Efficient Data Management for GPU Databases

Peter Bakkum
NEC Laboratories America
Princeton, NJ
pbb7c@virginia.edu

Srimat Chakradhar
NEC Laboratories America
Princeton, NJ
chak@nec-labs.com

ABSTRACT

General purpose GPUs are a new and powerful hardware device with a number of applications in the realm of relational databases. We describe a database framework designed to allow both CPU and GPU execution of queries. Through use of our novel data structure design and method of using GPU-mapped memory with efficient caching, we demonstrate that GPU query acceleration is possible for data sets much larger than the size of GPU memory. We also argue that the use of an opcode model of query execution combined with a simple virtual machine provides capabilities that are impossible with the parallel primitives used for most GPU database research. By implementing a single database framework that is efficient for both the CPU and GPU, we are able to make a fair comparison of performance for a filter operation and observe speedup on the GPU. This work is intended to provide a clearer picture of handling very abstract data operations efficiently on heterogeneous systems in anticipation of further application of GPU hardware in the relational database domain. Speedups of 4x and 8x over multicore CPU execution are observed for arbitrary data sizes and GPU-cacheable data sizes, respectively.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; H.2.4 [**Database Management**]: Parallel Databases

Keywords

GPGPU, CUDA, Databases, SQL

1. INTRODUCTION

Originally intended purely for graphics acceleration, graphics processing units, or GPUs, are now used for a vast array of interesting and challenging computational tasks. While the CPU is built to execute perhaps 4 or 8 threads simultaneously, GPUs are constructed from a fundamentally different perspective. By sacrificing complexity and complete thread independence, modern GPUs efficiently manage thousands of threads simultanously and allow the programmer to process data at throughputs over 100 gigabytes per second.

Increasingly, programmers are applying this power to problems outside the realm of graphics with general purpose graphics processing units, or GPGPUs, such as the NVIDIA Tesla hardware line. With no video output, these cards are intended solely for general computation. GPGPUs can accelerate certain applications by an order of magnitude [6], despite the fact that data must be transferred between main memory and GPU memory before processing occurs. Problems such as matrix multiplication, which has a high degree of parallelism, are ideal for GPU acceleration.

From a software perspective, GPU development is a low-level and difficult task, particularly for programmers inexperienced in handling high levels of parallelism. Development on NVIDIA GPUs is done in CUDA, an extension of the C programming language, and transformed with a proprietary compiler to PTX, an assembly language used with modern NVIDIA hardware. CUDA uses the stream programming paradigm; It executes a single kernel function simultaneously a massive number of times, with each call becoming a thread and handling an assigned chunk of data. Rather than a classic SIMD architecture, NVIDIA refers to its model of parallelism as single instruction, multiple thread, or SIMT. On the Tesla C2070 there are 448 simple cores organized into groups called streaming multiprocessors. When the kernel is executed the threads of execution are grouped into threadblocks and mapped to a streaming multiprocessor. Threadblocks are most efficient when an instruction is executed simultaneously across all member threads, but threads can diverge based on the data they process.

NVIDIA GPUs utilize a number of unique memory spaces. Global memory is the largest and has longest latency, sized at 6GB on the Tesla C2070. Register memory is associated with a thread/core and has the lowest latency, but is relatively small, so Local memory is a space used to overflow memory scoped at the thread level into global memory. Additionally, each streaming multiprocessor contains shared memory, a pool that can be written and accessed by any thread within the threadblock, enabling extremely efficient cooperation between threads. A drawback of GPU-managed memory is the fact that its global memory exists separately from that of the machines main memory, necessitating expensive memory transfers before this data can be processed on the GPU.

Perhaps the most powerful feature of the GPU architecture is memory coalescing. Coalescing occurs when every

thread in a threadblock accesses GPU global memory in a simultaneous and aligned pattern. The GPU hardware combines these accesses into a single memory fetch, concurrently feeding each core with data. This feature makes it possible to achieve memory throughput of over 100 GB/s on the Tesla line.

Our research attempts to utilize this new and powerful hardware to handle classic relational database management system (RDMBS) problems. Though some research has been conducted in this field, it focuses more on optimizing parallel data primitives rather than adapting RDBMSs to the GPU. Very few commercial databases use GPU acceleration in any respect, and no database exploits it to its full potential, despite the recent interest in high performing "NoSQL" databases, such as Cassandra, CouchDB, or MongoDB. In some sense, data processing software has yet to catch up with new and powerful GPGPU hardware [8]. The thesis underlying our work can be summarized simply: converging cutting-edge GPGPU research with traditional database technology advances both fields and produces impressive results.

The most important factor in writing efficient GPGPU applications is careful handling of data. The programmer's many options for moving data between GPU and main memory, in addition to the many memory spaces on the GPU itself, creates a large space of implementation possibilities. A certain data structure, for example, can prevent memory coalescing when moving data between GPU register and global memory, drastically reducing performance. Thus, intelligent implementation of GPU database acceleration involves rethinking of the database's entire structure.

Through implementation of a simple experimental database, this paper demonstrates solutions for a very general data structure called the Tablet (Section 3), an efficient mechanism for transferring this structure between the CPU and GPU (Section 4), an overall implementation guide and method of breaking computation into 'opcodes' (Section 5), and a discussion of why this method is superior to others (Section 6). Our database is limited to a class of SQL filter operations, for which we provide testing results, yet demonstrates many important GPU database concepts. Though many GPGPU research projects focus on execution performance and ignore data transfer to and from the GPU, our Tablet data structure and novel transfer mechanism are designed specifically for efficient end-to-end performance. Thus, the performance of our implementation proves that *GPUs can accelerate database operations on arbitrarily large data sets.*

We compare highly optimized GPU and multicore CPU implementations with a focus on demonstrating the fastest achievable query execution speeds on each under our data and workload models. To our knowledge, this remains the only published line of research that *specifically* examines handling database tasks through an opcode model of execution employed by most databases and easily accessible through SQL, rather than in the context of data parallel primitives such as `map`, `scatter`, or `reduce`. We believe that our results provide important insight into practical implementation of GPU-based databases that mimics the way many classic CPU-based databases are written.

Though we believe the GPU techniques we describe to be the most important results of this work, our testing results indicate the power of our approach. Execution of queries on the GPU shows speedups of at least an order of magnitude to single core CPU execution, and speedups of 4x and 8x for our mapped memory and cached memory implementations over multi-core CPU execution. These results are achieved on a SQL filter operation compiled to an intermediate opcode language that can be executed on either our CPU or GPU virtual machines, chosen by setting a simple flag. Ultimately, programmers use GPGPU processing to speed up their programs, and this class of database problem sees significant acceleration on the GPU.

2. RELATED WORK

This research continues the work published as *Accelerating SQL Database Operations on a GPU with CUDA* and *Accelerating SQL Database Operations on a GPU with CUDA: Extended Results* [1, 2]. These papers presented a project that re-implemented a segment of the SQLite database to enable certain queries to execute in parallel on the GPU rather than serially on the CPU. SQLite tranforms a SQL query into a program of opcodes executed with an internal virtual machine. By re-implementing the virtual machine as a CUDA kernel, certain SQL `select` queries, including aggregations, could be run on the GPU. The imlementation was tested with a battery of 13 queries run over 10 million rows of unindexed numerical data. An average running time speedup of 35 times was observed with GPU execution. Though this project's implementation shares no code with previous research, many ideas have been directly inherited and implemented as a standalone platform. In addition to this previous work, a handful of other researchers have experimented with GPU database processing relevant to databases.

The simplest method of GPU database access is through stored procedures. Many databases allow programmers to extend its functionality through user-defined functions or external procedures. These methods allow user-written code to directly manipulate data controlled by the DBMS, but do not make this extension transparent to the query-writer, meaning this extension must be explicitly called; It is not accessed during a vanilla database operation. One such effort extends an Oracle database to accelerate queries involving spatial operations, which have a high ratio of processing to I/O [3]. The authors concluded that a GPU external procedure could significantly accelerate this workload. Another article describes implementing a stored procedure in a PostgreSQL database that uses a CUDA program to rapidly generate random numbers, a common GPU-accelerated operation [21]. This procedure is accessed directly through SQL.

The majority of research into GPU acceleration of database functionality has been through a set of fairly standard parallel primitives. These operations, such as `sort`, `scan`, and `filter` are implemented as CUDA kernels and can be executed in succession, producing results much like a relational database. There is a direct correlation between many relational operations and this set of standard primitives; `filter`, for instance, is a type of database *selection*.

Beginning with more general predicate evaluation and aggregations [16], research has focused on finding the best GPU optimizations in each area. Joins are a vital database operations, and work has developed GPU-targeted nested-loop, sort-merge, and hash joins, observing significant speedups on GPU hardware [11, 19, 20, 31]. Research has also fo-

cused on the `scatter` and `gather` primitives [18, 19]. Other work has examined GPU acceleration for parallel search operations often performed within databases [23].

Sorting is another important area where GPUs have excelled. The GPU's unique architecture means that very specialized algorithms are required to achieve optimal execution speeds. Most algorithms are based on the radix sort method, often employing parallel scans or bitonic merges during the sorting process [11, 12, 15, 19, 20]. The most recent work in this area boasts sorting speeds of 482 million key-value pairs per second [26, 27].

Database indices have also been implemented and accelerated on the GPU. Some implementations use CSS-Trees, a type of cache-conscious index applicable to the GPU because it is stored as a flat array, enabling access through simple arithmetic rather than through pointers [11, 19, 29]. Research published recently claims that a method called binhash indexing is an even faster way to access indices on a GPU [14]. Importantly, significant speedup has also been shown for more traditional B+ Trees, demonstrating the outperformance of the GPU in an important piece of the modern database [13].

The `scan` operation, often called a parallel prefix-sum, sets every element n in array B to the sum of elements $1..n$ in array A. It is an important piece of many data processing operations, such as sorting, and has been widely researched and accelerated on GPUs. Implementations attempt to optimize the process by utilizing the GPU's shared memory and using novel parallel forms of aggregation for each cell of the destination array [9, 19, 25].

The popularity of the MapReduce programming paradigm paradigm has also spurred GPU development, adapting a familiar framework to the powerful GPU platform [7, 17, 24]. MapReduce frameworks such as Hadoop have been used to replace traditional databases in certain applications, though they are generally more applicable to workloads with unstructured data. The inherent parallelism of this approach is normally exploited in clouds of distributed machines, but it proves a natural match for the GPU architecture. These frameworks are much simpler than a full RDBMS, and thus do not encounter many of the issues associated with developing a framework like ours.

Several research projects have developed a higher level framework built upon the traditional parallel primitives that manages overall query execution [19, 32]. These manage a query plan as a directed graph of discrete operations, such executing a primitive or moving data from main memory to the GPU or vice versa, that in its entirety represents a course of action for the query. This model has been somewhat inherited from the distributed computing world, where it can be used to assign independent segments of the query plan to separate machines to run in parallel. In the GPU context, this framework separates out the memory transfers and primitive executions, sometimes in multiple branches, allowing a query optimizer to calculate the cost of transferring data to the GPU versus the benefit of the accelerated primitive [30]. It also allows division of labor between the CPU and GPU. In this paper we argue that the usual implementation of this pattern on the GPU, with a CUDA kernel representing a node in the graph, is sub-optimal.

In September 2010 a German software company, empulse GmbH, introduced ParStream, a database capable of exploiting GPU hardware. ParStream is a distributed column-oriented database intended for exceptionally fast queries of billions of records [10, 22]. Like many research designs, ParStream's query optimizer breaks the query into a directed graph of segments, called query nodes, which it then intelligently assigns both between separate machines and on the heterogeneous level between the CPU and GPU. ParStream uses a custom column-oriented bitmap index capable of fitting into GPU memory, and empulse advertises that it can handle climate research queries over 3 billion rows in as little as 100ms. It uses the GPU only for index and filter operations, however, leaving the door open for future research and development with other operations. ParStream's development supports our thesis that GPU-based databases will soon become impossible to ignore, given their exceptional speed and low cost.

3. TABLETS

We have carefully designed a data structure, the Tablet, to flexibly handle information on the GPU. This name was chosen because of the similarity to the vertically-partitioned tablets used in Google's BigTable [5]. The data structure used during query execution has significant bearing on overall execution speed and the relative speeds of CPU versus GPU execution. Thus, we give our data structure thorough treatment. We intend our data structure to be read-optimized and efficient for both CPU and GPU execution.

The tablet's most basic feature is vertical partitioning of table data; Records are split into fixed-size groups of rows. Accessing an entire table of data may involve multiple tablets, but accessing a single row of data involves only a single tablet. Vertical partitioning is useful in the context of heavily distributed database by enabling efficient management of data between networked machines. In our implementation, however, tablets are useful because they vastly simplify the process of moving data to and from GPU memory. Tablets allow the GPU to operate exclusively on known-size chunks of records which can be transferred serially in succession to the GPU or streamed to overlap with kernel execution.

GPUs are not able to process tree data structures efficiently because of their necessary lack of parallelism at the levels near the tree's root. A data structure in which each core locates its data without communication or traversal of a tree proves much more applicable to GPU execution. Additionally, memory access coalescing is necessary for efficient GPU execution since coalesced accesses can be as much as an order of magnitude faster than uncoalesced accesses [28]. Coalescing requires that memory accesses from a threadblock be adjacent or at a small fixed interval, a requirement that necessitates fixed-size data records. Some research has focused on applying CSS trees to GPU data processing, since it maintains its leaves at fixed-size intervals, but we have not examined their use in our implementation [11, 19, 29].

While GPUs use coalescing to reduce memory accesses, CPUs take advantage of their cache heirarchies. A fair comparison of the two architectures utilizes both, and we design our data structure specifically for this purpose. Our data structure has been influenced by cache-conscious database design, notably MonetDB, which stores records in column-major form [4]. This means that data items within different records but the same column are stored adjacent to one another. Thus, data columns within the same record are separated. This organization's efficiency lies in the fact that

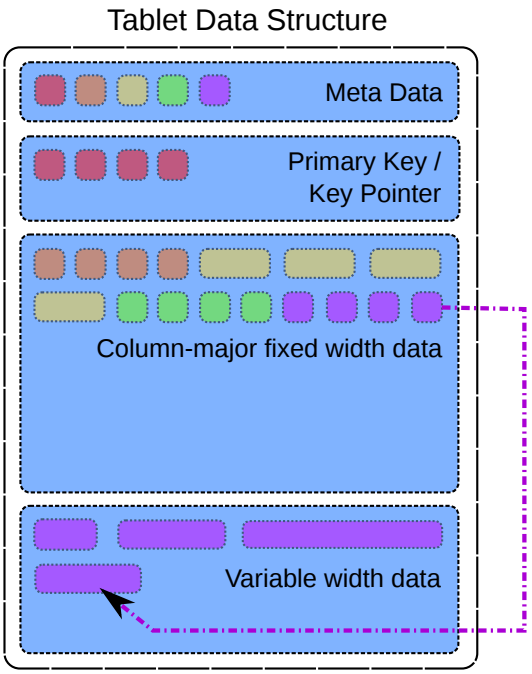

Figure 1: The tablet is divided into a section of meta data, a section of primary keys and pointers, a section of fixed width data for fast and efficient reading, and an area of variable width data accessed through a relative pointer.

some of the most of the memory access intensive operations of a database examine elements of a column in succession. If an entire block of column data is loaded into a cache line then a column in multiple records can be accessed without a cache miss. Consequently, we use a column-major organization for our data. Note that the column major format simultaneously targets both GPU coalescing and CPU cache consistency.

An orthogonal problem addressed by the tablet is the issue of handling queries executed on variable-sized data, such as strings or the value of a key-value pair. Though the GPU is less efficient relative to the CPU on this type of workload, ideally the programmer would make his own choice about where to handle any query. There are two reasons that variable-sized data processing is difficult and expensive on the GPU. First, accesses to this data can not be coalesced, since this requires fixed-size intervals between access locations. Second, variable-sized data objects such as strings are often stored separately from relevant fixed-size data and accessed through a pointer. This makes it difficult to process the data on the GPU, since this pointer is not valid within the GPU's memory space. Thus, these kinds of pointers to variable-size data must be explicitly managed when tranferring information to the GPU to ensure that pointers resolve to the correct data, a tedious process.

Tablets address this problem by allocating a portion of the total tablet space for variable-size data and requiring all pointers to this data to be relative to the start of the tablet, rather than relative to the start of the memory space. In other words, variable-sized data is accessed through a pointer stored on the tablet that points to another location on the same tablet, making it completely self-contained and memory-space agnostic. When a tablet is transferred between main and GPU memory, both fixed and variable-size data is moved simultaneously. When variable-size data is accessed during a query, it is a two step process. First, the pointers to the data are retrieved in a coalesced access from the fixed-size area. These pointers locate the variable-sized data relative to the start of the tablet, which are then accessed in the second step. Thus, variable-size data processing is possible even when moving tablets between memory-spaces and when only a portion of the database's records can be stored in GPU memory at a moment in time.

Figure 1 is a visual overview of our implementation of the tablet concepts described above. Each tablet has a fixed size chosen at compile time (intended to be in the range of around 4 to 128MB) and four strictly defined areas described below.

Meta Data The meta block is a fixed size area that contains identifying information about the table membership of the tablet, the sizes of the other three areas, and the types, sizes, and names of the primary key and columns contained in the tablet. Our tablets support only vertical partitioning, and thus the number of columns is capped and the column meta-data has a fixed size.

Primary Key The primary key area holds the primary key of the table, along with a pointer that can be used to refer directly to variable-size information, making it possible to employ this data structure as a key-value store.

Fixed-Size Data The fixed-size data area holds the tablet's information that has a known size, such numerical data, in column-major form. Thus information from a single column is adjacent and accesses can be coalesced.

Variable Data The variable-size data block holds information such as strings with unknown sizes. While fixed-size data records are accessed based on the key location, the variable-size area is accessed through a relative pointer, either from the key pointer or from a pointer stored as a fixed-size data column. Figure 1 shows such a column in purple.

Note that the key area of the tablet is sized corresponding to the number of records allowed in the tablet, but the remaining area can be allocated to fixed or variable-size data based on the character of the tablet's information.

4. TABLET MANAGEMENT

The overarching problem with processing large amounts of data on the GPU is that it has limited memory space, thus managing this space is essential. Though we designed our tablet structure specifically to handle transmission between the CPU and GPU, there are a number of ways to actually implement this transmission. Data transfer is such a large component of total GPU processing time that any overlap between transfer time and kernel execution time can significantly accelerate a query. We must also manage the transfer of query result data off of the GPU, leading to a difficult problem of bi-directional data flow.

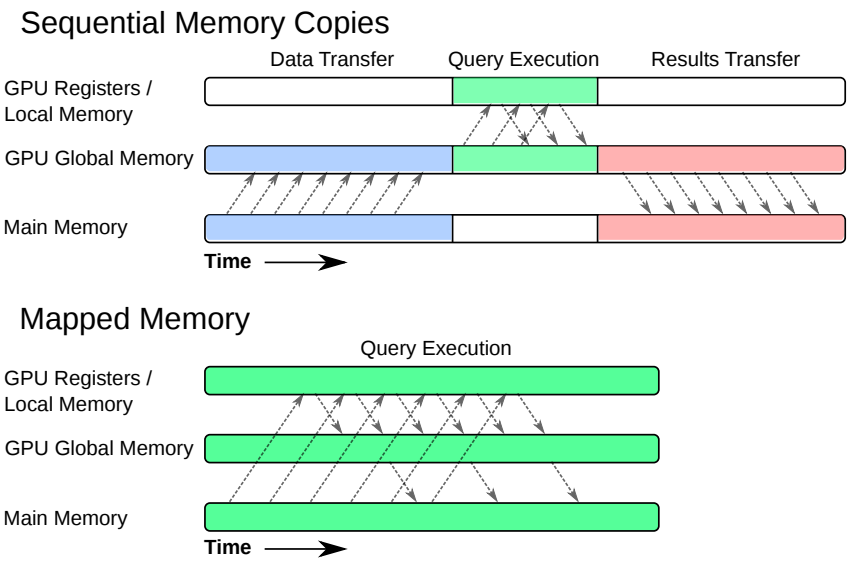

Figure 2: Mapped memory removes GPU global memory as an intermediate step in the data transfer, but buffers there to guarantee coalesced writes of results. Data and results transfer occur while data is being processed rather than in separate steps.

The efficient movement of information between main memory and GPU memory is a somewhat arbitrary restriction of current hardware. There is little reason not to expect that future hardware will include machines with Tesla-like GPUs that share global/main memory with the CPU or even exist on the same die as the CPU. This scenario already exists on current NVIDIA ION motherboards, which have CUDA capable GPU processors embedded directly on the board, using system's main memory as their CUDA global memory. These GPUs, however, are not nearly as powerful as general purpose GPUs such as the Tesla C2070. Our results indicate a machine in which a powerful GPGPU could access main memory with a latency similar to GPU global memory can only improve the execution time advantage of GPU query processing. Large data management would be simpler under such a scheme, and this development could push GPGPU technology closer to the mainstream.

Serial transfer of data and results is the simplest memory management scheme. There are three distinct steps in this configuration: moving data to the GPU, executing the query over this segment of data, and transferring the results of the query back to main memory. If performed serially, most of the total execution time is spent waiting on data transfers. We use this as our baseline for GPU execution time.

The next management option is asynchronous streaming of data to and from the GPU. The CUDA API provides the capability of defining several streams of execution that run asynchronously. Each step in a stream is dependent on the one before it, but streams are independent of one another. In particular, streaming was designed to allow memory transfers to occur while a kernel executes. If we assume that either the query data or the results fit entirely into GPU memory then there is a significant advantage to using the streaming API. Assuming kernel execution is the quickest step (which it is with our test queries), it fits within the time needed for memory transfers as the streaming API overlaps them. Thus, the query execution time becomes roughly the time it takes to transfer data on to, or results off of, the GPU. Unfortunately, our assumption that neither the data nor the results fit entirely in GPU memory means that both data and results transfers must be included in the streaming.

This sort of simultaneous bi-directional data transfer complicates things, since our tests indicate that current CUDA technology is either unable to exploit the full bidirectional nature of the PCI bus or unable to schedule pending data transfers and kernel executions effectively enough to significantly outperform simple serial execution. Based on our tests it appears the streaming API schedules asynchronous tasks based on when they were added to the streams, rather than checking at runtime which streams are ready to run considering current memory transfers. With these restrictions, little kernel execution overlap occurs. Even with an optimal streaming configuration that overlaps data transfer with kernel execution as much as possible, this method would outperform serial execution only as much as eliminating the data transfer time of either the data or results transfer, whichever is shorter. Thus, our next option for data management outperforms even the best streaming.

The final, and ultimately best, option for handling tablets during execution is mapped memory. The CUDA API provides a method for allowing the GPU to map a portion of main memory onto the device, provided the memory has been declared as pinned. Pinned, or page-locked memory, is an allocation that the operating system can not swap out of memory to disk, hence it is guaranteed to be at a certain location. Mapped memory means that a kernel can directly access pinned information in main memory with no transfers. Mapped memory accesses must travel across the PCI bus, and thus are significantly slower than accesses to GPU global memory, particularly if uncoalesced. Our tests, however, have found that executing a kernel that uses mapped memory is faster than the aggregate execution time of a program with memory transfers before and after the kernel execution. This works because the GPU is extremely

efficient at swapping out information-starved threadblocks for threadblocks ready for execution. The mapped memory method can be used for both data and results transfers, and our tests have shown that it performs 2 to 3 times faster than serial and streaming data transfers.

The results transfers are more complex than the data transfers, however. Since we have carefully aligned the *data* columns to 64-byte locations and the threads per block is a power of two, all of these accesses our easily coalesced. The results however, are neither in order within the threadblock nor 64-byte aligned, and consequently not naturally coalesced when writing to mapped memory. According to the CUDA documentation and our own tests, unaligned but adjacent out-of-order memory writes to GPU global memory are coalesced. However, based on our testing it appears that mapped memory has more conservative requirements for coalescing. Writes to mapped memory that are unaligned or out-of-order take at least an order of magnitude longer. Thus, we use a lazy, two-step procedure to write results back to mapped memory.

Our two-step results write procedure is designed to guarantee that all writes to mapped memory are coalesced. We assume that when we execute the opcode that handles writing results that certain threads within the threadblock are 'valid,' in that parts of the data row with which the thread is associated will be written to the results block; Only the valid rows will need to perform writes. We perform an atomic scatter operation within the threadblock by using CUDA's `atomicAdd()` operation on a variable in shared memory, thus establishing both an area for each thread to write and the total number of valid rows within the threadblock. This is more efficient than a shared-memory scan operation because it is not necessary to guarantee that each thread writes its results in order, and we access shared memory only as many times as we have valid rows. We then atomically increment a global variable of the total number of result rows output to this point, thus allocating ourselves a block of GPU global memory for the current threadblock.

Once allocated, we take advantage of the relaxed coalescing requirements of the GPU-resident memory to perform an initial write. It proceeds with each thread writing to its area assigned in the scatter operation. We call the `__threadfence()` function to ensure data has reached global memory and atomically increment a counter making note of this. Essentially, this process writes a variable number of rows onto a grid of threadblock-sized data areas. After incrementing the counter, we check if a threadblock-sized area has been filled with result rows. If so, each thread copies data from this area to mapped memory, which transfers data back to main memory. Since this is both in-order and aligned to a multiple of the threadblock size, we guarantee that these writes are coalesced. Thus, as we write results we perform lazy copies to mapped memory only as they are needed, efficiently overlapping these writes with the execution of other threadblocks.

In addition to significantly improved query performance, we emphasize that effective tablet management of queries eliminates the size restriction of GPU global memory. Whether transferred serially, streamed, or mapped into GPU memory, breaking table data into chunks and managing multiple transfers during query execution means that GPU global memory is re-used during the query process; We do not assume that data is already on the device. Most importantly we emphasize the following point: *the results for the relative speed of GPU query execution are identical for arbitrarily large table datas and query results.* This means that this class of SQL `SELECT` queries is no longer dependent on GPU memory size. In fact, in the mapped memory case, we only need to explicitly allocate slightly more than a tablet size of global memory to handle query execution, independent of the size of the table being processed.

5. IMPLEMENTATION

Our model of query execution separates the query plan from the management of data and target execution architecture (either the CPU or GPU). The query plan is stored as a sequence of *opcodes*, which we call an opcode program or statement. Execution of the opcodes and state management is performed by the *virtual machine*. Each opcode represents a distinct operation that can range from extremely simple and granular to complex and reminiscent of the primitives discussed previously. Opcodes can have up to 3 integer arguments and 1 argument of any type. The virtual machine interperates these arguments to change the effect of the opcode and the locations from which data affected by the opcode is retrieved and to which it is stored. The structure of the opcodes is similar to assembly code, and concepts such as registers and jumping to instructions are carried over and added to the advanced data parallelism of our model. Opcodes serve as the building blocks of each query.

Since opcodes serve as a lower-level representation of a query, the high-level representation, SQL, must be compiled into this new format. Our compiler parses SQL, identifies columns drawn from data table records and derived expressions, handles conditions placed on this query by a `WHERE` clause, and finally, uses a code generator to output a program of opcodes. The output somewhat resembles assembly, with the exceptions that programs are executed over data managed by the virtual machine, and individual opcodes are implicitly parallel over each row of the table. Values drawn from columns are treated as expressions that can be manipulated with math opcodes such as `Add` and combined with constant values or other columns. Conditions are formed by comparing the values of two expressions with an opcode such as `Lt` (less than) or `Ge` (greater than or equal to). If this result evaluates to true, then we jump to another opcode later in the program, otherwise falling through to the next opcode. In this way the structure of `AND`s and `OR`s of a SQL statement's `WHERE` clause can be represented opcodes. Our SQL compiler must also manage the allocation of virtual machine registers and their data types, ensuring that opcodes operate on the proper pieces of data.

The opcodes are transparent to both data type and destination architecture. This means that our opcodes have been explicitly designed to execute on either the CPU or the GPU with no change at the opcode level. In fact, each opcode has been implemented twice, once in a C function and once in a CUDA kernel. Each virtual machine is essentially a giant switch statement. It maintains a program counter and executes a certain block of code based on the opcode value.

An example opcode from our implementation is `Column`, which loads data from a column for a given row and stores it in a virtual machine register, a location in memory used by opcodes for intermediate results. This loaded value can then be compared to another register's value with the `Lt` opcode, which jumps to a certain opcode elsewhere in the program

if one register's value is less than the other's, creating data-based divergence. This destination opcode could be `Result`, which writes data to a result tablet for output as the query's result. Note that data type is transparent to these opcodes. Parallelism for executing instructions over an entire table is started with the `Parallel` opcode, which the virtual machine handles by jumping to a lower level virtual machine (a C function for CPU execution or a CUDA kernel for GPU execution) that executes subsequent opcodes in parallel.

In addition to being GPU-friendly, our model of parallel opcode execution combined with column-major tables means that our CPU virtual machine is enormously cache efficient. Though we do not explicitly use the processor's vector operations, which have been proven to significantly accelerate certain queries[35], we consider this type of execution to be SIMD, since each opcode executes over a block of rows. The column-major data format means that data in this SIMD block can be moved simultaneously with `memcpy()` since it is adjacent, can fit into a single cache line accessed in a tight inner loop.

A major advantage of CPU query execution over GPU execution is the capability of the CPU to perform indirect jumps, i.e., to jump to an instruction who's location is stored in a variable, rather than in the program itself. In the model that we have adopted, each opcode must be switched to in order to execute. On the CPU, this is accomplished with an explicitly defined jump table. The jump table is an array that maps the parsed opcode value to the opcode's location within the program, so each opcode is accessed in constant time[1]. Though the newest version of the PTX assembly language describes instructions for indirect jumps, our experimentation indicates these have not yet been implemented, and are thus not yet functional. Instead, the virtual machine must use a switch and compare an opcode with each possible value. This means that as many as n comparisons could be required, where n is the total number of opcodes. This limitation of current hardware means that the GPU has needless overhead in this type of abstract query processing, since we see no fundamental architectural reason for no indirect jump instruction. We expect that future implementation of this feature would increase GPU acceleration.

Current GPU hardware seems to be targeted more towards specific data applications rather than the type of abstract data processing presented here. One area this is demonstrated is with the behavior of `__syncthreads()`. This function causes a thread to block and wait for other threads in the same threadblock to catch up and synchronize. However, others have noted that `__syncthreads()` has significant undocumented behaviors [33]. It appears that `__syncthreads()` on current hardware waits only for threads that have followed an identical code branch. In other words, if there is data-driven thread divergence, and one group of threads executes the function, the kernel does not block as expected, but rather only the branched threads block. This becomes an issue with our opcode model because certain threads diverge and jump over opcodes. Thus we are forced to have every thread move in lockstep over each opcode in the opcode program, even if some do not execute every opcode's logic. This has a small effect on GPU performance and makes the kernel needlessly complex.

[1]We define this process explicitly, though many compilers now make this optimization automatically.

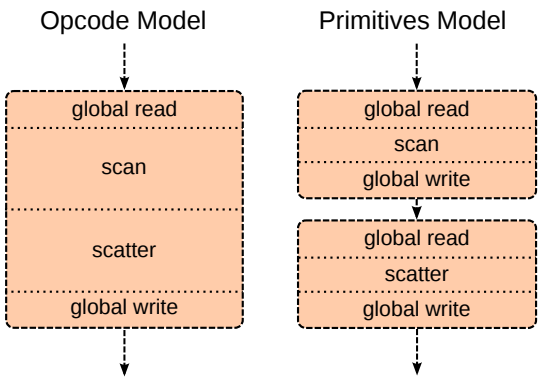

Figure 3: Separately executed primitives can be grouped together in the same kernel invocation under our opcode model of query execution.

6. OPCODES VS. PRIMITIVES

The difference between our opcode model and the primitives model of many research projects is the location of the kernel boundaries. Our query plans execute multiple opcodes within the same kernel, whereas most GPU parallel primitives are implemented as a black-box CUDA kernel. Executing a query plan with these primitives involves a kernel invocation for each primitive. For the purposes of discussion we will assume this is true when discussing the "primitive model."

We believe the opcode model of execution is fundamentally superior because of the very nature of stream programming. In this context, stream programming refers to the succession of data moved through the processing elements of the GPU. Because it is a stream, there is necessarily no retention of register or shared memory between kernel calls. The ending of one kernel and the calling of the next represents a global synchronization of the GPU, and in fact is the only way to ensure complete synchronization between threadblocks in the CUDA programming paradigm. Thus data must be written to the GPU's global memory in order to be retained between kernel calls. In many cases however, it is unnecessary to synchronize globally and write data that will just be read again in the next kernel call. In effect, the entire stream is being unnecessarily cleared between primitives.

The primitives model is unnecessarily restrictive, and the cost of this restriction is additional memory accesses which result in poorer performance. Our alternative opcode model is this: primitives need not be split into separate kernels. We place all of our GPU code in a single kernel and access it through opcodes. Thus, we retain intermediate state between execution of primitives and perform global synchronizations only when absolutely necessary. Ultimately this leads to more efficient code while retaining the abstract nature of classic primitives.

An excellent example of this limitation of primitives is given in *Revisiting Sorting for GPGPU Stream Architectures*, which describes the current state-of-the-art optimal GPU sorting procedure [26]. The sorting operation consists of a binning operation, several intemediate scans, and a scatter operation. The report notes that certain operations, such as the final scan operation and subsequent scatter can be ex-

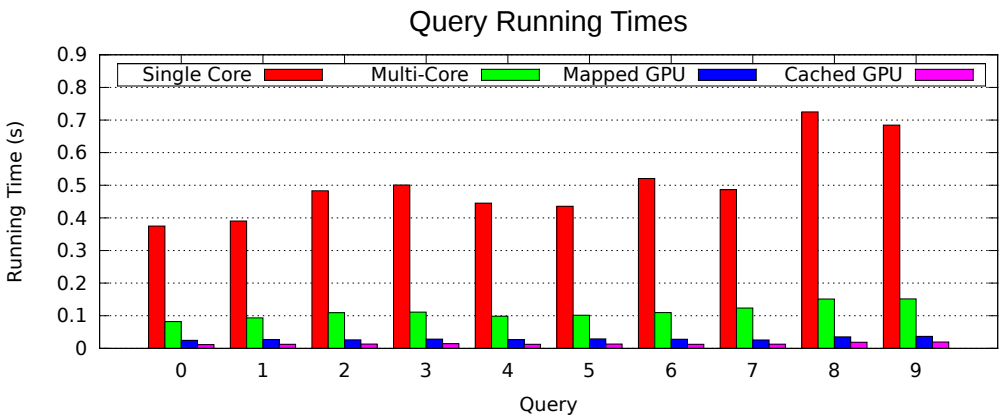

Figure 4: Queries demonstrated consistent speedup on the GPU, especially when assuming data and results reside on the GPU.

ecuted in the same kernel, using what the authors refer to as the "visitor pattern," conceptually identical to our opcode system. The advantage here is that "the overall number of memory transactions needed by the application is dramatically reduced because we obviate the need to move intermediate state (e.g., the input/output sequences for scan) through global device memory."

In addition to efficient memory handling between classic primitives, our opcode pattern also allows a wide range in operation granularity. Not only can complex primitives such as `scan` be fit into this model, but the extremely fine-grained operations such as `Column`, and `Add` that we describe earlier fit comfortably into this system. Provided there is a virtual machine to manage the intermediate data associated with these operations, it is trivial to call assembly-like operations adjacent to primitives with arbitrary complexity, performing global synchronization only when it is required. We expect that future improvements to GPU query processing operations will be forced to use this opcode pattern to best the current state-of-the-art applications.

7. TESTING

Testing was performed using a 8 million row randomly generated numerical dataset. The columns consisted of an integer primary key and 2 columns each with a random distribution in [-100,100], a normal distribution with a sigma of 5, and another with a sigma of 20. Each of these distributions was generated once each for a 32-bit integer column and a IEEE 754 32-bit floating point column. The GNU Scientific Library was used to ensure the quality of the random distribution. The results shown are for an NVIDIA GTX 570 GPU, which has the latest generation Fermi architecture, and a 3.2 GHz Intel Core i7 CPU with 4 hyperthreaded cores, supporting 8 possible hardware threads.

We divide our execution configurations into the following categories.

Single Core Execution using a single CPU core.

Multi-Core Execution using multiple CPU cores, up to the 8 hardware threads possible on our test machine.

Serial Execution on the GPU where data a tablet is transferred to the device, the query is executed, and the results are transferred off the device. This process occurs serially for multiple tablets.

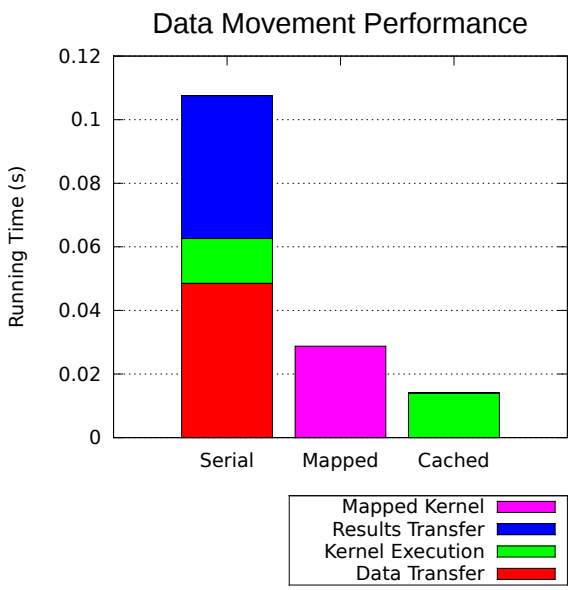

Figure 5: Streaming kernel execution far outpaces serial execution, though faster faster speeds can be achieved if data and results are cached on the GPU and no transfers are required.

Mapped Execution on the GPU where main memory is mapped onto the device for faster data access and results writes.

Cached Execution that assumes data and results can remain resident on the GPU. This is identical to serial execution with the data and results tranfer times removed. Since GPU memory is limited, these results are not possible for arbitrary data sizes.

Figure 5 demonstrates the advantage of using mapped data access. During each of our ten test queries multiple tablets must be pushed through the GPU for processing.

	Single	Multi	Mapped	Cached
Integer	0.510	0.110	0.028	0.014
Floating Pt.	0.499	0.116	0.029	0.015
All	0.505	0.113	0.029	0.014

Table 1: Running times in seconds for CPU single and multi-core and GPU mapped and cached executions shown for floating point and integer arithmetic queries.

	Over Single-core		Over Multi-core	
	Mapped	Cached	Mapped	Cached
Integer	18.125	37.512	3.919	8.111
Floating Pt.	16.995	34.383	3.955	8.002
All	17.547	35.895	3.937	8.054

Table 2: Speedup of mapped and cached GPU implementations over single and multi-core GPU implementations.

The serial execution bar shows the total time spent tranferring data and results to and from the device averaged across these 10 queries, demonstrating that memory transfers consume the majority of execution time. Using mapped memory obviates the need for these transfers as separate steps, instead including them in the kernel execution time. This roughly doubles the kernel run time, but the total mean query time is reduced significantly. The cached execution assumes that both data and results are small enough to be resident on the GPU, and thus the expensive transfer time is avoided.

Figure 4 visually presents the query running times observed. While single-core CPU execution took an average of .51s, multi-core execution was predictably much faster, with an mean running time of .11s. Both the mapped and cached GPU implementations saw running times faster still, with .03s and .01s means, respectively. The odd numbered queries used mostly integer arithmetic, while the subsequent even numbered queries had identical query plans and expected results sizes, but used mostly 32-bit floating point arithmetic. Thus, comparing these pairs provides interesting insight into the relative performance of these operations on both the CPU and the GPU.

Table 1 shows the mean running time in seconds for these categories, while Table 2 shows the mean speedup of the GPU tests against the CPU. Both GPU tests shown performed faster than the highly optimized multi-core implementation, demonstrating the capability to accelerate these database operations with GPGPU hardware. Note also that for the mapped GPU implementation, this speedup applies to arbitrarily large data sets, while the cached implementation assumes that data and results fit into device memory.

Figure 6 shows the growth of running time as a function of the data size, averaged over the 10 queries in our suite. Multi-core execution experiences irregular growth because of different levels of CPU data saturation. We assign a tablet to a thread and limit the number of threads to our CPU's possible hardware threads, which in this case is 8. When 8 tablets are processed, each is assigned a thread and finishes processing in a similar timeframe. When a 9th tablet is added, however, we wait until a thread finishes processing its first tablet before assigning it a second, significantly increasing execution time. Thus, the step pattern observed is a function of the maximim tablet size; With smaller tablets the steps would be more frequent but more overhead would be incurred.

8. FUTURE WORK

Our implementation has been designed partly to demonstrate a very general framework for GPU data processing. Using this framework, a next step is to implement and test additional database features, such as joins and indices, proven in other literature to be applicable to the GPU. Modern RDBMSs are extremely complex, and much more work in this area is required to fully replicate this functionality in a GPU-friendly manner. We believe our opcode framework will be adaptable to this additional functionality, with modifications to our virtual machine as appropriate to facilitate inter-opcode communication.

Though we took great care ensuring our data structures could expand to handle variable-size data such as strings, processing these efficiently on the GPU is an entire research area in itself that deserves more thorough work to implement and investigate performance. Our tablet data structure has also been designed to be abstract enough to function as a simple key/value store by simply associating a relative pointer with each fixed-size key. Under this model the structured column area of the tablet has a size of zero. Though processing these kinds of abstractly large data objects is more challenging because of the GPU's architecture, past research has convincingly demonstrated acceleration for certain text processing applications [34].

Another interesting expansion would be to examine multi-GPU and GPU/CPU concurrency. The NVIDIA Tesla S2050 Server, the current state-of-the-art NVIDIA server solution, fits four dedicated Fermi-based GPUs into a standard 1U server. Dividing a single query among several GPUs would not only increase the processing power relative to the amount of data processed, but would also increase the total amount of data that could be cached in GPU global memory, up to a possible 24 GB over 4 GPUs. Additionally, GPUs could also handle disparate queries concurrently. Though we have not had time to experiment with such configurations, our tablet data structure naturally invites partitioning over multiple memory spaces and execution environments. Additionally, processing data on the CPU concurrently with the GPU would also increase total productivity. The Fermi generation of NVIDIA architecture makes handling multiple CUDA contexts much easier, and we expect future innovations in this area. We firmly believe that such implementations are the natural software realization of the massive processing power now available on the GPU.

A recurring feature the programmer discovers when experimenting with the unique and raw computing ability of the GPU is that even minor tweaks and additions can significantly change program performance. For example, we arrived at our 128 threads per CUDA block configuration through experimentation over our battery of test queries. This configuration is influenced by the specific structure of the Tesla C1060, the memory access intensity of our query kernel, the amount of shared memory necessary for certain operations, and many other seen and unseen factors. It is very possible that on future hardware, or even on specific queries, this value and other configuration values like it will be sub-optimal. Future research could include both re-optimizing for other hardware, or developing models that

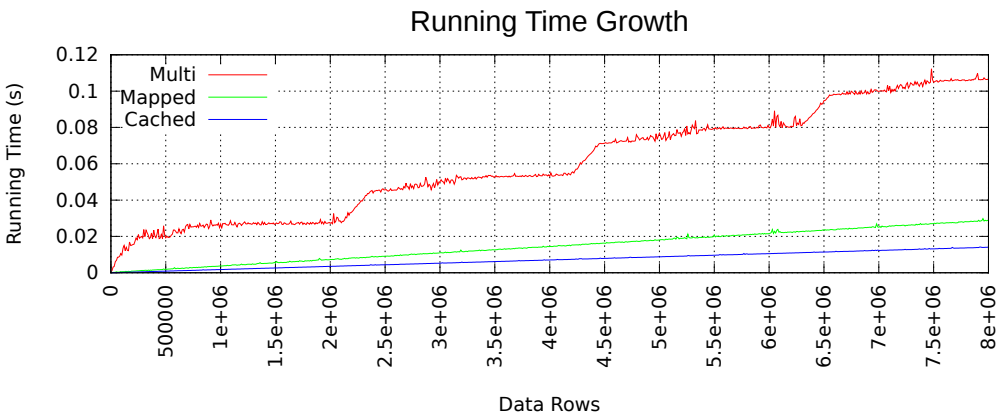

Figure 6: Multi-core execution increases more irregularly than mapped or cached execution on the GPU, which exeriences almost linear growth in execution time.

attempt to predict the optimal configuration values based on hardware architecture and expected query characteristics.

A major effort of this work has been to prove that GPU data processing is limited more by hard disk speed and main memory size than by the bandwidth between main and GPU memory, as is the case with virtually all databases. For our test we assumed that the data fit completely into main memory and attempted to optimize transfer to and from GPU memory. Future research could attempt to improve the total latency of transfers from disk to GPU memory. Another possibility is that future GPU hardware is able to access the disk more directly, which would open up a host of other possibilities for acceleration. Regardless of the direction, it is clear that this general area of GPU application development is ripe for further research.

9. CONCLUSION

The simple fact is that database software development has yet to catch up with the new capabilities of GPGPU hardware; This research attempts to advance understanding of how GPUs can accelerate certain RDBMS operations. The tablet data structure has been combined with the two-step mapped memory reading and writing technique to demonstrate that memory transfers with GPU memory are not a major obstacle to GPU data handling. Thorough examination of our opcode model of execution shows that it allows the programmer to choose any granularity for database operations in conjunction with our relatively simple virtual machine, while also enabling more efficient data handling than is possible with the parallel primitives used in many other research projects.

A speedup of 4x over multi-core CPU query execution was observed for arbitrarily large data sizes, with a speedup of 8x when assuming that data and results can be cached in GPU global memory. Given the rapid development of cheap and powerful GPUs, we expect this relative advantage of the GPU to increase. We also expect a significant amount of research and development in applying GPUs to databases in both the academic and commercial arenas. Though the GPU is a new and complex device, its incredible power and the major challenges faced in processing huge amounts of data means that it will inevitably become a much more important piece of general data processing in the near future.

10. REFERENCES

[1] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, New York, NY, USA, 2010. ACM.

[2] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA: Extended results. Technical Report CS-2010-08, University of Virginia Department of Computer Science, May 2010.

[3] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1021–1032. VLDB Endowment, 2004.

[4] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51:77–85, December 2008.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *In proceedings of the 7th conference on USENIX symposium on operating systems design and implementation - volume 7*, pages 205–218, 2006.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention*, Dec 2004.

[8] A. di Blas and T. Kaldeway. Data monster: Why graphics processors will transform database processing. *IEEE Spectrum*, September 2009.

[9] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on

graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 205–213, New York, NY, USA, 2008. ACM.

[10] empulse GmbH. ParStream – turning data into knowledge. White Paper, November 2010.

[11] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. GPUQP: query co-processing using graphics processors. In *ACM SIGMOD International Conference on Management of Data*, pages 1061–1063, New York, NY, USA, 2007. ACM.

[12] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. Hel, Q. Luo, P. V. Sander, and K. Yang. Parallel data mining on graphics processors. Technical report, Hong Kong University of Science and Technology, 2008.

[13] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided B+ Tree searches on a GPU with CUDA. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, June 2011.

[14] L. Gosink, K. Wu, W. Bethel, J. Owens, and K. Joy. Bin-hash indexing: A parallel GPU-based method for fast query processing. Technical Report LBNL-728E, Lawrence Berkeley National Laboratory, 2008.

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

[16] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 206, New York, NY, USA, 2005. ACM.

[17] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[18] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[19] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):1–39, 2009.

[20] 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.

[21] T. Hoff. Scaling postgresql using cuda, May 2009. `http://highscalability.com/scaling-postgresql-using-cuda`.

[22] M. Hummel. ParStream – a parallel database on GPUs. GPU Technology Conference, San Jose Convention Center, CA, September 2010.

[23] T. Kaldeway, J. Hagen, A. Di Blas, and E. Sedlar. Parallel search on video cards. Technical report, Oracle, 2008.

[24] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, New York, NY, USA, 2008. ACM.

[25] D. Merrill and A. Grimshaw. Parallel scan for stream architectures. Technical Report CS-2009-14, University of Virginia Department of Computer Science, December 2009.

[26] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS-2010-03, University of Virginia Department of Computer Science, February 2010.

[27] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[28] NVIDIA. *NVIDIA CUDA Programming Guide*, 2.3.1 edition, August 2009. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`.

[29] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[30] N. Satish, N. Sundaram, and K. Keutzer. Optimizing the use of GPU memory in applications with large data sets. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 408 –418, December 2009.

[31] C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration for spatial selections and joins. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 455–466, New York, NY, USA, 2003. ACM.

[32] N. Sundaram, A. Raghunathan, and S. T. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[33] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235 –246, March 2010.

[34] Y. Zhang, F. Mueller, X. Cui, and T. Potok. GPU-accelerated text mining. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM 2009)*, March 2009.

[35] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.