Toward GPUs being mainstream in analytic processing

An initial argument using simple scan-aggregate queries

Jason Power
powerjg@cs.wisc.edu

Yinan Li
yinan@cs.wisc.edu

Mark D. Hill
markhill@cs.wisc.edu

Jignesh M. Patel
jignesh@cs.wisc.edu

David A. Wood
david@cs.wisc.edu

Department of Computer Sciences
University of Wisconsin–Madison

ABSTRACT

There have been a number of research proposals to use discrete graphics processing units (GPUs) to accelerate database operations. Although many of these works show up to an order of magnitude performance improvement, discrete GPUs are not commonly used in modern database systems. However, there is now a proliferation of *integrated* GPUs which are on the same silicon die as the conventional CPU. With the advent of new programming models like heterogeneous system architecture, these integrated GPUs are considered first-class compute units, with transparent access to CPU virtual addresses and very low overhead for computation offloading. We show that integrated GPUs significantly reduce the overheads of using GPUs in a database environment. Specifically, an integrated GPU is 3× faster than a discrete GPU even though the discrete GPU has 4× the computational capability. Therefore, we develop high performance scan and aggregate algorithms for the integrated GPU. We show that the integrated GPU can outperform a four-core CPU with SIMD extensions by an average of 30% (up to 3.2×) and provides an average of 45% reduction in energy on 16 TPC-H queries.

1. INTRODUCTION

To continue scaling performance at past rates, computer architects are creating general-purpose commodity hardware accelerators that work with the CPU. One such example is general-purpose graphics processing units (GPGPUs). Figure 1 shows how GPU and CPU performance has scaled over the past seven years. This figure shows that data-parallel architectures, like the GPU, are scaling performance at a faster rate than conventional CPUs. By leveraging data-parallelism, GPUs and other data-parallel architectures have less overhead per processing element, leading to increased scalability.

Scans and aggregates are data-parallel operations that are strong candidates for offloading to the GPU, and in this work we make our argument by focusing our attention on scan-aggregate queries in in-memory settings. Scans are an important primitive and the workhorse in high-performance in-memory database systems like SAP HANA [6, 29], Oracle Exalytics [8], IBM DB2 BLU [22] and Facebook's Scuba [1]. A series of scan algorithms have been developed in the database community to exploit this parallelism using hardware artifacts such as the parallelism within regular ALU words (e.g., [17, 23]), and SIMD (short vector units) to accelerate scans (e.g., [4, 14, 17, 23, 28, 29, 31]). The GPU hardware is designed for highly data-parallel workloads like scan-aggregate queries and we show that it can provide higher performance and higher energy efficiency than CPUs. As more processors include integrated on-die GPUs, it will become important for database systems to take advantage of these devices.

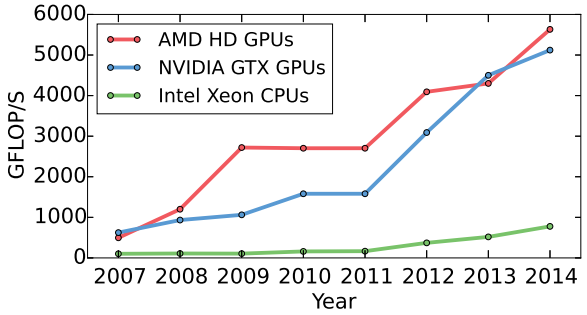


Figure 1: Comparison of CPU and discrete GPU performance improvement over the last seven years. Data from various sources and Rupp [25]

Because of the GPU's potential for increased performance, there has been work accelerating database operations with *discrete GPUs* (e.g., [9, 10, 15, 26]). However, mainstream database systems still do not commonly use GPGPUs. Unfortunately there are many overheads associated with discrete GPUs that negate many of their potential benefits. Due to the limited memory capacity of discrete GPUs, when accessing large data sets, the data must be copied across the relatively low bandwidth (16 GB/s) PCIe interface. This data copy time can be up to 98% of the total time to complete a scan using the discrete GPU. Additionally, applications must frequently access the operating system-level device driver to coordinate between the CPU and the GPU, which incurs significant overhead. Many previous works have discounted the overhead of copying the data from the

DaMoN '15, June 1, 2015, Melbourne, VIC, Australia

http://dx.doi.org/10.1145/2771937.2771941

CPU memory, which results in optimistic performance predictions. We find that by including these overheads, the discrete GPU can be 3× slower than a multicore CPU. This result shows that although discrete GPUs may seem a good fit for performing scans, due to their limited memory capacity they are not practical today. Not surprisingly, GPUs are not mainstream today in data analytics environments. However, we argue that this situation is likely to change due to recent GPU hardware trends.

Today, many systems are integrating the GPU onto the same silicon die as the CPU, and these *integrated GPUs* with new programming models reduce the overheads of using GPGPUs in database systems. These integrated GPUs share both physical and virtual memory with the CPU. With new GPGPU APIs like heterogeneous system architecture (HSA) [24], integrated GPUs can transparently access all of the CPUs' memory, greatly simplifying application programming (see Figure 3b and Section 2.1). HSA reduces the programming overhead of using GPUs because the CPU and GPU share a single copy of the application data, making it possible to make run-time decisions to use the CPU or the GPU for all or part of the query. As integrated GPUs become more common, it will be important for database systems to take advantage of the computational capability of the silicon devoted to GPUs. Furthermore, GPGPUs have become far easier to program, making it more economical to write and maintain specialized GPU routines.

We implement an in-memory GPU database system by leveraging the fast-scan technique BitWeaving [17] and the database denormalization technique WideTable [18], and a new method to compute aggregates on GPUs efficiently. Using this implementation, we show the benefits of our approach for an important, but admittedly limited, class of scan-aggregate queries. We find that the integrated GPU can provide a speedup of as much as 2.3× on scans and up to 3.8× on aggregates. We also evaluate our algorithms on 16 TPC-H queries (using the WideTable technique) and find that by combining the aggregate and scan optimizations, the integrated GPU can increase performance by an average of 30% (up to 3.2×) and decrease energy by 45% on average over a four-core CPU. Thus, we conclude that it is now practical for database systems to actually deploy methods for GPUs in production settings for in-memory scan-aggregate queries. With the proliferation of integrated GPUs, ignoring this computational engine may leave significant performance on the table.

The remainder of this paper is organized as follows. Section 2 presents a background on GPUs and a new programming API, HSA. Section 3 details the implementation of our scan and aggregate algorithms for the integrated GPU. Section 4 presents our experimental methodology, and Section 5 describes the performance of our system on TPC-H queries as well as scan and aggregate microbenchmarks. Finally, we discuss the impact of future architectures on our work in Section 6, and Section 8 concludes.

2. GPU BACKGROUND

Graphics processing units (GPUs) have recently become more easily programmable, creating the general purpose GPU (GPGPU) computing landscape. There are two key characteristics that differentiate GPUs from CPUs. GPUs have an order of magnitude more execution units (e.g., an AMD A10-7850 has 4 CPU cores each with 5 execution units, and an AMD HD7970 discrete GPU has 32 compute units each with 64 execution units [3]), and GPUs provide much higher memory bandwidth (e.g., an AMD A10-7850 has a memory bandwidth up to 34 GB/s [2], and an AMD HD7970 has a memory bandwidth of 264 GB/s [3]). In this paper, we investigate scan performance on three different architectures: a multicore CPU, a high-end discrete GPU, and a modern integrated GPU. Below we discuss three important details of GPGPU microarchitecture.

First, GPGPUs employ very wide data-parallel hardware. An AMD HD7970 can operate on 131,072 bits in parallel (32 compute units (CUs) × 4 SIMD lanes × 16-word SIMD × 64-bit words). Compare this to the data parallelism that is available in CPUs or SIMD, which is many orders of magnitude smaller (SIMD registers today are 256 bits wide). Thus, there is a potentially higher level of data parallelism that is available in each cycle with a GPU.

Second, GPGPUs are programmed with SIMT (single-instruction multiple-thread) instead of SIMD (single-instruction multiple-data). The SIMT model significantly simplifies programming in very wide data-parallel hardware of the GPU. For instance, SIMT allows arbitrary control flow between individual SIMD lanes.

Finally, and importantly, GPU architecture can be more energy-efficient than CPU architecture for certain workloads (e.g., database scans). Since many SIMD lanes share a single front-end (instruction fetch, decode, etc.), this per-instruction energy overhead is amortized. On CPU architectures, the execution front-end and data movement consumes 20–40× more energy than the actual instruction [16]. Additionally, all of the parallelism is explicit for GPUs through the programming model, while CPUs require high energy hardware (like the re-order buffer and parallel instruction issue) to implicitly generate instruction-level parallelism, which wastes energy for data-parallel workloads.

2.1 Integrated GPU APIs: HSA

New GPGPU APIs simplify programming and increase performance of GPGPU computing. Until very recently, GPGPU APIs were designed for discrete GPUs. Communication between the CPU and the GPU was high overhead, and all data was explicitly declared and copied before it was used on the GPU. However, while GPGPUs are becoming more physically integrated with CPUs, they are also becoming more logically integrated. For instance the runtime we use in this work, heterogeneous system architecture (HSA), provides a coherent and unified view of memory [24].

In HSA, programming the integrated GPU is greatly simplified. For instance, after the memory is allocated and initialized in the CPU memory space, the GPU kernel is launched as if it were a CPU function. The pointers created for the CPU can be referenced from the GPU. Thus, after the kernel completes, there is no need to copy data back to the CPU memory space. This unified view of memory also enables the GPU to access complicated pointer-based data structures, like hashtables. Section 3.1 discusses how this support facilitates implementing group-by operations on the integrated GPU.

In addition to simplifying programming, HSA allows the database system to decide to use the CPU or GPU on-the-fly. Since all of the data structures can be transparently shared between the two devices, the decision to offload to the GPU is easier. There is no need to translate data structures and

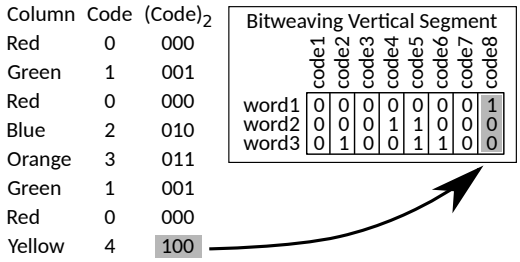


Figure 2: Overview of BitWeaving's memory layout assuming an 8-bit word. The actual implementation uses 128-bit word for the SIMD CPU and 16,384-bit word for the GPU.

high-overhead data copies are no longer required.

The HSA programming model shares some aspects with other programming models that target discrete GPUs like NVIDIA's unified virtual addressing [19]. In both of these programming models, the CPU and GPU shared the same virtual addresses. However, HSA is designed specifically for integrated accelerators. Two advantages of this design are that HSA never needs to copy data from one physical address space to another, and HSA does not require the programmer to explicitly allocate memory for use on the GPU. These characteristics allows HSA to have higher performance and a simpler programming interface than the discrete GPU programming models.

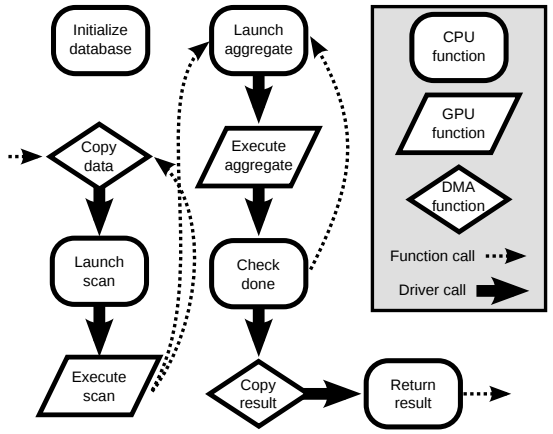
3. IMPLEMENTATION

To study scan-aggregate query processing on GPUs, we leverage previous work accelerating analytical scan query processing on the CPU: i.e., we use the BitWeaving scan algorithm [17]. BitWeaving uses a coded columnar layout, packing multiple codes per word. The output of the scan is a bitvector where each 1 bit corresponds to a matching row.

In this work, we use the BitWeaving/V scan algorithm. This algorithm encodes columns using order-preserving dictionaries and then stores the encoded values for a column grouped by the bit position. At a high-level, BitWeaving/V can be thought of as a column store at the bit-level with algorithms that allow evaluating traditional SQL scan predicates in the native bit-level columnar format using simple bitwise operations, such as XOR, AND, and binary addition. (See [17] for details). Figure 2 shows an example of a BitWeaving/V representation for a sample column.

The BitWeaving/V method needs small modifications to execute efficiently on GPUs. The largest change to adapt BitWeaving to the GPU is to increase the underlying word size to the logical SIMD width of the GPU. Thus, instead of packing coded values into one 64-bit word as in CPU algorithms, or 128- or 256-bit words in SIMD scan implementations, the GPU scan implementation uses a logical 16,384-bit word (256 consecutive 64-bit words).

For the discrete GPU, Figure 3a shows an overview of how the CPU and the GPU interact using OpenCL—the legacy GPU API. After the database is initialized, when a query arrives, the columns which the query references must be copied to the device, which is a DMA call. Next, the scan is launched to the GPU, which requires a high-latency user-mode to kernel-mode switch and high-latency PCIe communication (shown with bold arrow in Figure 3a). Then, the scan is actually executed on the GPU, which may take significantly less time than the previous operations. Once the (possibly many) scan operations have completed, the aggregate operation is launched in a similar fashion incurring the high OS kernel overheads. Finally, the result must be copied back to the CPU memory again paying the OS kernel and PCIe latencies.



(a) OpenCL flow for the discrete GPU.

(b) HSA flow for the integrated GPU.

Figure 3: Flowchart showing the CPU-GPU interaction.

Performing this scan on the discrete GPU requires many calls to the operating system's kernel display driver, shown as bold arrows in Figure 3a. These driver calls incur significant overhead for short running GPU jobs such as column scans. Additionally, since the GPU cannot hold the entire database in its memory, each column must be copied to the GPU before the scan kernel can begin. As most queries require scans over multiple columns, this scan loop is repeated many times.

However, most of these overheads are eliminated on integrated GPUs. Figure 3b shows an overview of how the CPU and the GPU interact in our system using HSA. With HSA, integrated GPUs do not interface through the OS kernel driver. Instead, they use user-mode queues which provide low-latency to offload work to the GPU [24]. Therefore, the overhead to use the integrated GPU is simply a few writes to coherent memory with no driver involvement. Additionally, since the integrated GPU shares memory with the CPU, the copies used in the OpenCL version are eliminated. As Figure 3b shows, executing a query on the integrated GPU with HSA only requires a few function calls similar to how the CPU executes the query.

3.1 Aggregate computation

Scan queries generally tend to have aggregation operations in analytic environments, and the aggregate component can dominate the overall query execution time in some cases. In the simplified, scan-based queries that we evaluate in this paper, a significant percentage of the time is spent in aggregates, an average of 80% across the queries that we use in this paper. In fact, we find that some queries spend over

99% of their execution time in the aggregate phase.

Since aggregate performance can dominate the overall query performance in some scan-aggregate queries, we investigate offloading the aggregate computation to the GPU (in addition to offloading the scan computation). Tightly-integrated programming models like HSA 2.1 significantly decrease the complexity of offloading the aggregate computation to the GPU. As discussed below, parts of the aggregate computation are more efficiently performed on the CPU and other parts are more efficiently performed on the GPU. It is easier to take advantage of this fine-grained offloading with current integrated GPUs and the HSA programming model.

In many modern systems, the output of a selection operation produces a result bitvector. In this bitvector, a bit value of 1 indicates the the tuple at the corresponding index position was selected by that operation. To actually fetch the data for the selected tuple, the index value must be converted to an absolute offset that points to the actual memory location of the selected tuple. Instead of evaluating every valid bit sequentially, we batch many valid column offsets and call the computational function only once, as shown in Algorithm 1. We split each column into blocks of 2^{20} or about one million codes. This approach optimizes the aggregate primitive in two ways. This algorithm gives much better memory locality when searching for valid bits in the bitvector, and by batching, we decrease the overhead of function calls on the CPU and launching jobs to the GPU.

Algorithm 1 Aggregate algorithm

Require: *bitvector* that encodes matching tuples
split into 2^{20}-bit blocks
Require: *reduce* functor to compute the aggregate
$offsets$ = [] // Indicies of matching tuples
2: **for** each *bv_block* in *bitvector* **do**
for each *word* in *bv_block* **do**
4: **for** each true bit in *word* **do**
$offsets$.append($word\_index * 64 + bit\_index$)
6: // call functor for group-by/agg.
reduce($offsets$) // Offloaded to integrated GPU
8: clear($offsets$)

This algorithm takes a bitvector with the selected tuples and a functor (see Algorithm 2 and 3) that is the actual aggregate computation. The bitvector is split into blocks which are each operated on in serial. For each word (8-bytes) of the bitvector block, we search for valid bits (lines 2–4). For each bit that is 1, which corresponds to a selected tuple, we insert the absolute offset into the column into an $offsets$ array. Then, we execute the user-provided functor which computes the aggregate given the computed $offsets$ for that column block.

We find that it is more efficient to use the CPU to compute the offsets than the GPU. When creating the $offsets$ array, many threads may simultaneously try to add new offsets, and it is inefficient for the GPU to coordinate between these threads [5]. Therefore, when performing aggregates on the GPU, we first compute the offsets on the CPU and only perform the reduction operation on the GPU.

Many scan-aggregate queries include group-by operations in their aggregate phase. The number of groups in the group-by statements vary greatly, from a single group (i.e., a simple reduction) to millions of different groups. For group-by operations with multiple columns the number of groups can be even larger.

We investigate two different algorithms to compute group-by aggregate: direct storage (Algorithm 2) and hashed storage (Algorithm 3). The direct storage algorithm allocates space for every group and directly updates the corresponding values. The hashed storage algorithm uses a hash of the group ID instead and does not require a one-to-one mapping. The hash-based algorithm is required when there are many groups or the groups are sparse as the direct storage method would use too much memory.

Algorithm 2 Direct group-by (reduce with direct storage)

Require: $offsets$ a list of tuple indicies
1: // array for storage of aggregate value for each group
2: static *aggregates*[*num_groups*]
3: **for** each $offset$ in $offsets$ **do**
4: // Materialize the column data and calculate *group_id*
5: // Perform aggregate based on *op*
6: *aggregates*[*group_id*] (*op*) *data*

In the direct group-by algorithm, we store the aggregate values of each group directly in an array (*aggregates* on line 2). These values are stored statically to persist across multiple instances of this function. For each offset in the provided $offsets$ array, we look up the value contained in the tuple at that offset, which may result in accessing multiple columns. Then, we perform the aggregate operation (*op* on line 6) to update the stored value of the aggregate. To perform averages and other similar operations, we can also count the number of matches in a similar fashion.

Algorithm 3 Hashed group-by (reduce with hashtable)

Require: $offsets$ a list of tuple indicies
1: static *hashtable*(*predicted_size*)
2: **for** each $offset$ in $offsets$ **do**
3: // Materialize the column data and calculate *group_id*
4: *entry* = *hashtable*.get(*group_id*) // get group entry
5: // Perform aggregate based on *op*
6: *entry.value* (*op*) *data*

In the hashed group-by algorithm, the values for each group's aggregate are stored in a hashtable instead of in a simple array. We can use profiling to predict the size for the hashtable (line 1). The hashed algorithm performs the same steps as the above direct algorithm, except it acts on values in the hash entries instead of the direct storage array. The hash table is implemented with linear probing and uses the MurmurHash function.

To use these algorithms in a parallel environment, we use synchronization to guard the reduction variables. The GPU does not have mature support for inter-thread communication. Therefore, we use lock-free algorithms for the hashtable in Algorithm 3 and a local reduction tree for the direct storage algorithm (Algorithm 2).

4. METHODOLOGY

There is a large space of potential hardware to run a database system. For a constant comparison point, we use AMD CPU and GPU platforms in our evaluation. We use a four-core CPU and two different GPUs, an integrated GPU that

is on the same die as the CPU, and a discrete GPU connected to the CPU via the PCIe bus. Table 1 contains the details of each architecture.

For power measurements, we use the full-system power measured with a WattsUp meter. The WattsUp meter is connected between the system under test and the power supply (wall jack). Since all of our runs last for minutes, we use the energy reported by the WattsUp meter as our primary metric for energy usage.

All of the CPU results are using the BitWeaving algorithm [17] with SIMD instructions. For the results presented for multiple CPU cores, we used OpenMP to parallelize the scan and the aggregate functions. Each function performs a highly data-parallel computation on the entire column of data. Thus, the fork-join parallelism model of OpenMP is appropriate.

We use the HSA programming interface described in Section 2.1 for the integrated GPU case. The GPU kernels (scan, simple aggregate, direct- and hashed-group-by) are written in OpenCL. For the integrated GPU, all of the data is allocated and initialized by the CPU in the main physical memory and directly referenced by the integrated GPU. The discrete GPU uses similar OpenCL kernels, but the data is explicitly copied from the CPU memory to the GPU, which is not necessary when using the integrated GPU and HSA.

5. RESULTS

In this section, we first discuss the performance and energy characteristics of scans on discrete and integrated GPUs. Next, we evaluate our scan and aggregate algorithms on a set of scan-aggregate TPC-H queries. Finally, we show the tradeoffs between the two GPU aggregate algorithms discussed and compare their performance to the CPU.

5.1 Scans (Discrete vs. Integrated)

We first investigate the performance of the scan operator on the two GPU architectures. The scan microbenchmark that we run performs a simple selection operation (e.g., `SELECT count(*) from TABLE`, omitting the actual count operation) on a single column. The microbenchmark applies a predicate which has a 10% selectivity. Thus, this benchmark measures the raw cost associated with scanning a column excluding the costs to process the result bitvector that is produced.

Figure 5 shows the performance and energy of the scan operation on a discrete GPU and an integrated GPU. This figure shows the number of seconds to complete a scan of

	CPU	Integrated GPU	Discrete GPU
Part name	A10-7850K	R7	HD 7970
Cores / CUs	4	8	32
Clock speed	3700 MHz	720 MHz	1125 MHz
Mem. bandwidth	21 GB/s	21 GB/s	264 GB/s
Total memory	16 GB (shared)		3 GB
Rated TDP	95 W (combined)		225 W

Table 1: Details of hardware architectures evaluated. (Note: TDP for the CPU and integrated GPU does not include the memory power. TDP for the discrete GPU includes the GDDR5 memory.)

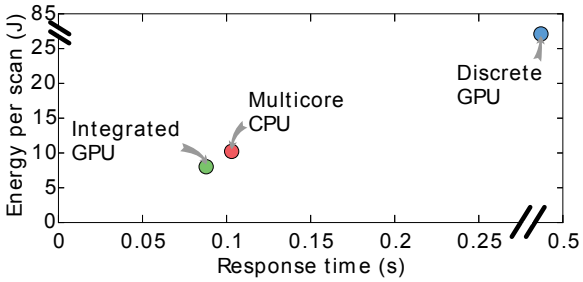


Figure 5: Performance and energy of scan microbenchmark. Discrete GPU on broken axes.

1 billion 10-bit codes averaged over 1000 scans. The total size of the column is about 1 GB. Performance is shown on the x-axis (response time), and energy is shown on the y-axis.

Figure 5 shows that performing a scan on the integrated GPU is both 3× faster and 3× lower energy than performing a scan on the discrete GPU. Although the discrete GPU has many more compute units and much higher memory bandwidth, the time to copy the data from the CPU memory to the GPU memory dominates the performance (about 98% of the execution time). This time can be decreased by overlapping the memory copy with computation; however, the performance on the discrete GPU is fundamentally limited by the low bandwidth and high latency of the PCIe bus.

The integrated GPU has a slightly faster response time than the multicore CPU (17% speedup), but it's energy consumption is significantly lower (27% less energy consumed). Since the integrated GPU and CPU share the same memory interface, the scan performance should be similar because scan is a bandwidth-bound computation. However, the power consumption of the integrated GPU is lower than the multicore CPU. The full-system power when using the integrated GPU is about 90 W compared to 110 W for the multicore CPU. Thus, in today's systems, if energy consumption is important, the integrated GPU provides a significant benefit over the multicore CPU for simple scans.

In the rest of this paper we do not evaluate the discrete GPU for two reasons. First, the integrated GPU outperforms the discrete GPU for the scan operation when the copy and initialization overheads are included. Second, it is difficult to implement the aggregate on the discrete GPU due to the complex data structures used in our algorithms (e.g., a hashtable). It may be feasible to design optimal scheduling strategies and new data structures for the discrete GPU. However, we find that the integrated GPU provides significant performance improvement and energy savings without paying the high programming overhead of the discrete GPU.

5.2 TPC-H

For the workload, we used sixteen TPC-H queries on a pre-joined dataset as was done in Li et al. and Sun et al. [18, 27]; i.e., we pre-joined the TPC-H dataset and ran queries on the materialized dataset (WideTable) using scans and aggregate operations. The queries not included have string-match operators, which have not been implemented in our system. The queries evaluated show a range of different behaviors: the percent of time in scan operations ranges from about 0% to more than 99%; the number of scans performed ranges from zero scans to 18 scans; and the columns vary

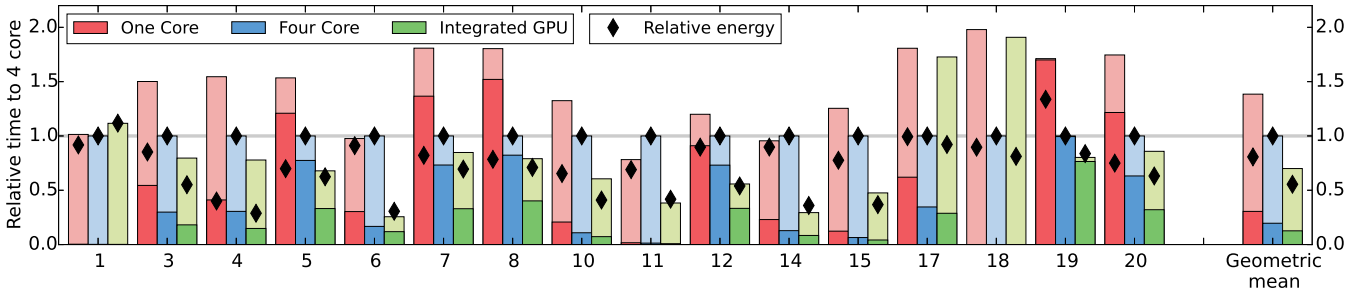


Figure 4: Performance and energy for 16 TPC-H queries normalized to the four-core CPU. Lower bold portion of each bar is the scan time and the upper shaded portion of each bar is the rest of the time spent in aggregate, group-by, and sort operations. Energy is shown on the right axis with diamonds relative to the four-core CPU.

in bit width from 3- to 32-bits. We run each query with a scale-factor 10 data set.

Execution time of the TPC-H queries relative to the multicore CPU is shown in Figure 4. The execution time of each query is broken into two sections: the time to perform the scan (the darker bottom part in each graph), and the time to execute the rest of the query. The non-scan parts of the queries include aggregate, group-by, and sort operations. Only the aggregate and group-by operations were ported to the GPU; all sort operations execute on the CPU.

The two queries which see the most performance improvement using the integrated GPU are query 6 and query 14. For these two queries, the GPU is able to significantly increase the performance of the aggregate computation. The reason for this performance improvement is that these queries perform a simple reduction (one group) and touch a large amount of data. These two queries have a relatively high selectivity (1–2%) and access large-width columns. For these two queries, the increased parallelism of the GPU can take advantage of the memory bandwidth in the system better than the multi-core CPU platform.

For some queries, the integrated GPU significantly outperforms the multicore CPU in the scan operation (e.g., query 5 and query 12). The reason for this higher performance is that these queries spend a large percentage of their execution time in single table multi-column (STMC) predicates. STMC predicates scan through two columns comparing each row (e.g., in query 12 `l_commitdate < l_receiptdate`). For STMC scans there is an even smaller compute to memory access ratio than for single-column predicate scans. Thus, the integrated GPU can utilize the available memory bandwidth more effectively than the multicore CPU.

Overall, the integrated GPU increases performance compared to the multicore CPU of both the scan and aggregate portion of the TPC-H queries evaluated as shown by the geometric mean in Figure 4. The integrated GPU shows a 35% average reduction in response time for scans, a 28% reduction in response time on aggregates, and an overall 30% average performance improvement.

The diamonds in Figure 4 show the the whole-system energy of the each TCH-H query evaluated relative to the multicore CPU (right axis, measured with a WattsUp meter). This data includes both the scan and the aggregate portions of each query. Surprisingly, the single-core CPU uses less energy than the multicore CPU, on average. The reason for this behavior is that the aggregate portion of the queries is not energy-efficient to parallelize. Performing parallel aggregates results in a performance increase, but not enough to offset the extra power required to use all four CPU cores.

Figure 4 also show that the integrated GPU is more energy-efficient when performing whole queries. The integrated GPU uses 45% less energy than the multicore CPU and 30% less energy than the single-core CPU.

We found that some aggregate queries perform poorly on the integrated GPU (Q1, Q5, Q7, and Q8). Thus, we use a mixed-mode algorithm for the aggregate computation (Section 5.3 presents a more detailed analysis). Additionally, we found that some aggregate queries (Q1, Q6, Q11, and Q14) perform poorly when parallelized. The aggregate operations in these queries do not parallelize efficiently due to overheads from locks to keep the shared data structures coherent. For the queries that aggregate operations perform poorly on the multicore CPU or the integrated GPU, we use a single CPU core to execute the aggregate operation.

5.3 Aggregates

In this section, we compare the aggregate algorithms discussed in Section 3.1. We use a microbenchmark which applies a selection predicate on a table with two (10-bit) integer columns and then performs an aggregate operation returning the sum of the values in the first column grouped by the second column's value. We vary the codes size of the two columns and the selectivity of the query.

Figure 6 shows the performance of the two group-by algorithms on the integrated GPU assuming a 0.1%, a 1.0%, and a 10% selectivity in the scan portion of the query. We found that selectivities above about 10% were dominated by the time to materialize the codes, not the group-by algorithm. Each graph sows the performance of the two algorithms discussed in Section 3.1, direct (Algorithm 2) and hashed (Algorithm 3). Since Algorithm 2 is only appropriate when there are a small number of groups and Algorithm 3 is only appropriate when there are a large number of groups, each algorithm was only evaluated on a subset of the evaluated groups (x-axis).

Figure 6 shows that for the simple reduction and a large numbers of groups, the GPU outperforms the CPU when using the hashed group-by algorithm. However, there is a range of groups for which neither the hashed nor the direct group-by algorithm performs well on the GPU. For this range, the CPU significantly outperforms the GPU (up to about 1024 groups). Therefore, we advocate using a mixed-

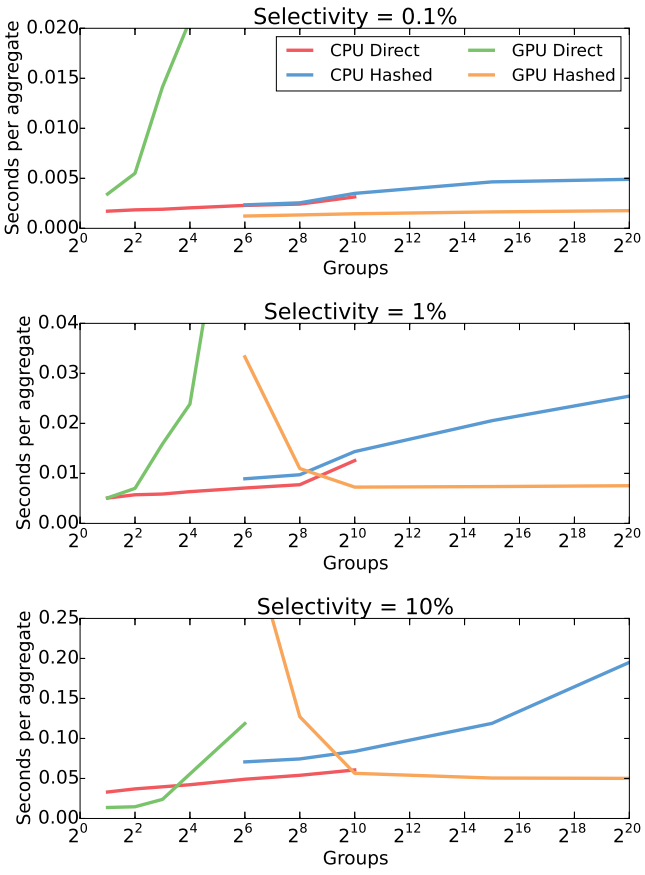


Figure 6: Performance of aggregates comparing number of groups with three selectivities.

mode algorithm. Luckily, the HSA programming model makes it possible to perform a simple run-time decision.

The reason the GPU performs poorly with a small number of groups is two-fold. First, when using the direct algorithm, the GPU must allocate memory to store the local results for every single GPU thread, which can be more than 10,000. This memory overhead means the direct algorithm can only be used when there are 64 groups or fewer. Additionally, this per-thread memory is not optimized on the GPU. Second, when using the hashed storage algorithm, for small numbers of groups (less than 1024), the hashed storage algorithm performs poorly on the GPU because there is contention for the data in the hash table. The contention causes the lock-free algorithm we use to perform poorly.

We also find that selectivity affects the speedup of the GPU over the CPU for the aggregate primitive for the simple reduction (group-by with one group) and hashed group-by algorithm. For a simple reduction (one group), as the selectivity increases and the total number of records that are materialized grows, the GPU is able to achieve more speedup over the CPU. When the selectivity is higher, there is more memory parallelism available and the GPU performs better. Additionally, we use the CPU to compute the offsets and the GPU only computes the reduction value. Thus, with very low selectivity rates, the CPU time dominates the aggregate and performing the entire aggregate on the CPU has the highest performance.

6. DISCUSSION OF IMPLICATIONS FOR FUTURE SYSTEMS

The advent of 3D die-stacking is a technical change on the horizon that may have profound impacts on analytic data processing [21]. 3D die-stacking allows multiple chips, possibly from disparate manufacturing processes, to be combined into a single package. These future systems will have two features that may significantly impact the performance of analytic database workloads, increased compute capability and greatly increased bandwidth.

A possible future 3D-stacked system could have a multicore CPU, a GPU, and DRAM all packaged together. This package would take a single socket in today's motherboards. Because it is on it's own die, the GPU in a 3D-stacked system can have all of the performance that discrete GPUs have promised with the same integrated GPU programming interface. Additionally, the main memory bandwidth in these systems can be up to 1 TB/s, more than 10× current platforms.

In these future very high-bandwidth systems, the integrated GPU shows more benefit over the multicore CPU than we see today. The GPU can more efficiently exploit memory-level parallelism than the CPU, providing a significant performance and energy improvement. Using a 3D-stacked GPU may provide a 16× performance improvement over today's multicore CPUs and 4× speedup over the a 3D-stacked CPU [21].

7. RELATED WORK

There is a rich body of work accelerating database applications on GPUs, but most of this prior work focuses on discrete GPU systems [10, 11, 15]. Our work builds off of these initial works. We argue that integrated GPUs will give these algorithms new life since they eliminate the high-overhead data movement costs of discrete GPUs.

However, some recent work has focused on integrated GPUs. He et. al show that the fine-grained co-processing enabled by integrated GPUs can provide significant performance improvements [12]. Additionally, He et. al show that integrated GPUs can benefit from the CPU's caches and show that these systems can increase the performance of analytical data processing systems [13]. Our work differs from these previous studies mainly because we use the emerging programming model, HSA. Previous work used the legacy OpenCL 1.X model, which still requires the programmer to explicitly declare how data will be used by the GPU. By using HSA, we are able to show the benefits of true virtual-address sharing and very low overhead GPU functions.

There has also been work optimizing aggregates on multicore processors [30], SIMD processing units [20], and intra-word parallelism [7]. Our work focuses on an initial aggregate algorithm for integrated GPU architectures. Applying the insights from these CPU-focused aggregate algorithms is left for future work.

8. CONCLUSIONS AND FUTURE WORK

Previous works have shown the huge potential of using GPUs for database operations. However, many of these works have neglected to include the large overheads associated with discrete GPUs when operating on large in-memory databases. We show that for scan-aggregate queries, current physically and logically integrated GPUs mitigate the

problems associated with discrete GPUs showing a modest speedup and energy reduction over multicore CPUs.

Looking forward, computer architects are pursuing many interesting avenues to increase the memory bandwidth significantly, such as 3D die-stacking. However, conventional multicore CPU architecture is not well suited to efficiently use this increased memory bandwidth. We believe that GPUs pave the way for databases to take advantage of these emerging architectures.

9. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (CCF-1218323, CNS-1302260, CCF-1438992, IIS-0963993, IIS-1110948, and IIS-1250886), the Anthony Klug NCR Fellowship, Cisco Systems Distinguished Graduate Fellowship, Google, and the University of Wisconsin (Kellett award and Named professorship to Hill). Hill and Wood have a significant financial interest in AMD and Google. Patel has a significant financial interest in Microsoft and Quickstep Technologies.

10. REFERENCES

[1] L. Abraham et al. Scuba: Diving into data at facebook. *PVLDB*, 6(11):1057–1067, 2013.

[2] AMD. AMD's most advanced APU ever. `http://www.amd.com/us/products/desktop/processors/a-series/Pages/nextgenapu.aspx`. Accessed: 2014-1-23.

[3] AMD. Graphics Card Solutions. `http://products.amd.com/en-us/GraphicCardResult.aspx`. Accessed: 2014-1-23.

[4] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD Conference*, pages 271–282, 2001.

[5] W. chun Feng and S. Xiao. To gpu synchronize or not gpu synchronize? In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3801–3804, May 2010.

[6] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[7] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *Data Engineering (ICDE), 2015 IEEE 31th International Conference on*, 2015.

[8] G. GLIGOR and S. Teodoru. Oracle Exalytics: Engineered for Speed-of-Thought Analytics. *Database Systems Journal*, 2(4):3–8, December 2011.

[9] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD Conference*, page 325, 2006.

[10] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD Conference*, page 215, 2004.

[11] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD Conference*, page 511, 2008.

[12] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB*, 6(10):889–900, 2013.

[13] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4):329–340, Dec. 2014.

[14] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[15] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *DaMoN Workshop*, pages 55–62, 2012.

[16] S. W. Keckler. Life after Dennard and How I Learned to Love the Picojoule. In *MICRO 44 Keynote*, 2011.

[17] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD Conference*, pages 289–300, 2013.

[18] Y. Li and J. M. Patel. WideTable: An Accelerator for Analytical Data Processing. *PVLDB*, 7(10), 2014.

[19] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[20] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern simd processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[21] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood. Implications of emerging 3D GPU architecture on the scan primitive. *SIGMOD Rec.*, 44(1), 2015.

[22] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[23] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE Conference*, 2008.

[24] P. Rogers. Heterogeneous System Architecture Overview. In *Hot Chips 25*, 2013.

[25] K. Rupp. CPU, GPU and MIC hardware characteristics over time. `http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`. Accessed: 2015-05-05.

[26] 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 *SIGMOD Conference*, pages 351–362, 2010.

[27] L. Sun, S. Krishnan, R. S. Xin, and M. J. Franklin. A partitioning framework for aggressive data skipping. *PVLDB*, 7(13):1617–1620, 2014.

[28] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing database column scans with complex predicates. In *AMDS Workshop*, pages 1–12, 2013.

[29] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[30] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 1–9, New York, NY, USA, 2011. ACM.

[31] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD Conference*, pages 145–156, 2002.