Towards a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration

Sina Meraji [#1], Berni Schiefer [*2], Lan Pham [*3], Lee Chu [*4], Peter Kokosielis [*5], Adam Storm [*6]

[#] *Cloud Innovation Lab, IBM Canada Ltd.*
[1] sinamera@ca.ibm.com

[*] *IBM DB2, IBM Canada Ltd.*
[2] schiefer@ca.ibm.com
[3] lpham@ca.ibm.com
[4] leechu@ca.ibm.com
[5] pkolosie@ca.ibm.com
[6] ajstorm@ca.ibm.com

ABSTRACT

In this paper, we show how we use Nvidia GPUs and host CPU cores for faster query processing in a DB2 database using BLU Acceleration (DB2's column store technology). Moreover, we show the benef ts and problems of using hardware accelerators (more specif cally GPUs) in a real commercial Relational Database Management System (RDBMS). We investigate the effect of off-loading specif c database operations to a GPU, and show how doing so results in a signif cant performance improvement. We then demonstrate that for some queries, using just CPU to perform the entire operation is more benef cial. While we use some of Nvidia's fast kernels for operations like sort, we have also developed our own high performance kernels for operations such as group by and aggregation. Finally, we show how we use a dynamic design that can make use of optimizer metadata to intelligently choose a GPU kernel to run. For the f rst time in the literature, we use benchmarks representative of customer environments to gauge the performance of our prototype, the results of which show that we can get a speed increase upwards of 2x, using a realistic set of queries.

CCS Concepts

•Information systems → Query optimization;

Keywords

Query Optimization, GPU, DB2 BLU, Group by, Aggregation, Heterogeneous system

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2903735

1. INTRODUCTION

In-memory columnar relational database management systems (RDBMSes) have been shown to be very performant for analytic query processing, as they can use the massive parallel processing of multi-core processors to quickly respond to user queries. As a result, in-memory columnar RDBMSes exist from multiple vendors, including HP (Vertica), SAP (HANA), Microsoft (SQL Server) and IBM (DB2). It has also been shown that hardware accelerators like GPUs, FPGAs and custom design ASICSs can be used to improve the performance of various applications [4, 5, 6, 7, 9, 11, 12, 13, 16, 24] . With that understanding, we have demonstrated two main achievements from our work. The f rst is to show that we can improve DB2 BLU query performance by using Nvidia GPUs. Improving performance through the use of a hardware accelerator requires some major changes in the software. As a result, our second achievement is the illustration of some of the challenges that arise when integrating such accelerators into a commercial database.

GPUs, such as those from Nvidia, are programmable hardware accelerators that provide massive parallel computing power. While GPUs have traditionally been used mostly for graphical applications, today's database applications (such as DB2 BLU) can exploit GPUs to off oad CPU computing intensive operations to run on the GPUs. The main goals of using GPUs for query processing in DB2 BLU are:

- Utilizing the massive parallel computing capability of GPU for DB2 engine operations that can be executed in parallel, thus reducing the runtime of such queries.
- Freeing up CPU cycles by off-loading CPU intensive operations to run in the GPU. The savings in CPU cycles can be applied to other running tasks in the system. This benef ts multiple concurrent user scenarios where the CPU savings from one query (while running in the GPU) can be applied to other running queries.

The amount of speed up we can achieve by using GPUs depends on the complexity of the operation, the size of the data, as well as the percentage of time the query spends within the costly operations such as sort, group by, aggregation, and join over the entire query.

Using GPUs to improve the performance of specific database operations such as sort [6, 9, 11, 22], join[12, 13, 14], indexing[16, 17], compression[8, 19], and selection[10, 25] have been studied in various literature. Also techniques in [20] can be utilized for GPU aggregation operations. Moreover, [23] shows how GPUs can provide a cost-effective computing platform to support processing of large amounts of data streamed into a push-based database system. Based on this, the authors also provide some recommendations on how to design a query processing engine for such systems that run on GPUs. In [26] the authors show very preliminary work on a kernel- adapter based database engine named OmniDB which uses both the CPU and the GPU for query processing. [15] designs a database system that can efficiently exploit highly heterogeneous hardware environments. The main contributions of our paper, in relation to previous work, are:

1. **Performance benefits to using GPUs in conjunction with CPUs-** We demonstrate a design that uses both Nvidia GPUs and CPU cores for fast processing of database operations such as group by, aggregation and sort. In our design we show a hybrid approach that performs portions of the query processing in the CPU, and then uses the GPU to process certain compute intensive parts of the query.

2. **Efficient integration of the Nvidia GPU to DB2 BLU run time**- One of the main bottlenecks in hardware accelerating our queries is in coming up with a data format that the accelerators can process efficiently. This requires copying data to the hardware accelerator's memory, which would be the primary penalty to be paid. In our design however, we ensure we have the same number of data copies in our prototype as in the original DB2 BLU product. We also design our GPU kernels such that they can process DB2 BLU data with minimum conversion cost.

3. **Using heuristics to intelligently process queries**- That is, at runtime we can decide to use either the CPU, GPU, or both. We also use the meta-data which is generated during initial processing of query at DB2 BLU runtime, to tune/optimize the size of data structures on the GPU on-chip memory. To the best of our knowledge, none of the existing approaches use runtime meta-data to tune the size of data structures on the GPU.

4. **Multiple group by/aggregation GPU kernels**- In our design we have different group by/aggregation kernels while functionally these kernels perform the same operation, each kernel suits a specific type of query. None of the existing techniques allows an accelerator kernel to be dynamically selected at runtime. We have implemented a software moderator that can intelligently make this decision at runtime based on factors such as the number of groups, the number of payloads, and the ratio of rows to number of groups. Moreover, if we have enough compute resources and memory on the GPU, we can use more than one kernel to execute the same query at the same time.

5. **Scheduling tasks across multiple GPUs-** We have built a simple scheduler which lets the DB2 BLU run time schedule tasks on the different GPUs. The scheduler takes into account parameters such as the resources required by the task and the resources currently available by each of the GPUs. This allows for flexibility in terms of hardware configurations, as the GPUs do not need to be homogenous in their specifications.

The rest of this paper is organized as follows. In section 2 we discuss how we have changed the DB2 BLU engine to add support for Nvidia GPUs. Section 3 explains how we use a hybrid design to use both CPU and GPU for sorting. Fast group by/aggregation GPU kernels are studied in section 4. Section 5 shows our performance result on some standard benchmarks. Finally, section 6 concludes with our thoughts on the future of GPU acceleration in databases.

2. DB2 BLU WITH GPU INFRASTRUCTURE

In order to integrate the Nvidia GPUs to DB2 BLU, we need to make infrastructure changes in the DB2 BLU runtime code. The main components that are added to the DB2 BLU engine are: 1) memory management unit 2) support for multiple GPU devices 3) performance monitoring. We will briefly describe all three components in the following section.

2.1 Memory Management Unit

In this section we explain the changes we have made in DB2 BLU memory management unit to add GPU support.

2.1.1 Device Memory Reservation

Device memory reservation is important when there are multiple concurrent tasks (threads or processes) that request device memory from the same GPU device at the same time. If there are multiple concurrent tasks all entering GPU kernel execution at the same time, the contention for device memory between these tasks could lead to a memory allocation error during task execution. In such a scenario, the calling function would have to enter the expensive error code path to handle the out of memory error and roll back the operation to a consistent point, and then retry the operation. Moreover, implementing a memory reservation system would allow a GPU operation to complete if it could successfully satisfy all of its memory needs up front. We have implemented a simple memory reservation approach in which we track device memory usage by all consumers on a given GPU device, such that a task can query and reserve memory up front before proceeding to run the devices kernel code. Once the GPU kernel code execution is finished, the reserved memory is released for use by other tasks. If a task fails to reserve the memory, it can then either 1) wait until device memory becomes available or 2) fall back and run the task on the CPU (host).

2.1.2 Host Memory Registration

Data transfers (memory copying) between the host and the GPU device can be very expensive if the memory is not registered (pinned) with the GPU device up front. If the host memory is registered with the GPU device, then the speed of data transfers from registered host memory can be more than 4X faster than the speed of data transfer from unregistered memory using PCI-e gen 3. For this reason, all memory that is allocated for GPU operations is pre-registered with the GPU device(s) during the DB2 BLU start up phase. Registering the individual memory block can be very expensive. As a result, it is preferable to avoid the registration on each invocation of the kernel call. With our approach, the registration of all required memory is thus performed up front, using a single large memory segment. This allows subsequent memory requests on each kernel invocation to be accommodated through an allocation from this single large (registered) memory segment. When the GPU kernel finishes its work and returns, the allocated memory is returned to the free pool of registered memory.

2.2 Support for Multiple GPU Devices

Our design supports multiple GPU devices by distributing workload to all of the available devices. To allow for good distribution, we must first track the number of outstanding jobs that are currently executing on the GPU devices, as well as track the amount of available memory on each device. In addition, we know the amount of memory that each kernel invocation call needs in advance. This can be calculated using the type of the query, size of the input data, and size of the internal data structures needed in the GPUs memory to perform the operation. After calculating the total memory size that a kernel invocation needs, we consult the GPUs to see if any of them has enough free resources to execute the given kernel call.

Supporting multiple GPU devices is highly important for 1) large workloads where the data set exceeds any single device's memory limit, and 2) scenarios in which multiple concurrent users are running on the system. In the case of large workloads, the input data is partitioned (typically using range partitioning) into multiple smaller chunks, and these smaller chunks are sent to some number of available GPU devices, to be operated on concurrently. The results are then merged together in the final step and returned back to the user. In the case of multiple concurrent users whose queries are able to exploit the GPUs, the set of queries can be scheduled to run on all GPUs and therefore fully utilize all available resources on the host.

2.3 Performance Monitoring

Monitoring the time spent inside the GPU and the time spent transferring data to and from the GPU is very important. This profiling information is mainly used to tune the kernel code in an effort to improve performance. In addition, this monitoring information can be used to optimize the implementation and therefore reduce the amount of data transferred between the host and the GPU device.

While there are existing tools (such as Nvidia's nvidia-smi tool) to monitor overall GPU performance and memory usage, they cannot profile detailed information about a GPU that is integrated to an existing application, which is the case in our implementation. As a result, we were forced to implement our own performance monitoring tooling. Our performance monitoring tool for the GPU is integrated with the existing DB2 BLU performance monitoring infrastructure and provides detailed information about the GPU related calls and kernels. This monitoring tooling was then used to tune the performance of the GPU kernels so that they could be compared with their CPU implementations.

3. OFFLOADING SORT DATA TO GPU

The DB2 BLU sort code receives incoming tuples for each column that is to be sorted. These tuples are stored in memory in a structure known as the Sort Data Store (SDS), which we refer to as the SDS buckets. Data in the SDS buckets remains unmodified and never moves during the sort. Since these tuples could be quite large, we do not want our swap operations during the sorting phase to have to move them around in memory. Instead, we have an intermediate structure that we call the partial key buffer.

Each partial key buffer entry consists of a 4 byte key and a 4 byte payload. The key is a partial binary sortable representation of the column that is being sorted, and subsequent fetches of the next partial key may be required to determine the final ordering. The payload is a pointer into the SDS bucket for the corresponding tuple. The payload is able to grow larger than 4 bytes, in cases where 4 bytes is not enough to address the number of rows in the SDS buckets.

DB2 BLU sort uses a job queuing system to represent each sorting task. Initially there is only one job, which represents the entire data set. When a job is started, it is taken off the queue and the host will generate (in parallel) a set of partial keys and payloads and populate the partial key buffer with this information. The DB2 BLU engine then determines if the job is a candidate for a GPU sort, based on the number of tuples in the job. For instance, if the number of input tuples is very small, there is no benefit in forwarding the sort job over to the GPU because the combined cost of the transfer time plus processing time overshadows the performance savings we would have gained from using the GPU to sort the job.

Though multiple threads are used to generate partial keys and payloads on the host side, only a single thread is needed to dispatch the job to the GPU. As explained in section 2-2, a scheduling algorithm is used to determine the GPU to which the job will be dispatched. Once a CPU thread has been chosen to dispatch the job to the GPU, the remaining CPU threads for this job are released since their work is complete. These released threads will process the next available job on the queue. We use the GPU Duane sort kernel[18] provided by Nvidia to perform the sort operation in the GPU. When the GPU returns from processing, it will have sorted the partial key buffer based upon the 4-byte partial key. This of course does not represent a completely sorted data set, since all we know up to this point is a partial ordering based upon the first 4 bytes. If there exists more than one tuple for which the first 4 bytes are identical, the next set of partial keys needs to be generated. We call such a group of tuples a duplicate range, which the GPU identifies for us. A new job is created for each duplicate range and is placed on the job queue. The sort is complete when there are no more jobs on the queue.

The job queuing system that we have in place is flexible to allow for concurrent jobs to be executed both by the GPU and the CPU. This allows for a truly hybrid sorting system and lets the host perform sorting where it does not make sense for the GPU to do so, such as for a small number of rows. This also facilitates load balancing where more threads can be directed at certain jobs.

Another feature of our sort design is that it is not dependent on the data type of the column being sorted, as we have transformed the underlying type into a binary stream that is sorted on 4 bytes at a time. Moreover, we have a merge free sort algorithm that uses both the CPU and GPU for sorting. Merging is a very costly step in all the sort algorithms that have a merge step. In our design, we remove the merging step by making conflict free partitions before sending sort jobs to the GPU.

4. GROUP BY / AGGREGATION

We have designed a hybrid hash-based group by/aggregation algorithm that utilizes features of DB2 BLU and Nvidia GPUs for fast query processing. Group by/aggregation can be very costly in all RDBMSes when there are a large number of grouping columns/ keys and/or aggregation functions. However, for queries with a small number of input rows, using the GPU would be slower when compared to using only the CPU. One of the advantages of our design is that we can dynamically decide where to execute a group by/aggregation query based on its runtime features. In order to have these features, we made changes in the execution chain of group by/aggregation queries.

In all hardware accelerators, there are a limited (but usually large) number of compute resources. Knowing the query requirements in advance has a lot of benefits. We can 1) check to see if we have enough free resources on the GPU to execute the same query with different types of kernels 2) choose a suitable GPU kernel dynamically at run time. In addition, if we have enough resources on the

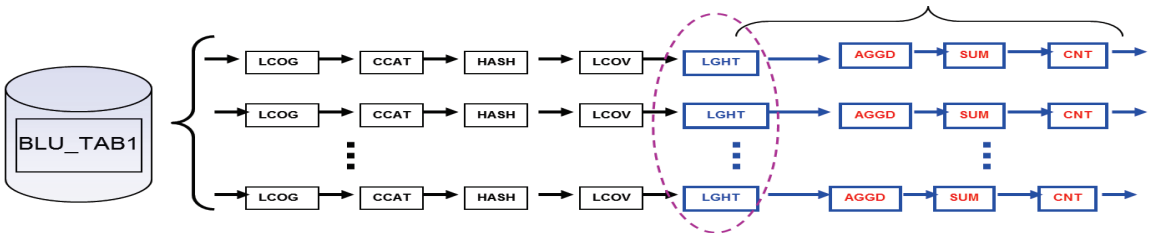

Figure 1: DB2-BLU Group by/Aggregation chain

GPU, we can run the query concurrently on two or more different kernels. This helps us f nd out the fastest kernel for a specif c query. We can then stop the other kernel(s) as soon as one of the kernels f nishes its job. This would give us more opportunity for faster processing of the query 3) add feedback logic to the design that informs a software moderator about the computation of the query using a specif c kernel. The moderator can then learn over time which of the kernels to use, given a specif c type of query. This feature is not yet implemented. Hence, we make use of metadata that is generated by the DB2 BLU runtime to accelerate query processing on the GPU. Such metadata includes the exact number of tuples, the estimated number of groups, and the number of aggregation functions. This information can help us call different kernels at runtime based on the type of the query.

In addition, this metadata allows us to optimize the size of the data structures we use in the GPU. This is very important because, as described, the GPU memory is very limited when compared to the host machine. The main data structure that we use to process the group by/aggregation queries in the hardware accelerator is a global hash table in the GPU memory. In our design we use the estimated number of groups to set the size of this hash table. If we do not know the number of groups then we need to set the size of hash table to be as big as the number of input rows which is much larger than number of groups in most queries. We describe our design in four main parts: 1) Changes we have made to the DB2 BLU runtime to integrate the GPU 2) The GPU run time 3) The group by kernels 4) The aggregation functions.

4.1 Eff cient Integration of GPU to DB2 BLU Runtime

Figure 1 shows the group by/aggregation chain in DB2 BLU. In the current design, group by/aggregation is performed in two phases. In the f rst phase, parallel threads read data from DB2 BLU tables and perform different operations on the data as depicted in f gure 1. When the f rst step f nishes, the f nal result is merged into a global hash table. The description of the different evaluators is as follows[21]:

LCOG, LCOV: Load grouping keys and payloads

CCAT: Concatenate keys for queries that group by on more than one column

HASH: Hash function that receives grouping keys and returns a hashed value

LGHT: First phase of BLU group by that creates groups in local hash tables

AGGD, SUM, CNT: Apply different aggregation functions

In our new design, as depicted in f gure 2, we moved some of these evaluators to the GPU and also added some new evaluators. As you can see, the LGHT and all aggregation functions are completely removed from the DB2 BLU group by evaluator chain. This is because we perform all of these operations in the GPU. We use CPU threads to load data and then use a simple hash function and KMV[2] algorithm to estimate the number of groups. We will use this estimate to tune the size of the GPU data structures.

Instead of having an LGHT evaluator, we have a simple MEMCPY evaluator that copies data to the pinned memory on the host machine. As explained, transfers to/from the GPU on-chip memory should be through the pinned memory because of its faster speed. After data is copied to the pinned memory, one of the CPU threads will call the GPU to perform the rest of operations. When the GPU f nishes its job, the result will be ready in the pinned memory. Details are in the next section. Moreover, we use input from the DB2 optimizer to choose a suitable group by/aggregation chain. The optimizer can provide some information like the number of groups/input rows before we start processing the group by chain. If the number of rows or groups (from the optimizer estimates) is smaller than a predef ned threshold (T1), we perform the entire computation in the host (i.e. in CPU, not GPU) using the DB2 BLU chains depicted in f gure 1. This optimization can be very benef cial for very small queries with small numbers of rows or groups. For these small queries, the cost of sending data to the GPU can be much more than the speed up we achieve by using the GPU. As a result, it is more suitable to perform the entire computation on the CPU for such small queries.

In contrast, if the number of rows is larger than a specif c threshold T1 while the number of groups is also larger than a threshold T2, we want to perform the entire computation on the GPU. This set of queries is very common and the GPU can be eff ciently used to improve the performance of these types of queries. We would like to emphasize that for queries with a very small number of rows or groups, the CPU is already very fast so there is no need to use the accelerator. As previously mentioned, we still use part of the original DB2 BLU group by chain to pass data to the accelerator, but the main computation is performed on the hardware accelerator. If the number of input rows is very large(larger than T3), the data will not f t in accelerator memory. In this case we will need to partition the data and use both the CPU and the GPU for query processing. In our current implementation, all of the large queries are processed in the CPU. This is depicted in f gure 3.

4.2 GPU Runtime

If it is decided that a query is to be executed on the GPU, only one of the CPU threads will launch the GPU runtime. As depicted in f gure 2, by this time all data have gone through the HASH evaluator. The HASH evaluator and KMV[2] algorithm together can help us to come up with a good estimate for the number of groups. Moreover, the exact number of input rows and aggregate functions is already determined by this point. All of the metadata is sent to the GPU runtime.

In the GPU, we use a hash based algorithm to perform the group by/aggregation operation. The details of the various group by and aggregation kernels are described in sections 4-3 and 4-4. In all of

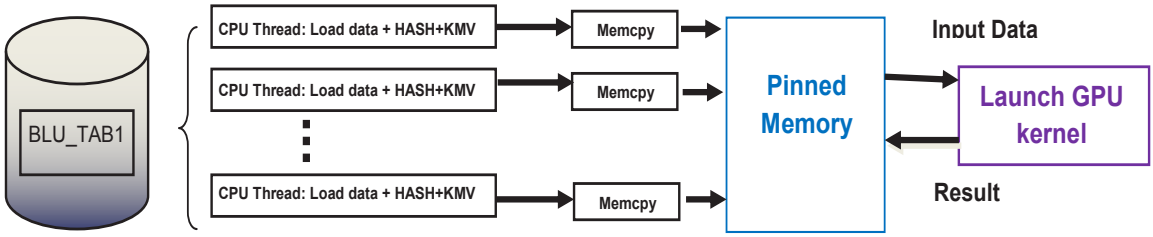

Figure 2: Group by/Aggregation chain in DB2-BLU with hardware accelerator

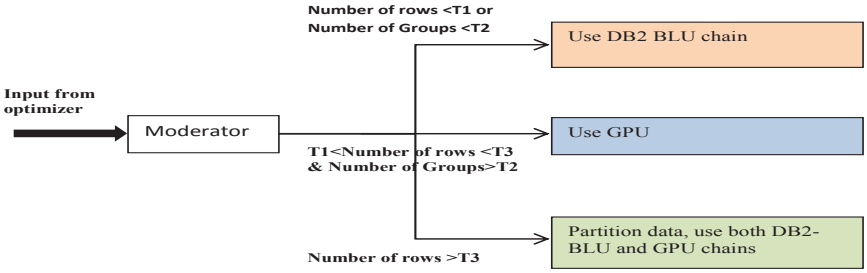

Figure 3: Using optimizer data to select the path for group by/aggregation queries

these hash based algorithms the size of the hash table is an important parameter that affects the performance of the system. Meanwhile, the estimated number of groups can be used to tune the size of the hash table. We also have an error detection code-path, so if the estimated number of groups is not correct (smaller than the exact number of groups) we could still process the query.

A moderator called the GPU moderator allows us to select an appropriate hardware accelerator kernel. The best kernel would be the one that can finish the computation in the fastest time using the fewest resources. However, there could be cases that the available resources on the GPU(compute resources, memory, cache, etc) is greater than what the query needs. For such scenarios, we can run the query using multiple kernels concurrently and then stop the computation in all kernels as soon as one of the kernels finishes its job. Furthermore, if we have multiple kernels running and the GPU becomes resource constrained (perhaps because new queries are sent to the GPU), we can terminate the kernels which have made the least amount of progress.

4.3 Group By Kernels

As explained, we use a hash based algorithm to perform the group by/aggregation operations in the GPU. We can have different implementations of hash-based algorithms based on: 1) number of input rows 2) estimated number of groups 3) number of aggregation functions. In the following we explain three different hash based group by/aggregation algorithms and show how we use them based on meta-data that the DB2 optimizer provides.

4.3.1 First Kernel: Regular Queries

These are queries that have neither a very small number of groups nor a large number of groups. They also do not have a very large number of aggregation functions. For these kinds of queries we use hash based group by/aggregation algorithm. We use the meta-data provided by the DB2 optimizer to set the size of global hash table in the GPU device memory to be slightly larger than the estimated number of groups. Each entry in the input table has one grouping key and can have N different payloads and corresponding N aggregation functions. The first step is to initialize the hash table using the size of the grouping keys and payloads, and also the type of aggregation functions. The data in the hash table should be 1, 2, 4, 8 or 16 byte aligned (Nvidia GPU requirement), so we may need to do some padding at the end of each row in the hash table. To initialize the hash table as quickly and as efficiently as possible, we create a hash table mask based on the size of the payloads and aggregation functions and then use parallel CUDA threads to copy that mask to the hash table. The initial value for the grouping portion of the mask is a sequence of Fs. If the size of the grouping key is M bits, then we will have M/4 Fs as the initial value of the grouping key in the mask. Then the initial value of payloads in the mask is determined based on payload type and the corresponding aggregation function. For instance, if we have a payload of type 32bit integer and SUM as the aggregation function for that payload, the initial value in the mask would be 0(32bits). As an example for a query like: Select SUM(C1), MAX(C2), MIN(C3) from table1 group by(C1) where C1 and C2 are 64 bit integers and C3 is 32 bit integer, the mask would be: FFFFFFFFFFFFFFFF,0,-9223372036854775808,2147483647,0 Where -9223372036854775808 is the smallest 64 bit integer number (initial value to perform MAX(C2)) and 2147483647 is the largest 32 bit integer(initial value to perform MIN(C3)). The 0s at the end are added for padding. After finding the mask, parallel threads will copy that to the hash table in parallel as it is shown in table 1.

After initializing the hash table, we launch CUDA threads to read data from the input table in parallel and insert groups to the hash table. Each CUDA thread reads a row and then uses a hash function and the grouping key to find the insertion location in the hash table. If the corresponding entry in the hash table is empty we use atomic operations to insert the key. In case the entry is occupied, we try to insert the key to the next empty slot in the hash table. If the keys are less than 64 bit, we use CUDA atomicCAS to insert the key. This guarantees that 2 different CUDA threads do not overwrite the same hash entry. If the key size is larger than 64 bit, we can not use CUDA atomic operations any more so we try to acquire a lock to get access to the entry and then insert the key. The hash function we use when the key size is larger than 64 bit is the Murmur hashing algorithm [3]. For keys smaller than 64 bit we use a mod hash function. We perform the aggregation right after

Table 1: Parallel insertion to the hash table

C1(64bit)	SUM(C1)(64bit)	MAX(C2)(64bit)	MIN(C3)(32bit)	Padding(32bit)
FFFFFFFFFFFFFFFF	0	-9223372036854775808	2147483647	0
FFFFFFFFFFFFFFFF	0	-9223372036854775808	2147483647	0
...	...	...	...	...
FFFFFFFFFFFFFFFF	0	-9223372036854775808	2147483647	0

the insertion/f nding of grouping key in the hash table. If a thread discovers a key is already in the hash table, it updates the payload entries by applying the corresponding aggregation functions. We use either atomic operation or locks for that, as we will explain in section 4-4.

4.3.2 Second Kernel: Small Number of Groups

Queries with a very small number of groups are very common in real customer benchmarks. For example, grouping the employees of a company by their birth month would yield 12 unique values. However, if the number of input rows is large, processing such queries could be very costly. Moreover, Nvidia GPUs have Streaming Multiprocessors(SMX) where each SMX has its own 64KB conf gurable shared memory/L1 cache. We use this shared memory to perform partial grouping for queries with small numbers of groups. In order to have more space to keep groups/aggregated values in shared memory, we conf gure the shared memory/L1 cache to have 48KB as shared memory and 16KB as L1 cache.

Our algorithm has 2 main steps where in the f rst step, we perform a hash-based group by in the shared memory of the GPU using small hash tables that f t in the shared memory of GPU. This is similar to what we explained in section 4-3-1. Following that, the small hash tables in shared memory are merged to a global hash table in device memory. The main idea is to perform as many operations as we can in shared memory and then if it gets full, we can merge the partial result into a global hash table in the device memory. The reason we need to merge the result in the device memory is because the hash tables in different SMXs shared memories are totally separate. Basically, different threads perform group by in parallel on different chunks of input table tuples and generate partial group by results in the shared memory of different SMXs. Afterward the partial results are aggregated into the global hash table in device memory. The GPU runtime moderator will execute this kernel for all queries with small numbers of groups.

4.3.3 Third Kernel: Large Number of Aggregation Functions

The main structure of this kernel is the same as the one we explain in 4-3-1. This means that we use a hash table in the device memory of the GPU and then threads try to insert grouping keys into the hash table in parallel. However, the main difference is in the way that we perform the aggregation. Our experiments show that if the number of aggregation functions is large (more than 5), using atomic operations to perform these functions one by one can be time consuming. Moreover, as mentioned for some data types like 128bit Integer or Decimal data type, there is no CUDA atomic call, so we have to use locks. However, getting and releasing locks per aggregation operation can be costly when the number of aggregation functions is large. The other scenario in which the f rst kernel performs poorly is cases in which the ratio of number of rows/number of groups is small. As a result, the contention is very low and the cost of acquiring/releasing locks per aggregation function or atomic operation is high. For these kinds of queries we use a global lock approach where each thread locks the entire row of the hash table after it f nds out that its key matches the key in the corresponding hash table entry. After getting the lock, the thread goes through all the aggregation functions and applies them one by one. Finally, the thread releases the lock so other threads that want to update the same hash table row can acquire the lock. If both 1) the number of aggregation functions is large 2) number of rows/number groups is small, this kernel would have very good performance.

4.4 Aggregations

As explained, aggregation happens immediately after we f nd a group in which a grouping key belongs to. This means that the same thread goes through all the aggregation functions and corresponding values in the hash table (and the input table row) and applies all the functions one by one. We use 2 main approaches to perform the aggregation: 1) Atomic CUDA calls: For large sets of data types and aggregation functions, we can use CUDA atomic function calls. This includes some common data types like 32bit, 64 bit integer and f oat and aggregation functions like Min, Max, Sum and Count. We can also use atomicCAS to perform atomic operations on 128bit double or 128bit integer data types as explained in Nvidia documents [1]. 2) Locks: For some data types like f xed/variable size string that can have sizes larger than 128bit, we have to use locks. This is because we do not have the support from Nvdia hardware/software to perform the atomic operation. As a result, the threads need to acquire a lock f rst, apply the aggregation function, and then f nally release the lock so that other threads can use it. While with this approach we can functionally perform group by/aggregation queries that have large sizes on GPU, the performance is lower than the other data types because acquiring/releasing locks is very costly operation. As described in section 4-3-3, we can optimize this for some queries by locking per row instead of per payload.

5. PERFORMANCE RESULTS

In this section we describe some performance results we have achieved with our prototype. We have run unmodif ed customer representative benchmarks using a commercial RDBMS (DB2 BLU) off oading processing to a GPU. We believe this is the f rst time that such a test has been attempted. We will describe these benchmarks, show the performance numbers with and without GPU acceleration, and also study the performance of running a subset of the queries from these benchmarks on the GPU. Our tests are run on an IBM Power S824 system with 2 sockets (24 cores running SMT 4 for a total of 96 hardware threads, running at 3.92GHz), 512GB RAM, Ubuntu 14.04 LTS for the ppc64le architecture. We also have 2 NVIDIA K40 cards on the box. Each K40 has around 3000 cores and 12G of memory. Transfers to/from GPU are through PCI-e gen3.

5.1 Benchmarks

5.1.1 BD Insights Workload Description

Big Data Insights or BD Insights is a workload developed by

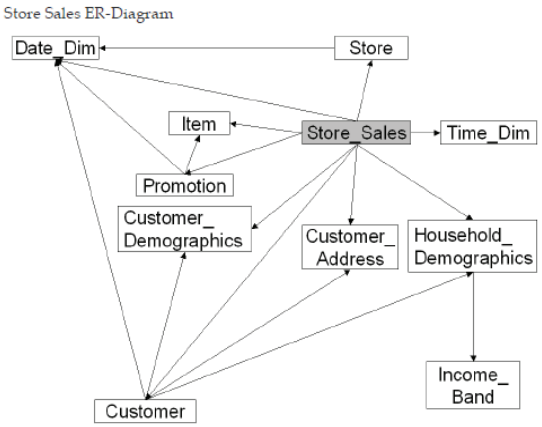

Figure 4: Store Sales fact table

IBM to model a day in the life of a customer representative business intelligence application.

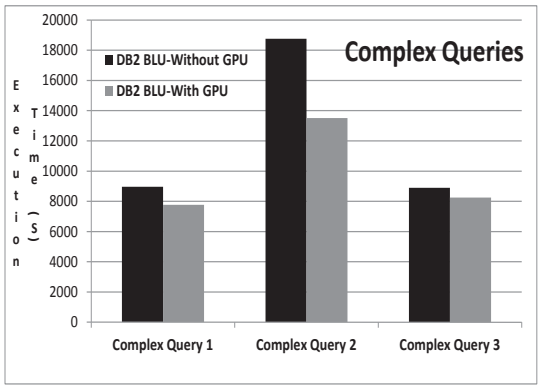

Figure 5: Complex queries in BD Insight benchmark

Database description: The data generator and database schema itself are derived from the industry TPC-DS Benchmark Standard (tpc.org). It represents a retail business and stores information about its merchandise sales through (1) its brick and mortar stores, (2) its online stores, and (3) through its catalog. The database also tracks customer demographic information, merchandise returns, inventory, and sales promotions among many other data points. There are seven fact tables in total and seventeen dimension tables in the schema. Figure 4 shows the star schema surrounding one of the fact tables which is called Store Sales in the database (representing merchandise sales from the brick & mortar stores).

BD Insights user description: The BD Insights workload has a separation of query types for three classes of typical business analytics users:

1. Returns Dashboard Analysts (Simple Queries): These users investigate the rate of merchandise returns and impact on the bottom line of the business. The queries can be characterized as short running against a narrow range of data, and likely to access a fact table.

2. Sales Report Analysts (Intermediate Queries): These users are generating sales reports to understand the profitability of their retail enterprise. The queries used can be characterized as being intermediate in complexity with a broader range of data accessed.

3. Data Scientists (Complex Queries): These users are hand crafting deep-dive analytic queries that answer questions identified by sales report analysts and the returns dashboard analysts. The queries can be characterized as long running and complex using complicated SQL constructs over a large or full data range.

Overall, the BD Insights workload contains 100 distinct queries: 5 are complex, 25 are intermediate, and 70 are simple. The workload can be run in several modes with both single user and varying multi-user combinations using the Apache JMETER load driver as a front-end client. For the purposes of this test, the interest was in whether the GPU acceleration modifications in DB2 would be able to functionally handle all of the queries, and then determine the query performance. The database was sized at 100GB so that it could fully fit in main memory, and would not overwhelm the memory available on the GPU cards.

5.1.2 Cognos ROLAP

Cognos ROLAP is an IBM internal analytical benchmark which simulates the core Business Intelligence(BI) workload from IBM Cognos suite. The Cognos ROLAP workload's schema and data generator are derived from the TPC-DS Benchmark Standard. In this particular instance, the queries were run against a BD Insights generated database. The Cognos ROLAP query set is composed of 46 complex analytical queries (a mix of join, group by, and sort), some of which include OLAP functions like RANK() that drive SORT. While the DB2 BLU engine is able to run all 46 queries, the prototype was only able to run 34 queries of these queries as the memory in the K40 GPU is limited, and 12 of the queries had memory requirements which exceeded the memory available.

5.2 Queries execution on the DB2 BLU with GPU

In this section we measure the performance of running BD Insight and Cognos ROLAP queries on DB2 BLU with GPU.

5.2.1 BD Insights Queries

Figures 5 and 6 show the execution of complex and intermediate queries on the GPU. The numbers in the Y axis of both charts shows the end to end execution time of the query i.e. what a real user would see. As you can see, our DB2 BLU with GPU prototype has done a good job of improving the performance of complex queries. The complex queries are heavy queries that have costly operations like join, group by, aggregation, sort, predicate evaluation, etc. The current prototype performs the bulk of each query in the CPU and only a subset on the GPU. With that in mind, we are able to improve the total execution time of complex queries by 20% which is promising for this type of benchmark.

However, the performance of our prototype is very close to baseline (DB2 BLU without GPU) as depicted in figure 5. This may sound disappointing at first glance. However, after looking at the SQL of all these queries we found out that: 1) they have a small number of Group by, aggregation and sort queries 2) the execution time of most of the intermediate query is already very short in the baseline (the average query execution time is 30 sec). So there is really not much room for improvement for this set of queries. Moreover, if we offload the Group by, aggregation, and sort components of these queries to the GPU, the execution time of all the queries will be larger than the baseline. This is because by offloading such small components, we will incur the relatively higher cost of transferring data to and from the GPU. However, using the optimizations explained in sections 3 and 4, we can find out about these short queries in advance and try to process them in the CPU instead

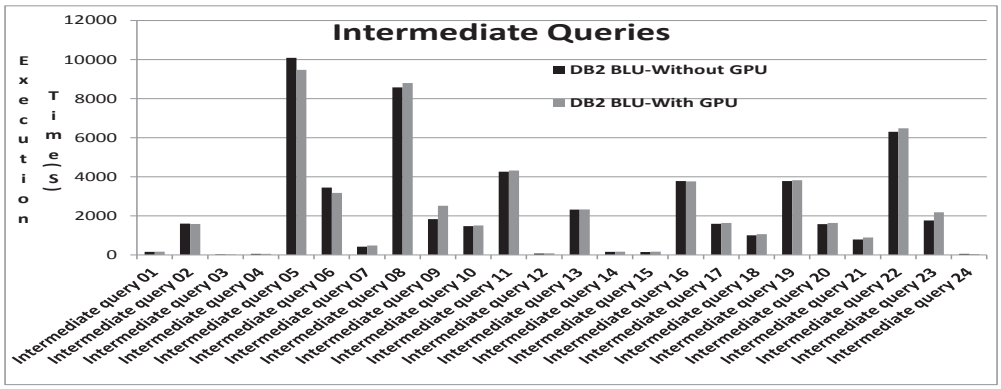

Figure 6: Intermediate queries in BD Insight Benchmark

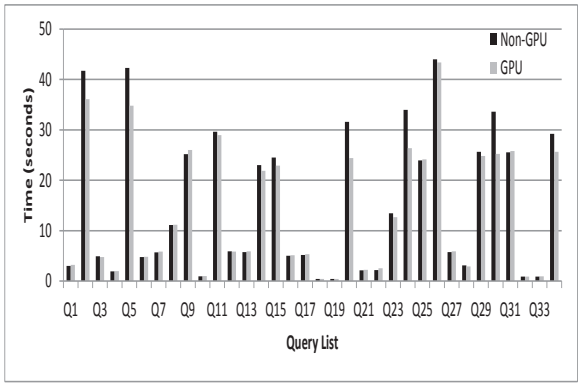

Figure 7: Query execution time for Cognos ROLAP benchmark

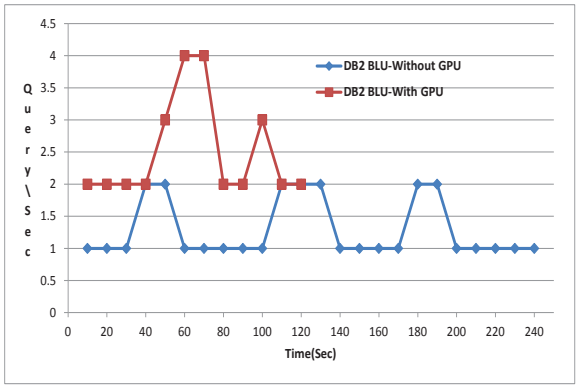

Figure 8: Concurrent query execution

of GPU. As a result, our prototype has performed well by keeping the execution time of all of these queries very close to the baseline. The remaining queries in the benchmark are simple queries that are very short (average is 150ms) and we do not send them to the GPU at all.

5.2.2 *Cognos ROLAP*

The Cognos ROLAP queries were run in two different modes: a serial run which aims to measure the query execution time, and a concurrent run where throughput was measured. Again, we compared our prototype (using GPU) with the standard DB2 BLU as the baseline (no GPU used). Table 2 shows the average serial execution of all 34 queries, with and without GPU. We run each query 5 times to eliminate the variation and get the average. As you can see, our GPU enabled prototype was able to save more than 8% of the total execution time. Figure 7 shows the per-query times during the run. Most of the queries take less time when GPU is used compared to the run with no GPU. The benef t of GPU off oading is apparent with longer running queries, but there is no benef t for shorter running queries (e.g. Q1 and Q4).

Table 2: Total query execution time for ROLAP benchmark

GPU On(ms)	GPU Off(ms)	GPU Gain
517133	474084	8.33%

We also measured the system throughput by running concurrent queries. Each connection thread continuously executed all the 34 Cognos ROLAP queries sequentially, and throughput (queries per Hour) is compared in Table 3. The throughput results show that the GPU benef ts are more pronounced in multi-user environments, rather than with higher degrees of parallelism. This is explainable because a single request launches a CUDA kernel and the parallelism in the GPU will be independent of the DB2 parallelism. However, more concurrent users means more parallel running CUDA kernels. So long as the GPUs have enough capacity to execute these kernels, DB2 can use the CPU capacity made available by off oading work to the GPUs.

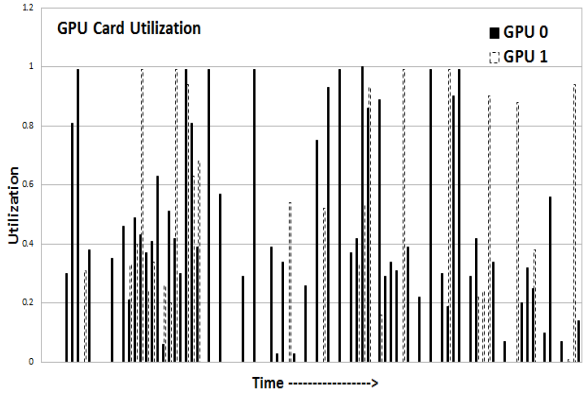

Figure 9: GPU utilization

Table 3: Throughput(Query/Sec for ROLAP benchmark)

#stream	#degree	GPU On	GPU Off	GPU Gain
1	24	403.96	385.51	4.79%
1	48	584.25	557.63	4.77%
1	64	630.9	602.1	4.78%
2	24	682.85	620.53	10.04%
2	48	868.09	773.46	12.23%
2	64	930.1	803.1	15.81%

5.3 Concurrent Query Execution

As already explained, we just run a subset of operations on the GPU in the current stage of our prototype i.e. queries that have group by, aggregation and sort operations. In order to better show how the GPU can help in accelerating the queries, we pick a set of queries where these three operations are represented heavily, moderately, or not at all. This test shows BLU with GPU acceleration handling cases of industry standard benchmark queries, handcrafted queries that would push the GPU to its limits, and queries where DB2 would decide to not use the GPU at all. It is a multi-user test to take advantage of CPU off oading and be more representative of real world workloads. The test was constructed using JMETER to have f ve thread groups of two threads each and thus 10 total users. The f rst three thread groups consist of complex queries sourced from the Cognos ROLAP workload that uses the GPU moderately (i.e. moderate use of group by, Aggregation and SORT processing) and a single BD Insights simple query that would not use the GPU. The fourth thread group consisted of two BD Insights complex queries (Complex Q1 and Q3) which would use the GPU moderately and one of the BD Insights simple queries which would not use the GPU. The f fth group contained two hand written queries that would perform group by and SORT on a large grouping set.

Figure 8 shows the elapsed time of running these queries with and without GPU. As you can see we can get almost a 2x speed up by using the GPU. This test demonstrates the impact of GPU acceleration for DB2 BLU analytic processing. The GPU heavy queries add a lot of processing cost to separate the groups and perform the aggregation for each group, as there are as many groups as there are rows in the table. Using a GPU kernel, this was off oaded and parallelized on an execution unit (2x K40s) that generally had the capacity to handle this type of work. The NVIDIA Tesla K40 has 2880 CUDA cores available for parallel batch jobs of a specif c nature (like group by), whereas the S824 had 24 cores that needed to handle all DB2 BLU processing. The GPU moderate queries would show about 15% benef t in standalone mode. The benef t comes from more parallelism in the CUDA kernel function than having DB2 sub-agent parallelism, which could be restricted due to degree, workload management, and CPU capacity. The Non-GPU queries performed similarly in both conf gurations, as would be the expectation. That leaves the majority of the benef t being due to off oad.

In f gure 9 we show the memory utilization for both GPUs that we have on our POWER 8 machine during the execution of queries. The GPU memory utilization characteristics for this workload shows a very spiky pattern. Clearly there is room for more GPU off-load, but it is also clear that at many points the workload is near GPU memory capacity. There were candidate queries for this test that were avoided simply due to the GPU memory restrictions, and no good way of handling them alternatively at the time.

6. CONCLUSION AND FUTURE WORK

In this paper, we have shown how we can improve the query processing time in DB2 BLU by using Nvidia GPUs. Moreover, we have provided realistic customer benchmarks on a commercial RDBMS with GPU acceleration, which has not been done before. We show a hybrid design that uses both CPU and GPUs for fast query processing. In our design we off oad specif c compute intensive operations like group by, aggregation and sort to GPU. Our main goals in doing this are: 1) To accelerate the computation of those specif c operations 2) To free up the CPU so that we can use it for other tasks. However, our experiments show that it is not always benef cial to off oad the operations to the GPU, because small data sets will perform optimally in the CPU. We have also discussed the necessary changes in the DB2 BLU runtime to integrate the GPUs. These changes included adding memory management, scheduling algorithms, and integration code for sort, group by, and aggregation operations. While we used Nvidia's fast sort kernel, we have also designed our very own group by/aggregation kernels. We show how we can use meta-data to tune the size of data structures in the GPU memory. As the memory in GPU is very limited, tuning the size of data structures is very important. Moreover, we demonstrated how we use meta-data to choose between these different kernels at run time to achieve the best performance.

Our performance results show a realistic representation of using GPU in a real RDBMS commercial tool. We show that the performance improvement was very signif cant for some queries, moderate for some other queries, and there were also queries that did not perform better when executed in the GPU. As mentioned, so far we have kernels for operations like group by, aggregation and sort. As one of our next steps, we would like to study the performance of other compute intensive operations (like join) on the GPU.

7. ADDITIONAL AUTHORS

Additional authors: Wayne Young (DB2, email: `wjyoung@ca.ibm.com`) and Chang Ge (DB2, email: `changge@ca.ibm.com`) and Geoffrey Ng (DB2, email: `Geoffrey@ca.ibm.com`) and Kajan Kanagaratnam (DB2, email: `kajanak@ca.ibm`).

8. REFERENCES

[1] *CUDA Atomic operations*. https://docs.nvidia.com/cuda/cuda-c-programming- guide/.

[2] *K-minimum values*. http://research.neustar.biz/tag/kmv/.

[3] *Murmur hashing algorithm*. https://en.wikipedia.org/wiki/MurmurHash.

[4] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake. Ocelot/hype: Optimized data processing on heterogeneous hardware. *Proc. VLDB Endow.*, 7(13):1609–1612, Aug. 2014.

[5] S. Breßand G. Saake. Why it is time for a hype: A hybrid query processing engine for eff cient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, Aug. 2013.

[6] D. Cederman and P. Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[7] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL restrictions and aggregations through fpga-based dynamic partial reconf guration. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013*, pages 25–28, 2013.

[8] W. Fang, B. He, and Q. Luo. Database compression on

graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, Sept. 2010.

[9] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[10] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[11] A. Greß and G. Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In *The 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, page 45, Apr. 2006.

[12] 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*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[13] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[14] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62, New York, NY, USA, 2012. ACM.

[15] T. Karnagel, M. Hille, M. Ludwig, D. Habich, W. Lehner, M. Heimel, and V. Markl. Demonstrating eff cient query processing in heterogeneous environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 693–696, New York, NY, USA, 2014. ACM.

[16] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[17] E. W. B. L. J. Gosink, K. Wu and K. I. Joy. Bin-hash indexing: A parallel method for fast query processing.

[18] 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.

[19] M. A. O'Neil and M. Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 7:1–7:7, New York, NY, USA, 2011. ACM.

[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] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, Aug. 2013.

[22] 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*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[23] Y.-C. Tu, A. Kumar, D. Yu, R. Rui, and R. Wheeler. Data management systems on gpus: Promises and challenges. In *Proceedings of the 25th International Conference on Scientif c and Statistical Database Management*, SSDBM, pages 33:1–33:4, New York, NY, USA, 2013. ACM.

[24] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, New York, NY, USA, 2014. ACM.

[25] R. Wu, B. Zhang, M. Hsu, and Q. Chen. Gpu-accelerated predicate evaluation on column store. In *Proceedings of the 11th International Conference on Web-age Information Management*, WAIM'10, pages 570–581, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and eff cient query processing on parallel cpu/gpu architectures. *Proc. VLDB Endow.*, 6(12):1374–1377, Aug. 2013.