Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission

Haicheng Wu*, Gregory Diamos†, Jin Wang*, Srihari Cadambi‡, Sudhakar Yalamanchili*, Srimat Chakradhar‡
*Sch. of ECE, Georgia Institute of Technology, Atlanta, GA. {hwu36,jin.wang,sudha}@gatech.edu
†Nvidia Research, Santa Clara, CA. gdiamos@nvidia.com
‡NEC Laboratories America, Princeton NJ. {cadambi,chak}@nec-labs.com

***Abstract*—**
Data warehousing applications represent an emergent application arena that requires the processing of relational queries and computations over massive amounts of data. Modern general purpose GPUs are high core count architectures that potentially offer substantial improvements in throughput for these applications. However, there are significant challenges that arise due to the overheads of data movement through the memory hierarchy and between the GPU and host CPU. This paper proposes a set of compiler optimizations to address these challenges.

Inspired in part by loop fusion/fission optimizations in the scientific computing community, we propose *kernel fusion* and *kernel fission*. *Kernel fusion* fuses the code bodies of two GPU kernels to i) eliminate redundant operations across dependent kernels, ii) reduce data movement between GPU registers and GPU memory, iii) reduce data movement between GPU memory and CPU memory, and iv) improve spatial and temporal locality of memory references. *Kernel fission* partitions a kernel into segments such that segment computations and data transfers between the GPU and host CPU can be overlapped. Fusion and fission can also be applied concurrently to a set of kernels. We empirically evaluate the benefits of fusion/fission on relational algebra operators drawn from the TPC-H benchmark suite. All kernels are implemented in CUDA and the experiments are performed with NVIDIA Fermi GPUs. In general, we observed data throughput improvements ranging from 13.1% to 41.4% for the SELECT operator and queries Q1 and Q21 in the TPC-H benchmark suite. We present key insights, lessons learned, and opportunities for further improvements.

***Keywords*-data warehousing; relational algebra; GPU; compiler; optimization;**

I. Introduction

The use of programmable GPUs has appeared as a potential vehicle for high throughput implementations of data warehousing applications with an order of magnitude or more performance improvement over traditional CPU-based implementations [1], [2]. This expectation is motivated by the fact that GPUs have demonstrated significant performance improvements for data intensive applications such as molecular dynamics [3], physical simulations [4] in science, options pricing [5] in finance, and ray tracing [6] in graphics. It is also reflected in the emergence of accelerated cloud infrastructures for the Enterprise such as Amazon's EC-2 with GPU instances [7].

However, the application of GPUs to the acceleration of data warehousing applications that perform relational queries and computations over massive amounts of data is a relatively recent trend [8] and there are fundamental differences between such applications and compute-intensive HPC applications. One of the factors that make the use of GPUs challenging for data warehousing applications is efficient GPU implementations of basic database primitives, e.g., relational algebra operators. A second challenge that is fundamental to the current architecture of GPU-based systems is the set of limitations imposed by the CPU-GPU memory hierarchy, as shown in Figure 1. Internal to the GPU there exists a memory hierarchy that extends from GPU core registers, through on-chip shared memory, to off-chip global memory. However, the amount of memory directly attached to the GPUs is limited, forcing transfers from the next level which is the host memory that is accessed in most systems via PCIe channels. The peak bandwidth across PCIe can be up to an order of magnitude or more lower than GPU local memory bandwidth. Data warehousing applications must stage and move data throughout this hierarchy. He et al. observed that 15-90% of the total execution time is spent in moving data between CPU and GPU when accelerating database applications [2]. Consequently there is a need for techniques to optimize the implementations of data warehousing applications considering both the GPU computation capabilities and system memory hierarchy limitations.

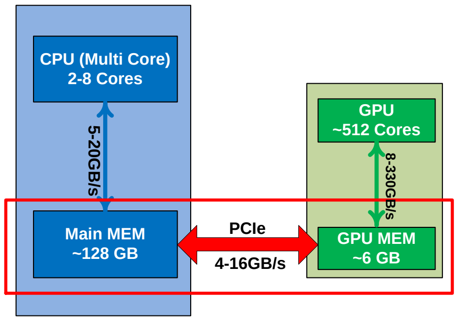

Figure 1: Memory hierarchy bottlenecks for GPU accelerators

To address the data movement overheads described above, we propose and demonstrate the utility of *kernel fusion* and *kernel fission* in optimizing performance of the memory hierarchy. Specifically, *kernel fusion* is analogous to traditional loop fusion and reduces transfers of temporary data through the memory hierarchy and reduces the data footprint of each kernel. Such transformations also increase the textual scope of compiler optimizations. *Kernel fission* is a transformation explicitly designed to partition data parallel kernels into smaller units such that data transfers between host and GPU can be fully overlapped. Both fusion and fission can be applied concurrently to the set of kernels. In this paper, we henceforth refer to kernel fusion and kernel fission as "fusion" and "fission" respectively.

This paper demonstrates the impact of kernel fusion and fission for optimizing data movement in patterns of interacting kernels found in the TPC-H benchmark suite. The goal of this paper is to provide insight into how and why fusion/fission works with quantitative measurements from actual implementations. The fusion and fission transformations are manually performed on CUDA implementations of operators mimicking a compiler-based optimization. However, the CUDA kernels themselves are *not* manually optimized after fusion/fission. Thus we expect that results reported in this work reflect the potential of the automated implementation within our compiler framework which is under development.

II. Relational Algebra Operators

Relational algebra (RA) operators can express the high level semantics of an application in terms of a series of bulk operations on relations. These are the building blocks of modern relational database systems. Table I lists the common RA operators and a few simple examples. In addition to these operators, data warehousing applications perform arithmetic computations ranging from simple operators such as aggregation to more complex functions such as statistical operators used for example in forecasting or retail analytics. Finally, operators such as SORT and UNIQUE are required to maintain certain ordering relations amongst data elements or relations. Each of these operators may find optimized implementations as one or more CUDA kernels. All of these kernels are potential candidates for fusion/fission. *Demonstrating that this is indeed the case and understanding and quantifying the advantages of fusion/fission is the goal of this paper.* It should be noted that the kernel mentioned here is different from the concept of CUDA kernel in a way that one operator kernel may have more than one CUDA kernel depending on its implementation.

A. Common RA Kernel Combinations

TPC-H [9] is a decision support benchmark suite that is widely used today. It is comprised of 22 queries of varying degrees of complexity. The queries analyze relations between customers, orders, suppliers and products using complex data types and multiple operators on large volumes of randomly generated data sets.

We perform a detailed analysis of the TPC-H queries to identify commonly occurring combinations of kernels. These combinations are potential candidates for fusion/fission. From the 22 queries in TPC-H, Figure 2 illustrates the

UNION	x = {(3,a), (4,a), (2,b)}, y = {(0,a), (2,b)}
	union x y $\rightarrow$ {(3,a), (4,a), (2,b), (0,a)}

INTERSECTION	x = {(3,a), (4,a), (2,b)}, y = {(0,a), (2,b)}
	intersection x y $\rightarrow$ {(2,b)}

PRODUCT	x = {(3,a), (4,a)}, y = {(True,2)}
	product x y $\rightarrow$ {(3,a,True,2), (4,a,True,2)}

DIFFERENCE	x = {(3,a), (4,a), (2,b)}, y = {(4,a), (3,a)}
	difference x y $\rightarrow$ {(2,b)}

JOIN	x = {(3,a), (4,a), (2,b)}, y = {(2,f), (3,c)}
	join x y $\rightarrow$ {(3,a,c), (2,b,f)}

PROJECTION	x = {(3,True,a), (4,True,a), (2,False,b)}
	project [0,2] x $\rightarrow$ {(3,a), (4,a), (2,b)}

SELECT	x = {(3,True,a), (4,True,a), (2,False,b)} select [field.0==2] x $\rightarrow$ (2,False,b)

Table I: Examples of RA operators (the first field is the "key")

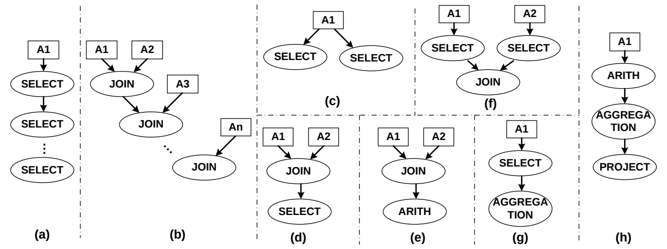

Figure 2: Common operator combinations to fuse.

frequently occurring patterns of operators. In the figure, (a) is a sequence of back-to-back SELECT operators that perform a filtering operation, for instance, of a date range, (b) is a sequence of JOIN operations that create a large table consisting of multiple fields, (c) represents the case when different SELECT operators need to filter the same input data, (d) and (e) are examples that perform SELECT or arithmetic operators with two fields generated by a JOIN, (f) corresponds to the JOIN of two small selected tables, (g) performs AGGREGATION on selected data and (h) is a common computing pattern, for example, the total discounted price of a set of items using $\sum(1 - discount) \times price$. PROJECT in (h) discards the source of the calculation and only retains the result. The above patterns can be further combined to form larger patterns that can be fused. For example, (e) can generate the input of (h). We will use Figure 2(a) as an example to explain the benefits and motivation of kernel fusion and kernel fission. In particular, we observe that SELECT occurs very often in these patterns. Therefore, we first focus our efforts in the SELECT operator.

Our work utilizes the optimized CUDA implementations of RA kernels from Diamos et al. [10] which is based on partitioning algorithms into stages. Figure 3 shows the four stages of the SELECT operator. The first stage partitions the input data into smaller chunks, each of which is handled by one Cooperative Thread Array (CTA) [11]. In the second stage, the threads in each CTA filter elements in parallel. Next, the unmatched elements are discarded and the rest

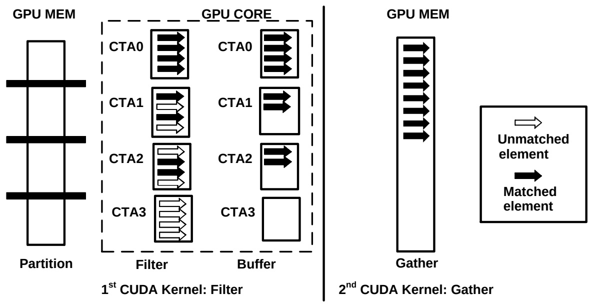

Figure 3: Selection in GPUs

CPU	2 quad-core Xeon E5520 @ 2.27GHz
Memory	48 GB
GPU	1 Tesla C2070 (6GB GDDR5 memory)
OS	Ubuntu 10.04 Server
GCC	4.4.3
NVCC	4.0

Table II: Experiment Environment

are buffered into an array. Finally, in the fourth stage, the scattered, matched results are gathered together in the GPU main memory. A global synchronization is needed before the gather step so that the filtered results can determine their correct position in the final array. Thus, the first three stages are implemented in one CUDA kernel and the final gather stage is in a second CUDA kernel.

We first quantify the raw GPU advantage and then address the impact of the PCIe bandwidth bottleneck. Table II lists the experimental environment used to generate the results reported in this paper. Figure 4(a) illustrates the relative performance of a basic SELECT operator between an NVIDIA C2070 GPU (PCIe transfer time excluded) and a dual quad-core CPU, the latter using 16 CPU threads to parallelize the operation. This is performed over random 32-bit integers. The parameters listed in the figure (10%, 50%, 90%) indicate the fraction of data selected from the inputs. The top three lines correspond to the GPU and the bottom three correspond to the CPU performance. On average, the GPU implementation is **2.88x**, **8.80x** and **8.35x** faster respectively for 10%, 50% and 90%. The figure also shows that the less data selected, the better performance on both the GPU and CPU due to the fact that less result data need to be written back. Other RA operators have the similar speedup when executed in GPU.

The PCIe bandwidth, as measured by using the scaled *bandwidthTest* of NVIDIA CUDA SDK 4.0 is shown in Figure 4(b) and is much smaller than its theoretical value (8GB/s) due to various hardware and software overheads. Pinned memory (memory that cannot be swapped to disk) exhibits higher bandwidth but when the data size becomes large, its advantage reduces because of the lower OS performance caused by large amount of pinned memory.

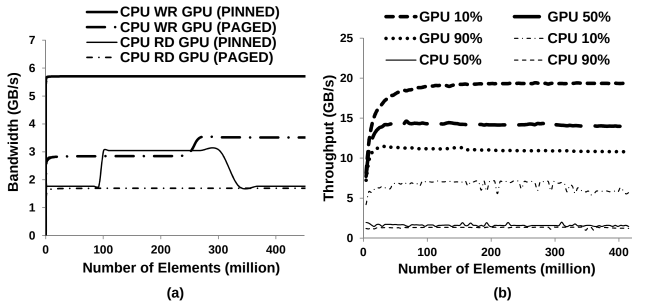

Figure 4: (a) The Performance of SELECT (GPU vs. CPU); (b) PCIe 2.0 bandwidth Measurement

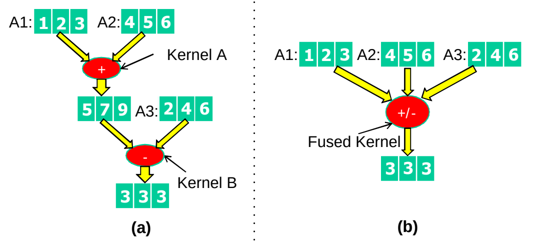

Figure 5: Example of kernel fusion

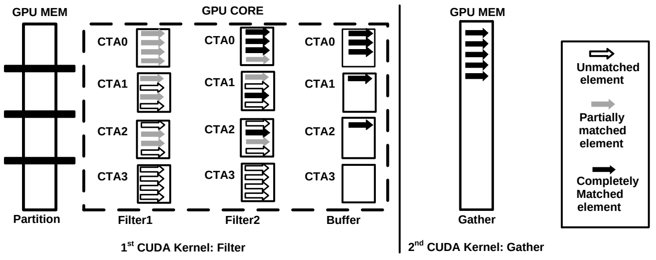

Figure 6: Fused Back-to-back Selection

From this simple experiment we observe that the GPU computational throughput rates are much higher than what the PCIe bandwidth will support. While the GPU compute capacity can maintain a high data rate of up to 20 GB/s for SELECT (Figure 4(a)), the PCIe bandwidth (Figure 4(b)) can effectively only supply data at a 2X-4X slower rate. Thus GPU capacity cannot be fully utilized. Gregg et al. provide a detailed analysis of this phenomena [12].

III. Kernel Fusion

Kernel fusion is designed to reduce the impact of the limited PCIe bandwidth. Figure 5 is an example of kernel fusion. Figure 5(a) shows two dependent kernels - one for addition and one for subtraction. After fusion, one single functionally equivalent new kernel (Figure 5(b)) is created. The new kernel directly reads in three inputs and produces the same result. Figure 6 illustrates the fusion of two back-to-back SELECT operations on the GPU. Compared to Figure 3, a second filter stage is inserted after the first filter stage in the original kernel to compute the second SELECT operation. The remaining stages remain the same.

A. Benefits of Kernel Fusion

Kernel Fusion has six benefits as listed below (Figure 7). The first four stem from creating a smaller data footprint

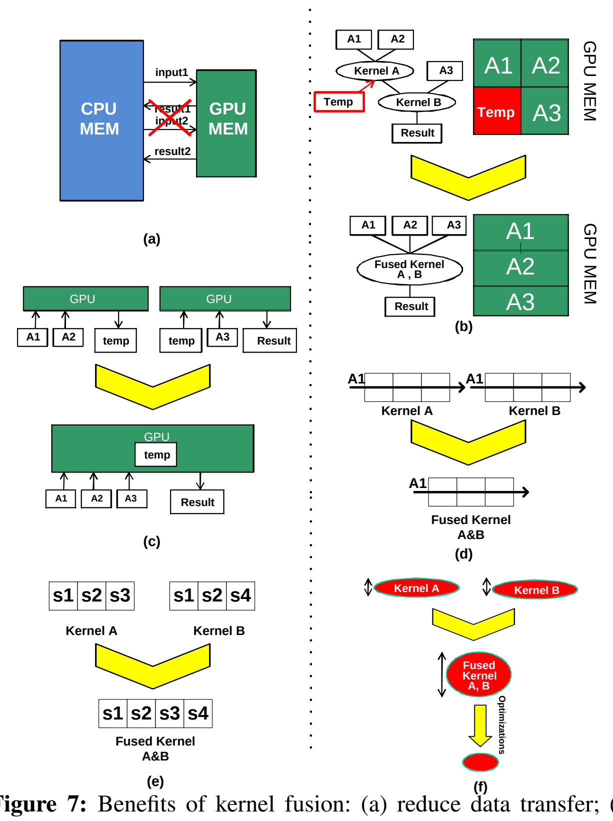

Figure 7: Benefits of kernel fusion: (a) reduce data transfer; (b) store more input data; (c) less GPU Memory access; (d) improve temporal locality; (e) eliminate common stages; (f) larger compiler optimization scope

through fusion, while the other two relate to increasing the compiler's optimization scope.

Smaller Data Footprint results in benefits:

Reduction in PCIe Traffic: Since kernel fusion produces a single fused kernel, there is no intermediate data (Figure 7(a)). In the absence of fusion, if the intermediate data is larger than the relatively small GPU memory, or if its size precludes storing other required data, the intermediate data will have to be transferred back to the CPU for temporary storage incurring significant data transfer performance overheads. For example, if kernels generating *A3* in Figure 5(a) need most of the GPU memory, the result of the addition has to be transfered to the CPU memory, and subsequently transfered back to the GPU before the subtraction can be executed. Fusion avoids this extra round trip data movement.

Larger Input Data: Consider very large data sets, e.g., 10X the size of GPU memory. Several transfers will have to be made between the CPU and GPU memories to process all of the data. If intermediate data does not have to reside on the GPU (as a result of kernel fusion), more memory is available to store input data on the GPU which can lead to a smaller number of overall transfers between the GPU and CPU (Figure 7(b)). This benefit grows as the applications working set size grows.

Reduction in GPU Global Memory Accesses: Kernel fusion also reduces data movement between the GPU device and its offchip main memory (Figure 7(c)). Fused kernels store the intermediate data in GPU registers (shared memory or cache), which can be accessed much faster than the offchip memory. Kernels which are not fused have a larger cache footprint necessitating more off-chip memory access.

	Statement	Inst # (O0)	Inst # (O3)

not fused	if (d<THRESHOLD1)
	if (d<THRESHOLD2)
		5×2	3×2

fused	if (d<THREASHOLD1 && d<THREADSHOLD2)	10	3

Table III: The impact of kernel fusion on compiler optimization

Temporal Data Locality: Kernel fusion can also reduce array traversal overhead and brings data locality benefits. The fused kernel often needs to access every array element once while kernels not fused need to perform accesses in each kernel, i.e., multiple times (Figure 7(d)). Moreover, fused kernels make better use of the cache, but kernels that are not fused may have to access off-chip GPU memory if the data revisited across kernels is flushed.

Larger Optimization Scope creates a larger body of code that the compiler could optimize and brings two benefits:

Common Computation Elimination: If two kernels are fused, the common stages of computations are sometimes redundant and can be avoided. For example, the original two kernels in Figure 7(e) both have stages S1 and S2 which need to be executed only once after fusion. As for the SELECT operator, the fused kernel only needs one partition, buffer and gather stage (Figure 6).

Improved Compiler Optimization Benefits: When two kernels are fused, there are larger bodies of code which is advantageous for almost all classic compiler optimizations such as instruction scheduling, register allocation, and constant propagation. These optimizations can speedup the overall performance (Figure 7(f)). Table III compares the speedup of using *O3* flag to optimize before and after fusion for a very simple, illustrative example. Without fusion, the two filter operations are performed separately in their own kernels (row 1, column 2). After fusion the two statements occur in the same kernel and are subject to optimization (row 2, column 2). The third and fourth columns show the number of corresponding PTX instructions produced by the compiler when using different optimization flags. Before optimization, the fused kernel has 5 more instructions than without fusion (10 vs. 5). Using compiler optimizations without fusion can reduce **40%** instruction count (from 5 to 3), while optimizing a fused kernel achieves a higher **70%** instruction reduction (10 down to 3). This simple example indicates that significant reduction in instruction counts are possible when applied to larger code segments.

In data warehousing applications, there are opportunities to apply kernel fusion across queries since RA operators from different queries can be fused. Further, fusion can be extended across multiple kernels, for example a chain of SELECT operators. In this case only one gather stage is needed. Thus the benefits increase with the number of kernels fused. However, the extent to which kernels can be fused is not clear since kernel fusion does increase register pressure. This analysis is the subject of ongoing work.

B. Measurements With Kernel Fusion

This section uses back-to-back SELECT operators as an example to demonstrate the benefits of kernel fusion described in the previous section. Three methods of running the SELECTs are evaluated: *with round trip*, *without round trip*, and *fused*. *With round trip* runs two SELECTs separately transferring the input data from the CPU to the GPU and the result data back to the CPU for *each* SELECT operation. Thus, this method needs to transfer data via PCIe four times for two SELECTs. *Without round trip* is similar except that it retains the intermediate result generated by the first SELECT in the GPU main memory. The third method, *fused*, copies the input to the GPU, launches a single fused kernel for SELECT, and copies the result back to the CPU. In practice, *With round trip* is very inefficient, but it has to be used when there is insufficient space on the GPU for storing the intermediate results of the executed kernels. Unless mentioned explicitly, all the SELECTs from this section onwards filter 50% of the input elements. Thus two back-to-back SELECTs keep 25% of the original data. Performance is measured in terms of the data throughput that can be achieved. The input data to all three methods are still randomly generated 32-bit integers representing compressed row data.

Figure 8(a) compares the performance of these three methods. On average, the throughput of *fused* is **49.9%** larger than *with round trip* and **6.2%** larger than *without round trip*. Its main advantage over *with round trip* is that large PCIe transfer times are avoided. It is better than *without round trip* because traversals to GPU memory are avoided. When only considering the GPU computation, *fused* is **79.9%** better than *without round trip* on average (Figure 8(b)).

Figure 9 breaks down the execution time of the three methods into three parts: input/output time (the time to transfer the initial input and final result between CPU and GPU), round trip time (the time to transfer the intermediate temporary result) and the computation time taken by GPU device. The figure shows that the PCIe time dominates the total execution time no matter which method is used. Comparing three methods, the input/output time is the same for all three methods since they transfer the same amount of data. The round trip time, which causes the difference between *without round trip* and *with round trip*, is **54.0%** of total time of *with round trip* and can be eliminated by kernel fusion. The difference between *without round trip* and *fused* represents the reduction in computation due to fusion.

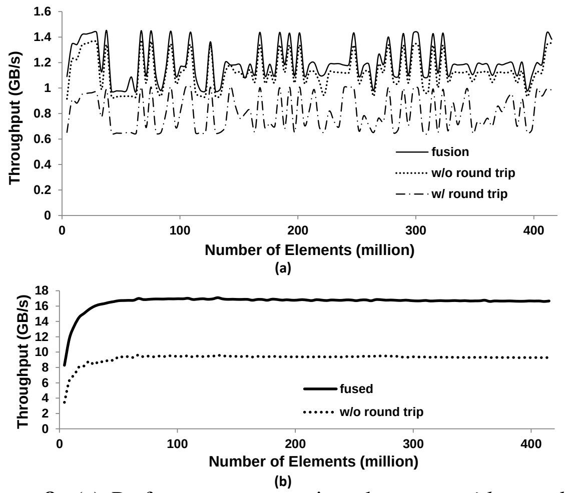

Figure 8: (a) Performance comparison between *with round trip*, *without round trip*, and *fused*; (b) Computation part comparison between *without round trip* and *fused*.

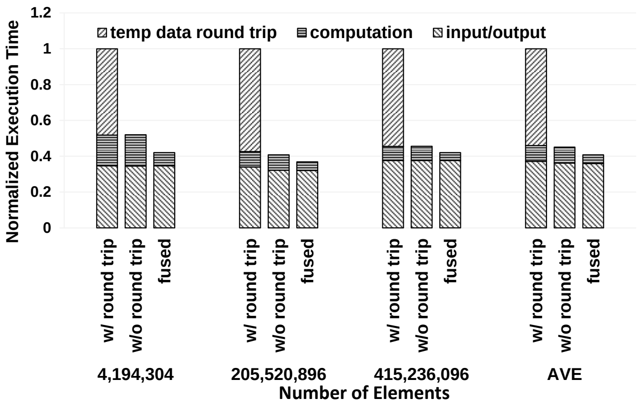

Figure 9: the breaking down execution time with different number of elements (normalized to *with round trip*)

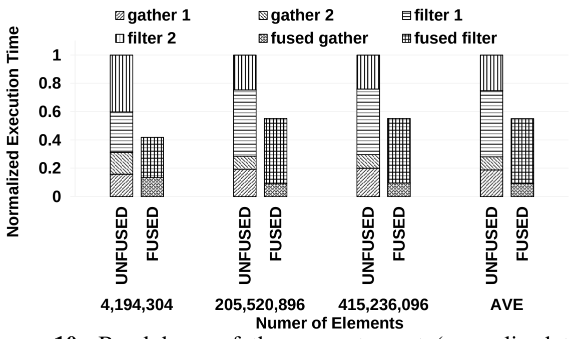

Figure 10: Breakdown of the compute part (normalized to the unfused execution time.)

Figure 10 further breaks down the computation time into the time taken by each CUDA Kernel: filter (including partition, filter and buffer of Figure 6) and gather. On average, the fused filter is **1.57x** faster than using separate filters and fused gather is **3.03x** faster than separate gathers. Benefits 3–6 of the previous section is responsible for the improvement with kernel fusion.

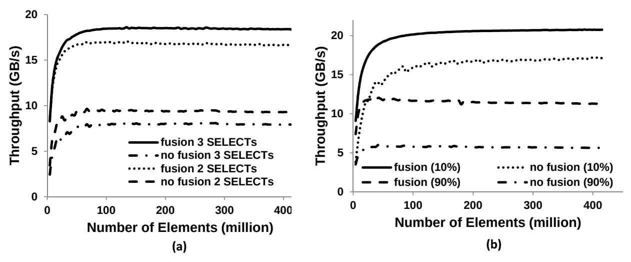

Figure 11: (a) Sensitivity to the number of kernels to fuse; (b) Sensitivity to the data selection rate

Figure 11(a) checks the sensitivity of kernel fusion to the number of fused kernels by comparing the GPU performance of fusing three versus two SELECTs. On average, fusing three SELECTs achieves a **2.35x** larger throughput while fusing two kernels achieves **1.80x**.

Last, Figure 11(b) records the sensitivity of the performance improvement to the percentage of the elements that are selected. The figure shows that the benefits of kernel fusion increase with the fraction of data selected. The reason is that data movement optimization has a more drastic effect when there is more data.

C. Automating Fusion

This section addresses some of the algorithmic requirements of fusion. The described automation process is a domain specific approach that is based on the knowledge of the detail behavior of each operators. Specifically, we address two key issues: how to judiciously select kernels to fuse, and how to automate the process of kernel fusion.

The first key problem is to discover feasible combinations of kernels to fuse via compiler analysis followed by the selection of the best options. Data dependence analysis is necessary to discovers candidate kernels to fuse. Two kinds of dependence may exist between two kernels: i) each element in the result array of the consumer kernel only depends on one element of its input array which is generated by the producer kernel (e.g. Figure 2(a)), or ii) the consumer kernel has to wait until the completion of the entire producer kernel. (e.g. Figure 2(b)). These two cases must be treated differently: i) is easier since the dependence between two arrays can decomposed into dependence between two scalars. As to ii), domain specific knowledge has to be used. For example, JOIN-JOIN can be fused, but SORT-JOIN cannot. In the latter case, the SORT must be completed before the JOIN can be performed. In particular, SORT and UNIQUE cannot be fused with any other operators.

The choice between alternative fusion opportunities is guided by a cost function that evaluates the potential benefits of fusion. As illustrated in Section III-B, fusing more kernels usually enhances performance. However, fusing too many kernels may cause problems. The main reason is that kernel fusion will created increased register (and shared memory) pressure since each thread has to store more intermediate data within the GPU. This can increase spill code or have adverse cache effects.

Automating kernel fusion is relatively straightforward if the GPU implementations of the kernels use multi-stage algorithms such as the four stages used by the SELECT. A new fused kernel is generated by interleaving the stages of different kernels following certain rules. All RA algorithms uses partition as the first stage and gather as the last stage. Thus, the first stage of a fused kernel is to partition all the input data, and the last stage is gathering the result from different CTAs. The middle stage of a fused kernel corresponds to computation stages such as the two filters in the SELECT example. Automatic code generation for this stage requires the dependence graph of RA operators. A topological sort of the graph determines the execution sequence of these RA operators. The computation stage of each RA operator produces results in temporary registers (or shared memory) and the next operator can use these results as input. The combination of two filters in Figure 6 is a good example.

Finally, besides the multi-stage structure, there are two other requirements for fusion -i) providing a unified data structure to access different types of data, and ii) using the same kernel configuration (CTA number and thread number) for the operators.

All of the preceding operations required of Kernel fusion can be performed in the source code level with the help of tool such as ROSE [13] or in the AST level by using Ocelot [14]. Moreover, kernel fusion is a general cross-kernel optimization that can also be applied to CPU programs. Thus, if using an execution model translator such as Ocelot [14], it is possible to execute fused kernels on both the CPU and GPU to fully utilize the available computation power. This is the subject of ongoing research.

IV. Kernel Fission

Distinct from kernel fusion, kernel fission partitions a kernel into segments whose execution can be scheduled to hide PCIe transfer time. While such optimizations are well known, here we apply this transformation to CUDA kernels. The implementation described here uses CUDA Streams, a feature provided by NVIDIA CUDA [15]. A CUDA stream represents a queue interface to the GPU device. Multiple streams can be established to a device and CUDA commands (PCIe transfer, CUDA kernel) in the same CUDA Stream are executed in order, while those in different streams can run concurrently relative to each other. We built a software runtime manager on top of CUDA Streams referred to as the *Stream Pool* to aid kernel fission.

A. The Stream Pool

Currently programmers bear the burden of CUDA stream management, including creating and destroying the stream, arranging synchronization points between streams by calling the low level CUDA APIs (since the GPU does not have

API	Comment
getAvailabeStream()	get an available stream
setStreamCommand()	assign a command to a specific stream
startStreams()	start the execution
waitAll()	wait for the end of the execution
selectWait()	assign point-to-point synchronization between two specific streams
terminate()	end the execution immediately

Table IV: APIs provided by Stream Pool

an OS yet), and so on. Besides improving performance by reducing PCIe overhead, our Stream Pool is designed to abstract away such details of CUDA stream management and enhance programmer productivity.

The Stream Pool is implemented as a library and provides some straightforward high level APIs listed in Table IV. To use it, the programmer links the library during compilation and uses the API to assign commands to streams and set synchronization points without any knowledge of which CUDA Stream is actually used. The Stream Pool is implemented as a wrapper around the CUDA Stream APIs and associated data structures. The latter stores information about several CUDA streams each of which is tagged with attributes such as availability, lists of commands waiting to execute, and so on. Then the provided APIs will check or set these attributes to communicate with the CUDA Stream that is actually used. All examples in this section use this library.

However, concurrent execution on GPU is not always beneficial. For example, when executing two kernels concurrently on a device, each kernel nominally has access to only half the resources (CTAs and threads). Figure 12 uses the SELECT operator to illustrate this point. The line *no stream (old)* is the previous implementation of SELECT that passes 50% of the elements as shown in Figure 4(a). The line *no stream (new)* is the same implementation as *no stream (old)* but uses half the number of threads and CTAs. The performance of *(new)* is worse than *(old)*. The line *stream* uses Stream Pool to concurrently run two independent SELECTs each using the same design as *(new)* (the element number in the Figure is the total element number of both SELECTs). The performance of *(stream)* is better than *(new)* since two SELECTs can run concurrently. However, *stream* is worse than *(old)* when number of elements exceeds 8 million. It shows that concurrency is beneficial only when number of elements is small because less data parallelism exists. For large numbers of elements, concurrent stream execution is not advantageous. In cases like this , using more threads in a single kernel is better than using a smaller number of threads in concurrent kernels. Thus, the application of kernel fission must distinguish between such cases.

B. Pipelined Execution

With regards to fission, we can partition the set of CTAs in a kernel into segments and overlap the data transfer of one segment of CTAs with the execution of another

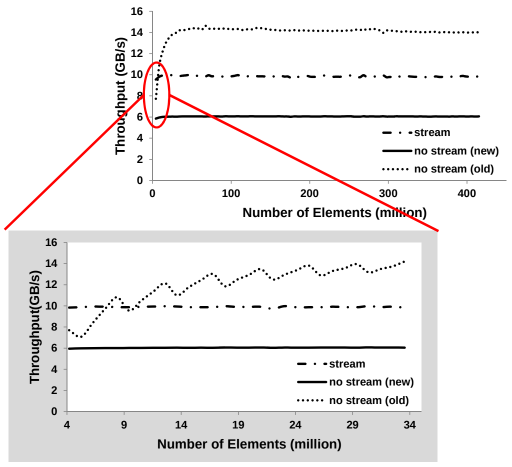

Figure 12: Performance of concurrently executing two selections

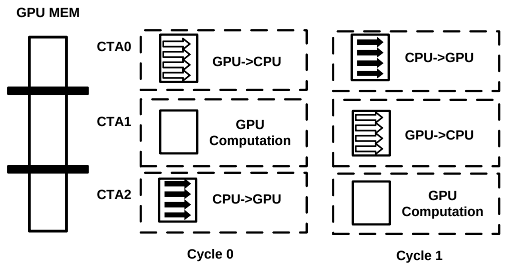

Figure 13: Example of kernel fission

segment of CTAs. Thus, the PCIe transfer is hidden by overlapping communication and computation. This is the simplest application of kernel fission, and is especially useful when the element number is large.

The GPU device used in this paper, the NVIDIA Tesla C2070, can overlap two PCIe transfers with a computation kernel which means the following three events can happen at the same time: one stream is downloading data to GPU, the other stream is computing and the third stream is uploading result to the CPU. For such a device, at least three streams are needed to fully utilize its concurrency capacity.

Figure 13 illustrates kernel fission with application to the SELECT operator. Consider the CTAs in the implementation of SELECT. In Cycle 0, CTA0 is transferring its result to the CPU, CTA1 is performing the computation, and CTA2 is loading input from the CPU. All three CTAs are running concurrently. After they finish their current tasks and next cycle begins when CTA0 loads new inputs, CTA1 transfers its new result to the CPU, and CTA2 starts computing on its newly received data. In this way, the PCIe transfer time is overlapped by the computation.

Figure 14 compares the performance of kernel fission with serial execution by using one SELECT operator. The data set used here is very large exceeding the size of GPU memory since these are the cases of interest. Note that our GPU's 6GB memory can hold less than 1.5 billion 32-bit integers. On average, the throughput achievable with

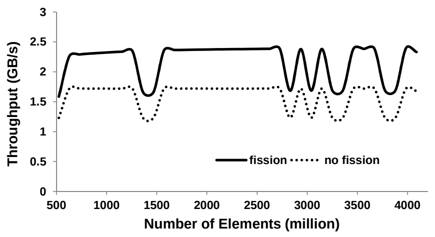

Figure 14: Performance of kernel fission

kernel fission is **36.9%** better than the baseline version which demonstrates that pipelining can provide significantly improved performance over simply concurrently executing the kernels (*stream* of Figure 12).

In principle, the execution time of kernel fission is the maximum of CPU→GPU transfer time, GPU compute time, and GPU→ CPU transfer time. Taking SELECT as an example, the maximum is typically the input transfer time because the result of SELECT is smaller than the input, and the operator itself is computationally simple. Thus, the performance of running one SELECT with kernel fission is relatively insensitive to the fraction of the operator taken by the filter operation. The drawback of kernel fission is that for performance reasons, one has to use pinned memory to transfer data which may hurt the CPU performance by reducing the available memory of CPU to perform other critical system tasks.

C. Combining Kernel Fusion and Fission

Kernel fusion and fission are orthogonal optimizations and can be used together when more than one RA operator is involved. Fission can be applied to a fused kernel or fusion can be applied to the kernels that result from fission. For example, consider the case of two back-to-back SELECT operators (Figure 15). When employing fusion followed by fission, CTAs performing computation (corresponding to fused kernels) are executing in parallel with CTAs performing data input and output. Compared with using fusion only, overlapped host transfers can further reduce the overall execution time. Finally we note that, since data is transferred to the CPU at different times, the CPU has to implement a gather stage the data at the end of the computation.

Figure 16 compares the performance of using four methods running two back-to-back SELECTs on a large volume of data. As expected, using both fusion and fission is on average **41.4%** better than serial, **31.3%** than fusion only, and **10.1%** than fission only.

V. Experimental Evaluation

In this section, we will evaluate kernel fusion and kernel fission with two real queries from TPC-H benchmark suites, Q1 and Q21. We first implement a baseline GPU version which does not use any optimization. Then, we manually apply kernel fusion and kernel fission to the baseline and

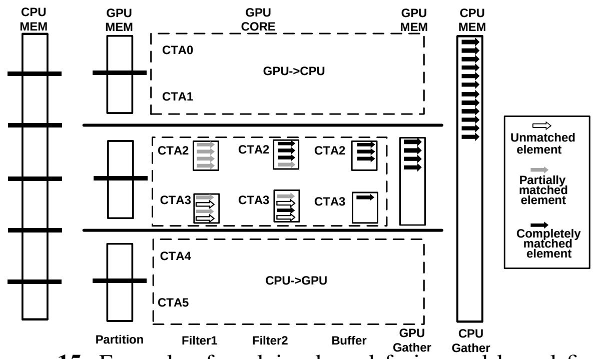

Figure 15: Example of applying kernel fusion and kernel fission together

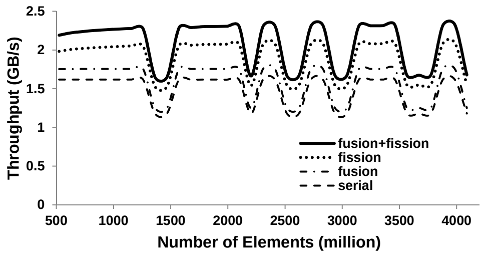

Figure 16: Performance Comparison of serial execution, kernel fusion, fission, fusion+fission

measure the speedup. The way we apply the fusion optimization follows the process discussed in the Section III-C. Thus the performance may be not as good as can be achieved if manually optimized more aggressively. However, we try to reflect what could be performed automatically by the compiler. This approach was also how all of the results in this paper were derived when evaluating fusion and fission transformations .

Q1, which calculates several price statistics of some selected entries, is one of the simplest queries in TPC-H. Figure 17(a) is the query plan generated for Q1. There are i) several JOINs and one SELECT to generate a large table from seven columns, ii) SORT by a different key, and iii) the arithmetic calculation over several fields of the table. The first part of the query including one SELECT and six JOINs can be fused into one kernel. All of the arithmetic computations performed as the final part of the query can be fused as well. Kernel fission can also be applied to the fused JOINS to hide CPU-GPU transfers. However, the fused arithmetic computations cannot use kernel fission because its input generated by the SORT operator is already located in the GPU memory. The SORT operator can neither be fused by kernel Fusion, nor be subject to kernel fission because it has to wait for the completion of the JOINs and arithmetic operations have to wait for the completion of the SORT.

Figure 18(a) compares the performance of applying kernel fusion only as well as applying both fusion and fission, against the baseline implementation. In the baseline, the most time consuming part is the SORT operator which takes around 71% of the total execution time, but cannot

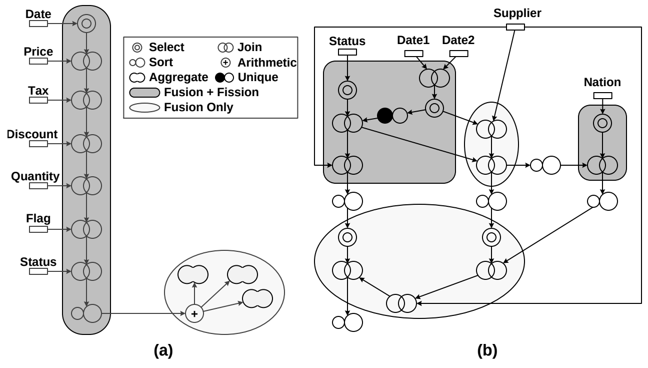

Figure 17: (a) Query plan for Q1; (b) Query plan for Q21

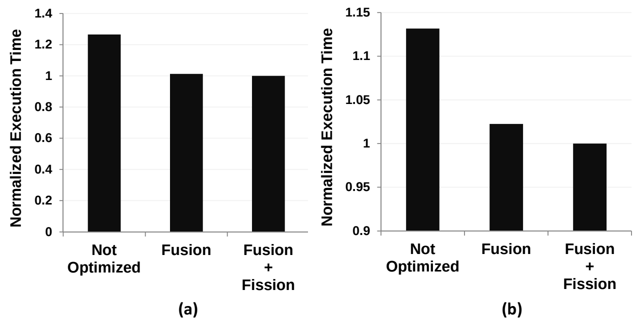

Figure 18: (a) Performance of Q1; (b) Performance of Q21 (both normalized to not optimized)

be optimized. However, the fusion dramatically speeds up the rest of the operators and contributes a **1.25x** speedup. In this case, kernel fission can bring another **1.01x** speedup because it hides the input transfer that occupies about 1% of the execution time of fusion. Thus, two optimizations combine bring **26.5%** performance improvement. Further study shows that SORT and PCIe data transfer are excluded, the remaining kernels that can be fused and kernel fusion can bring **3.18x** speedup from combining 6 JOINS and 1 SELECTs into a single kernel.

Query Q21 identifies certain suppliers who were not able to ship required parts in a timely manner. Compared with Q1, Q21 contains less arithmetic computation but has much more relational operations. Figure 17(b) is its simplified query plan (simple operators such as PROJECTs are omitted due to figure size limit). SORTs form a boundary for the application of kernel fusion since it can not be fused with any other operators. Figure 18(b) shows the performance result for Q21. Overall, kernel fusion and kernel fission together realize **13.2%** performance improvement which is not as much a performance improvement as achieved with Q1 mainly because of the number of kernels that are not fused. Only considering kernel fusion and operators that can be fused, this optimization can realize a **1.22x** speedup across that block of code (operators).

VI. Related Work

To our best knowledge, there is no generic or database specific GPU kernel fusion technique. However, kernel fusion is used as a domain specific optimization in some other areas. Copperhead, developed by Catanzaro et al. [16] fuses Python primitives to reduce global synchronizations when accelerating them by GPUs. Chakravarty et al. [17] proposes to use kernel fusion to accelerate Haskell array operations with GPUs. Coutts et al. [18] also fuse Haskell operators. Although their techniques only apply to their own domains, the experience and lessons learned from their projects such as the classification of kernel dependencies represent general principles valuable to our optimizations of RA operators.

On the CPU side, Lee et al. [19] proposes a runtime framework, Thread Tailor, which uses fusion and fission techniques albeit at a different level of granularity. Their framework partitions an application into a large number of threads and use greedy heuristic to combine these small threads later based on their dependences.

There are also several ongoing projects using GPUs to accelerate database applications. In particular, He et al. [2] implement a complete GPU database system, GDB, which is also based on the GPU implementation of relational algebra operators. Further, other groups focus on designing algorithms to accelerate individual operators [1], [20], [21], [22], [23], [24]. All of the preceding projects achieve several factors of speedup in comparison with their CPU counterparts. However, none of them use any optimizations to further improve the overall performance of the database system on GPUs. Moreover, He et al. also point out that the PCIe transfer time may outweigh the speedup brought by the GPUs and suggest the use of data compression techniques to reduce the amount of transfered data [25]. Our work differs in that we are seeking to discover and develop mainstream compiler passes that can automatically provide inter-kernel optimizations.

VII. Summary and Future Work

This paper proposes two inter-kernel optimization techniques for improving the performance of relational algebra primitives used in data warehousing applications on GPUs. We empirically evaluate the benefits of fusion on a combination of the SELECT operators as an example to demonstrate the potential benefits for relational operators in general. Through the implementation of the Stream Pool library we also study the impact of fission on SELECT operators. Finally, we evaluate the impact of concurrently applying fusion and fission on queries Q1 and Q21 from the TPC-H benchmark suite. All measurements are based on CUDA implementations and fusion and fission are performed manually reflecting potential compiler implementation (i.e., the fused kernels are not manually further optimized). We believe the presented data with the SELECT operator is encouraging and reflects the gains possible when applied to all operators. We point out that the result with applying fusion and fission to the two TPC-H queries supports this optimism. Our current efforts are focused on automation of these optimizations in the compiler.

Acknowledgement

This research was supported by the National Science Foundation under grants IIP-1032032 & CCF 0905459, by LogicBlox Corporation, and by equipment grants from NVIDIA Corporation.

References

[1] P. Trancoso, D. Othonos, and A. Artemiou, "Data parallel acceleration of decision support queries using cell/be and gpus," in *Proceedings of the 6th ACM conference on Computing frontiers*. ACM, 2009, pp. 117–126.

[2] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 21, 2009.

[3] J. Anderson, C. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[4] J. Mosegaard and T. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of Eurographics Workshop on Virtual Environments*, vol. 11, 2005, pp. 105–111.

[5] V. Podlozhnyuk, "Black-scholes option pricing," *Part of CUDA SDK documentation*, 2007.

[6] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics*, vol. 29, pp. 66:1–66:13, July 2010.

[7] E. Walker, "Benchmarking amazon ec2 for high-performance scientific computing," *Usenix Login*, vol. 33, no. 5, pp. 18–23, 2008.

[8] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. ACM, 2009, pp. 34–42.

[9] T. Council, "Tpc benchmark h, standard specification revision 1.3. 0," 1999.

[10] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili, "Efficient relational algebra algorithms and data structures for gpu," CERCS, Georgia Institute of Technology, Tech. Rep. GIT-CERCS-12-01, Feb. 2012.

[11] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[12] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 134–144.

[13] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[14] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proceedings of PACT '10*. ACM, 2010, pp. 353–364.

[15] *CUDA C Programming Guide*, Nvidia, May 2011, http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[16] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941562

[17] M. Chakravarty, G. Keller, S. Lee, T. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. ACM, 2011, pp. 3–14.

[18] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: From lists to streams to nothing at all," in *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. ACM, 2007, pp. 315–326.

[19] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread Tailor : Dynamically Weaving Threads Together for Efficient , Adaptive Parallel Applications," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, 2010.

[20] T. Lauer, A. Datta, Z. Khadikov, and C. Anselm, "Exploring graphics processing units as parallel coprocessors for online aggregation," in *Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*. ACM, 2010, pp. 77–84.

[21] P. Volk, D. Habich, and W. Lehner, "GPU-based speculative query processing for database operations," in *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[22] M. Lieberman, J. Sankaranarayanan, and H. Samet, "A fast similarity join algorithm using graphics processing units," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 2008, pp. 1111–1120.

[23] 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*. ACM, 2006, pp. 325–336.

[24] N. 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*. ACM, 2004, pp. 215–226.

[25] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 670–680, 2010.