bit keys with 4 passes each processing 8-bits at a time. On the other hand, LSB radix sort can processes only 7-bits at a time, and hence needs 5 radix partitioning passes processing 6,6,6,7,7 bits each.

Model: In the histogram phase, we read in the key column and write out a tiny histogram. The expected runtime is:

$$runtime_{histogram} = \frac{4 \times R}{B_r}$$

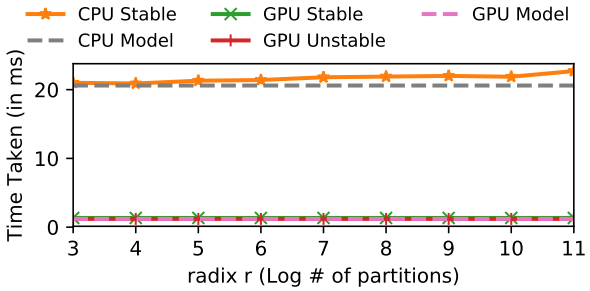
where R is the size of the input array and B_r is the read bandwidth. In the shuffle phase, we read both the key and payload column and at the end write out the radix partitioned key and payload columns. If the step is memory bandwidth bound, the runtime is expected to be:

$$runtime_{shuffle} = \frac{2 \times 4 \times R}{B_r} + \frac{2 \times 4 \times R}{B_w}$$

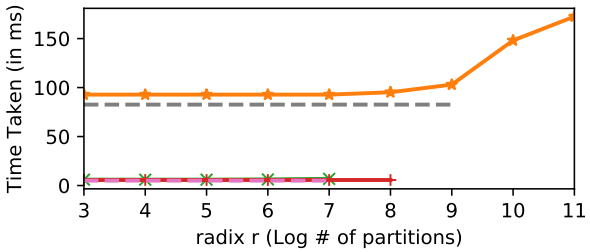
where B_w is the write bandwidth.

Performance Evaluation: We evaluate the performance of histogram and shuffle phase of the three variants: `CPU Stable` (stable partitioning on CPU), `GPU Stable` (stable partitioning on GPU), and `GPU Unstable` (unstable partitioning on GPU). We set the size of the input arrays at 256 million entries and vary the number of radix bits we partition on. Figure 14a shows the results for the histogram phase. Note that in the histogram phase there is no difference between `GPU Stable` and `GPU Unstable`. The histogram pass is memory bandwidth bound on both the CPU and GPU. Figure 14b shows the results for the shuffle phase. `GPU Stable` is able to partition up to 7-bits at a time whereas `GPU Unstable` is able to partition 8-bits at a time. `CPU Stable` is able to partition up to 8-bits a time while remaining bandwidth bound. Beyond 8-bits, the size of the partition buffers needed exceeds the size of L1 cache and the performance starts to deteriorate.

Now that we have the radix partitioning passes, we look at the sort runtime. On the CPU, we use stable partitioning to implement LSB radix sort. It ends up running 4 radix partitioning passes each looking at 8-bits at time. On the GPU, MSB radix sort also sorts the data with 4 passes each processing 8-bits at a time. The time taken to sort 2^{28} entries is $464ms$ on the CPU and $27.08ms$ on the GPU. The runtime gain is 17.13× which is close to the bandwidth ratio of 16.2×.



(a) Radix histogram on CPU and GPU



(b) Radix shuffling on CPU and GPU

Figure 14: Sort Microbenchmark

```
for each y in R:
  if y > v:
    output[i++] = v
```

(a) With branching

```
for each y in R:
  output[i] = y
  i += (y > v)
```

(b) With predication

Figure 15: Implementing selection scan

5 WORKLOAD EVALUATION

Now that we have a good understanding of how individual operators behave on both CPU and GPU, we will evaluate the performance of a workload of full SQL queries on both hardware platforms. We first describe the query workload we use in our evaluation. We then present a high-level comparison of the performance of queries running on GPU implemented with the tile-based execution model versus our own equivalent implementation of the queries on the CPU. We also report the performance of Hyper [29] on CPU and Omnisci [6] on the GPU which are both state-of-the-art implementations. As a case study, we provide a detailed performance breakdown of one of the queries to explain the performance gains. Finally, we present a dollar-cost comparison of running queries on CPU and GPU.

We use two platforms for our evaluation. For experiments run on the CPU, we use a machine with a single socket Skylake-class Intel i7-6900 CPU with 8 cores that supports AVX2 256-bit SIMD instructions. For experiments run on the GPU, we use an instance which contains an Nvidia V100 GPU. We measured the bidirectional PCIe transfer bandwidth to be 12.8GBps. More details of the two instances are shown in Table 2. Each system is running on Ubuntu 16.04 and the GPU instance has CUDA 10.0. In our evaluation, we ensure that data is already loaded into the respective device's memory before experiments start. We run each experiment 3 times and report the average measured execution time.

5.1 Workload

For the full query evaluation, we use the Star Schema Benchmark (SSB) [30] which has been widely used in various data analytics research studies [15, 24, 44, 48]. SSB is a simplified version of the more popular TPC-H benchmark. It has one fact table *lineorder* and four dimension tables *date, supplier, customer, part* which are organized in a star schema fashion. There are a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments we run the benchmark with a scale factor of 20 which will generate the fact table with 120 million tuples. The total dataset size is around 13GB.

5.2 Performance Comparison

In this section, we compare the query runtimes of benchmark queries implemented using block-wide functions on the GPU (`Standalone GPU`) to an equivalent efficient implementation of the query on the CPU (`Standalone CPU`). We also compare against Hyper (`Hyper`), a state-of-the-art OLAP DBMS and Omnisci (`Omnisci`), a commercial GPU-based OLAP DBMS.

In order to ensure a fair comparison across systems, we dictionary encode the string columns into integers prior to data loading and manually rewrite the queries to directly reference the dictionary-encoded value. For example, a query with predicate `s_region = 'ASIA'` is rewritten with predicate `s_region = 2` where 2 is the dictionary-encoded value of `'ASIA'`. Some columns have a small number of distinct values and can be represented/encoded with 1-2 byte values. However, in our benchmark we make sure all column entries are 4-byte values to ensure ease of comparison with other systems and avoid implementation artifacts. Our goal is to understand the nature of the performance gains of equivalent implementations on GPU and CPU, and not to achieve best storage layout. We store the data in columnar format with each column represented as an array of 4-byte values. On the GPU, we use a thread block size of 256 with tile size of 2056 (= 8×256) resulting in 8 entries per thread per tile.

Platform	CPU	GPU
Model	Intel i7-6900	Nvidia V100
Cores	8 (16 with SMT)	5000
Memory Capacity	64 GB	32 GB
L1 Size	32KB/Core	16KB/SM
L2 Size	256KB/Core	6MB (Total)
L3 Size	20MB (Total)	-
Read Bandwidth	53GBps	880GBps
Write Bandwidth	55GBps	880GBps
L1 Bandwidth	-	10.7TBps
L2 Bandwidth	-	2.2TBps
L3 Bandwidth	157GBps	-

Table 2: Hardware Specifications

Figure 16 shows the results. Comparing `Standalone CPU` to `Hyper` shows that the former does on an average 1.17x better than the latter. We believe `Hyper` is missing vectorization opportunities and using a different implementation of hash tables. The comparison shows that our implementation is a fair comparison and it is quite competitive compared to a state-of-the-art OLAP DBMS. We also compared against MonetDB [12], a popular baseline for many of the past works on GPU-based databases. We found that the `Standalone CPU` is on an average 2.5× faster than MonetDB. We did not include it in the figure as it made the graph hard to read. We also tried to compare against Pelaton with relaxed-operator fusion [27]. We found that the system could not load the scale factor 20 dataset. Scaling down to scale factor 10, its queries were significantly slower (>5×) than Hyper or our approach.

Comparing `Standalone GPU` to `Omnisci`, we see that our GPU implementation does significantly better than `Omnisci` with an average improvement of around 16×. Both methods run with the entire working set stored on the GPU. `Omnisci` treats each GPU thread as an independent unit. As a result, it does not realize benefits of blocked loading and better GPU utilization got from using the tile-based model. The comparison of `Standalone GPU` against `Omnisci` and `Standalone CPU` to `Hyper` serve as a sanity check and show that our query implementations are quite competitive.

Comparing `Standalone GPU` to `Standalone CPU`, we see that the `Standalone GPU` is on average 25× faster than the CPU implementation. This is higher than the bandwidth ratio

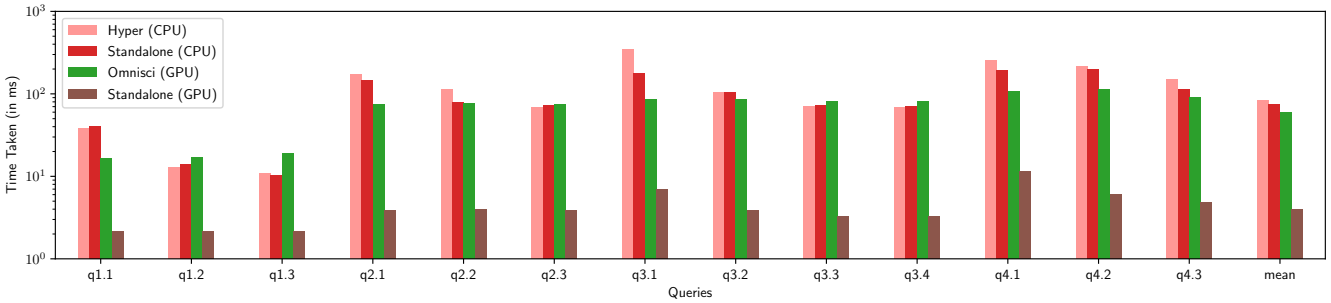


Figure 16: Star Schema Benchmark Queries

```
SELECT SUM(lo_revenue) AS revenue, d_year, p_brand
FROM lineorder, date, part, supplier
WHERE lo_orderdate = d_datekey
AND lo_partkey = p_partkey AND lo_suppkey = s_suppkey
AND p_category = 'MFGR#12' AND s_region = 'AMERICA'
GROUP BY d_year, p_brand
```

Figure 17: SSB Query 2.1

of 16.2. This is surprising given that in Section 4 we saw that individual query operators had a performance gain equal to or lower than the bandwidth ratio. The key reason for the performance gain being higher than the bandwidth ratio is the better latency hiding capability of GPUs. To get a better sense for the runtime difference, in the next subsection we discuss models for the full SQL queries and dive into why architecture differences leads to significant difference in performance gain from the bandwidth ratio.

5.3 Case Study

The queries in the Star Schema Benchmark can be broken into two sets: 1) the query flight $q1.x$ consists of queries with selections directly on the fact table with no joins and 2) the query flights $q2.x$, $q3.x$, $q4.x$ consist of queries with no selections on fact table and multiple joins — some of which are selective. In this section, we analyze the behavior q2.1 in detail as a case study. Specifically, we build a model assuming the query is memory-bandwidth bound, derive the expected runtime based on the model, compare them against the observed runtime, and explain the differences observed.

Figure 17 shows the query: it joins the fact table `lineorder` with 3 dimension tables: `supplier`, `part`, and `date`. The selectivity of predicates on `p_category` and `s_region` are $1/25$ and $1/5$ respectively. The subsequent join of `part` and `supplier` have the same selectivity. We choose a query plan where `lineorder` first joins `supplier`, then `part`, and finally `date`, this plan delivers the highest performance among the several promising plans that we have evaluated.

The cardinalities of the tables `lineorder`, `supplier`, `part`, and `date` are $120M$, $40k$, $1M$, and $2.5k$ respectively. The query runs build phase for each of the 3 joins to build their respective hash tables. Then a final probe phase runs the joins pipelined. Given the small size of the dimension tables, the build time is much smaller than the probe time, hence we focus on modeling the probe time. On the GPU, each thread block processes a partition of the fact table, doing each of the 3 joins sequentially and updating a global hash table at the end that maintains the aggregate. Past work [26] has shown that L2 cache on the GPU is an LRU set associative cache. Since hash tables associated with the `supplier` and `date` table are small, we can assume that they remain in the L2 cache. The size of the `part` hash table is larger than L2 cache. We model the runtime as consisting of 3 components:
1) The time taken to access the columns of the fact table:

$$r_1 = (\frac{4|L|}{C} + min(\frac{4|L|}{C}, |L|\sigma_1) + min(\frac{4|L|}{C}, |L|\sigma_1\sigma_2) + min(\frac{4|L|}{C}, |L|\sigma_1\sigma_2)) \times \frac{C}{B_r}$$

where σ_1 and σ_2 are join selectivities associated with join with `supplier` and `part` tables respectively, $|L|$ is the cardinality of the `lineorder` table, C is size of cache line, and B_r is the global memory read bandwidth. For each column except the first, the number of cache lines accessed is the minimum of: 1) accessing all cache lines of the column ($\frac{4|L|}{C}$) and 2) accessing a cache line per entry read ($|L|\sigma$).
2) Time taken to probe the join hash tables:

$$r_2 = (2\times|S| + 2\times|D| + (1-\pi)(|L|\sigma_1)) \times \frac{C}{B_r}$$

where $|S|$ and $|D|$ are cardinalities of the `supplier` and `date` table, $(|L|\sigma_1)$ represents the number of lookups into the `part`hash table and π is the probability of finding the `part` hash table lookup in the L2 cache.
3) Time taken to read and write to the result table:

$$r_3 = |L|\sigma_1\sigma_2 \times \frac{C}{B_r} + |L|\sigma_1\sigma_2 \times \frac{C}{B_w}$$

The total runtime on GPU is $r_1 + r_2 + r_3$. The key difference with respect to CPU is that on the CPU, all three hash tables fit in the L3 cache. Hence for CPU, we would have $r_2 = (2\times|S| + 2\times|D| + 2\times|P|)$. To calculate π, we observe that the size of the `part` hash table (with perfect hashing) is $2\times4\times1M = 8MB$. With the `supplier` and `date` table in cache, the available

cache space is 5.7MB. Hence the probability of `part` lookup in L2 cache is $\pi = 5.7/8$. Plugging in the values we get the expected runtimes on the CPU and GPU as 47 ms and 3.7 ms respectively compared to actual runtime of 125 ms and 3.86 ms.

We see that the model predicted runtime on the GPU is close to the actual runtime whereas on the CPU, the actual runtime is higher than the modeled runtime. This is in large part because of the ability of GPUs to hide memory latency even with irregular accesses. SIMT GPUs run scalar code, but they "tie" all the threads in a *warp* to execute the same instruction in a cycle. For instance, gathers and scatter are written as scalar loads and stores to non-contiguous locations. In a way, CPU threads are similar to GPU warps and GPU threads are similar to SIMD lanes. A key difference between SIMT model on GPU vs SIMD model on CPU is what happens on memory access. On the CPU, if a thread makes a memory access, the thread waits for the memory fetch to return. If the cache line being fetched is not in cache, it leads to a memory stall. CPU have prefetchers to remedy this, but prefetchers do not work well with irregular access patterns like join probes. On the GPU, a single streaming multiprocessor (SM) usually has 64 cores that can execute 2 warps (64 threads) at any point. However, the SM can keep > 2 warps active at a time. On Nvidia V100, each SM can hold 64 warps in total with 2 executing at any point in time. Any time a warp makes a memory request, the warp is swapped out from execution into the active pool and another warp that is ready to execute ends up executing. Once the memory fetch returns, the earlier warp can resume executing at the next available executor cores. This is similar to swapping of threads on disk access on CPU. This key feature allows GPUs to avoid the memory stalls associated with irregular accesses as long as enough other threads are ready to execute. Modeling query performance of multi-join queries on CPUs is an interesting open problem which we plan to address as future work.

5.4 Cost Comparison

The paper has so far demonstrated that GPUs can have superior performance than CPUs for data analytics. However, GPUs are known to be more expensive than CPUs in terms of cost. Table 3 shows both the purchase and renting cost of CPU and GPU that match the hardware used in this paper (i.e., Table 2). For renting costs, we use the cost of EC2 instances provided by Amazon Web Services (AWS). For CPU, we choose the instance type `r5.2xlarge` which contains a modern Skylake CPU with 8 cores, with a cost of $0.504 per hour. For GPU, we choose the instance type `p3.2xlarge` whose specs are similar to `r5.2xlarge` plus it has an Nvidia V100 GPU, with a cost of $3.06 per hour. The cost ratio of the two systems is about 6×. For purchase costs, we compare the estimate of a single socket server blade to the same server blade with one Nvidia V100 GPU. The cost ratio of the two systems at the high end is less than 6×. The average performance gap, however, is about 25× according to our evaluation (cf. Section 5.2), which leads to a factor of 4 improvement in cost effectiveness of GPU over CPU. Although the performance and cost will vary a lot across different CPU and GPU technologies, the ratio between the two will not change as much. Therefore, we believe the analysis above should largely apply to other hardware selection.

	Purchase Cost	Renting Cost
CPU	$2-5K	$0.504 per hour
GPU	$CPU + 8.5K	$3.06 per hour

Table 3: Purchase and renting cost of CPU and GPU.

5.5 Discussion

In this paper, we showed through our model-based analysis and empirical evaluation that there is limited gain from using GPUs as a coprocessor and that the runtime gain from running queries on the GPU vs CPU is $1.5x$ the bandwidth ratio of the two devices. We believe that these results should help pivot the community towards treating GPUs as primary execution engine. However, this paper largely focused on using a single GPU, which has limited memory capacity. There are many challenges that need to be addressed before GPUs have widespread adoption that were beyond the scope of this paper and make for exciting future work:

- **Distributed+Hybrid** It is possible to attach multiple GPUs onto a single machine that can greatly increase the aggregated HBM memory capacity. These machines will also having significant CPU memory. Executing queries on this heterogeneous system is still an open problem.
- **Compression** Data compression could be used to fit more data into GPU's memory. GPUs have higher compute to bandwidth ratio than CPUs which could allow use of non-byte addressable packing schemes.
- **Strings/Non-Scalar Data Types** Handling arbitrary strings and array data types efficiently on GPUs is still an open problem.

6 CONCLUSION

This paper compared CPUs and GPUs on database analytics workloads. We demonstrated that running an entire SQL query on a GPU delivers better performance than using the GPU as an accelerator. To ease implementation of high-performance SQL queries on GPUs, we developed Crystal, a library supporting a tile-based execution model. Our analysis on SSB, a popular analytics benchmark, shows that modern GPUs are 25× faster and 4× more cost effective than CPUs. This makes a strong case for using GPUs as the primary execution engine when the dataset fits into GPU memory.

REFERENCES

[1] 1000X faster data exploration with GPUs. https://www.omnisci.com/blog/mapd.

[2] BlazingDB. https://blazingdb.com.

[3] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500. https://bit.ly/2UcBInt.

[5] Kinetica. https://kinetica.com/.

[6] OmniSci. https://omnisci.com.

[7] Opencl. https://www.khronos.org/opencl/.

[8] Thrust. https://thrust.github.io/.

[9] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[10] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

[11] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.

[12] P. A. Boncz et al. *Monet: A next-generation DBMS kernel for query-intensive applications.* Universiteit van Amsterdam [Host], 2002.

[13] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17, 2007.

[15] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1603–1618. ACM, 2018.

[16] N. Govindaraju et al. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[17] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[18] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, 2008.

[19] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB*, 2013.

[20] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 2013.

[21] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[22] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *DaMoN*, 2012.

[23] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *In Memory Data Management and Analysis*, pages 3–14. Springer, 2015.

[24] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[25] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *Proceedings of the VLDB Endowment*, 9(3):231–246, 2000.

[26] X. Mei and X. Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[27] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[28] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[29] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[30] 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.

[31] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[32] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014.

[33] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[34] K. A. Ross. Selection conditions in main memory. *ACM Transactions on Database Systems (TODS)*, 29(1):132–161, 2004.

[35] R. Rui and Y.-C. Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, page 17. ACM, 2017.

[36] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362. ACM, 2010.

[37] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1961–1976. ACM, 2016.

[38] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on gpus. Technical report, 2019.

[39] E. A. Sitaridi and K. A. Ross. Ameliorating memory contention of olap operators on gpu processors. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 39–47. ACM, 2012.

[40] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on gpus. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, page 4. ACM, 2013.

[41] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM, 2011.

[42] E. Stehle and H.-A. Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *SIGMOD*. ACM, 2017.

[43] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[44] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.

[45] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *European Conference on Parallel Processing*, pages 160–169. Springer, 2011.

[46] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012.

[47] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima. Relational joins on gpus: A closer look. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2663–2673, 2017.

[48] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *PVLDB*, 2013.