The 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012

Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation

Haicheng Wu
Georgia Institute of Technology
hwu36@gatech.edu

Gregory Diamos
NVIDIA Research
gdiamos@nvidia.com

Srihari Cadambi
NEC Laboratories America
cadambi@nec-labs.com

Sudhakar Yalamanchili
Georgia Institute of Technology
sudha@ece.gatech.edu

Abstract

Data warehousing applications represent an emerging application arena that requires the processing of relational queries and computations over massive amounts of data. Modern general purpose GPUs are high bandwidth 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 data movement optimizations to address these challenges.

Inspired in part by loop fusion optimizations in the scientific computing community, we propose kernel fusion as a basis for data movement optimizations. Kernel fusion fuses the code bodies of two GPU kernels to i) reduce data footprint to cut down data movement throughout GPU and CPU memory hierarchy, and ii) enlarge compiler optimization scope. We classify producer consumer dependences between compute kernels into three types, i) fine-grained thread-to-thread dependences, ii) medium-grained thread block dependences, and iii) coarse-grained kernel dependences. Based on this classification, we propose a compiler framework, Kernel Weaver, that can automatically fuse relational algebra operators thereby eliminating redundant data movement.

The experiments on NVIDIA Fermi platforms demonstrate that kernel fusion achieves 2.89x speedup in GPU computation and a 2.35x speedup in PCIe transfer time on average across the micro-benchmarks tested. We present key insights, lessons learned, measurements from our compiler implementation, and opportunities for further improvements.

1. Introduction

The arrival of big data [20] has energized the search of architectural and systems solutions to sift through massive volumes of data. 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 [36, 18]. This expectation is motivated by the fact that GPUs have demonstrated significant performance improvements for data intensive scientific applications such as molecular dynamics [2], physical simulations in science [32], options pricing in finance [34], and ray tracing in graphics [33]. It is also reflected in the emergence of accelerated cloud infrastructures for the Enterprise such as Amazon's EC-2 with GPU instances [40].

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 [14] and there are fundamental differences between such applications and compute-intensive high performance computing (HPC) applications. Relational algebra (RA) queries form substantial components of data warehousing applications and on the surface appear to exhibit significant data parallelism. Unfortunately, this parallelism is generally more unstructured and irregular than other domain specific operations, such as those common to dense linear algebra, complicating the design of efficient parallel implementations. RA operators also exhibit low operator density (operations per byte) making them very sensitive to and limited by the memory hierarchy and costs of data movement.

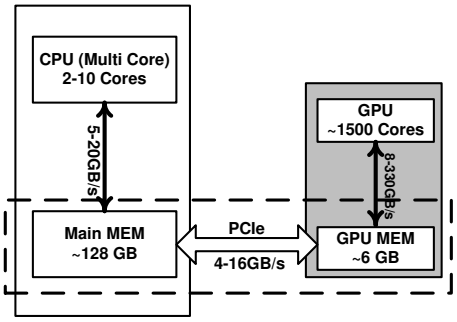

Figure 1: Memory hierarchy bottlenecks for GPU accelerators.

Overall, the nature and structure of RA queries make different demands on i) traversals through the memory hierarchy, ii) choice of logical and arithmetic operators, iii) control flow structure, and iv) data layouts. Consequently, there arise two fundamental issues. First there is a need for the efficient GPU implementations of RA primitives. Second, an issue 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 (the off-chip global memory) 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 although GPU can bring 2-27x speedup compared with CPU if only considering computation time, 15-90% of the total execution time is spent on moving data between CPU and GPU when accelerating database applications [18]. 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.

This paper addresses the challenge of optimizing data movement through the CPU-GPU memory hierarchy in the context of data warehousing applications (and hence their dominant primitives). Specifically, we propose and demonstrate the utility of **Kernel Weaver** as a framework for optimizing data movement. Kernel Weaver applies a cross-kernel optimization, *kernel fusion*, to GPU kernels. *Kernel fusion* is analogous to traditional loop fusion and its principal benefits are that it i) reduces transfers of intermediate data through the CPU-GPU memory hierarchy, ii) reduces the overall memory data footprint of a sequence of kernels in each level of the memory hierarchy, and iii) increases the textual scope, and hence benefits, of many existing compiler optimizations.

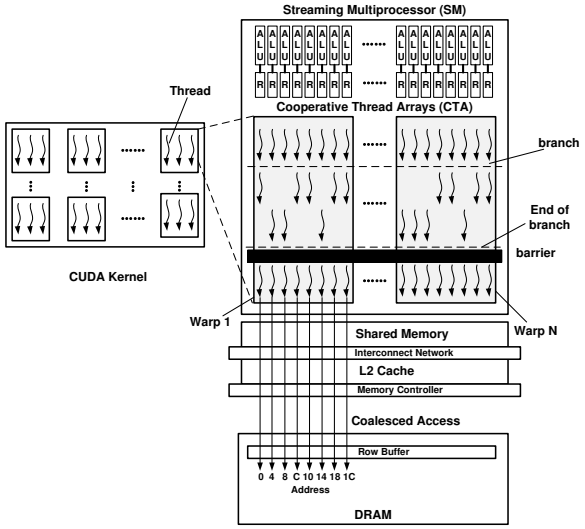

Figure 2: NVIDIA C2050 architecture and execution model.

This paper proposes the Kernel Weaver optimization framework and demonstrates the impact of kernel fusion for optimizing data movement in patterns of interacting operators found in the TPC-H benchmark suite. The goal of this paper is to provide insight into how, why, and when kernel fusion works with quantitative measurements from implementations targeted to NVDIA GPUs. This paper makes the following specific contributions:

- Introduction of the Kernel Weaver framework and algorithms for automated kernel fusion;
- Definition of basic dependences and general criteria for kernel fusion applicable across multiple application domains;
- Quantification of the impact of kernel fusion on different levels of the CPU-GPU memory hierarchy for a range of RA operators;
- Proposes and demonstrates the utility of compile-time data movement optimizations based on kernel fusion.

2. Background and Motivation

2.1. Programmable GPU

This paper uses NVIDIA devices and CUDA as the target platform. Figure 2 shows an abstraction of NVIDIA's GPU architecture and execution model. A CUDA program is composed of a series of multi-threaded kernels. Kernels are composed of a grid of parallel work-units called *Cooperative Thread Arrays* (CTAs) [37], that are mapped to Single Instruction Multiple Thread (SIMT) units called stream multiprocessors (SMs) where each thread has support for independent control flow. Different CTAs can execute in arbitrary order and synchronization between threads only exists within a CTA. Global memory is used to buffer data between CUDA kernels as well as to communicate between the CPU and GPU. Each SM has a shared scratch-pad memory with allocations for each CTA and can be used as a software controlled cache. Registers are privately owned by each thread to store immediately used values. CTAs execute in SIMD chunks called warps; hardware warp and thread scheduling hide memory and pipeline latencies. Effective utilization of the memory subsystem is also critical to achieving good performance.

CUDA and OpenCL are the dominant programming models in GPU computation. CUDA is dedicated to NVIDIA devices, and OpenCL is supported by NVIDIA, AMD and Intel GPUs. Terms used to describe GPU abstractions such as data parallel threads and shared scratch-pad memory typically vary depending on the specific programming model being considered. CUDA typically uses the terms *thread* and *shared memory*, and OpenCL typically uses *work item* and *local memory*. The CUDA terminology is adopted in this paper because Kernel Weaver is currently implemented based on it. However, the same concept and technology can also be applied to OpenCL and its supported devices.

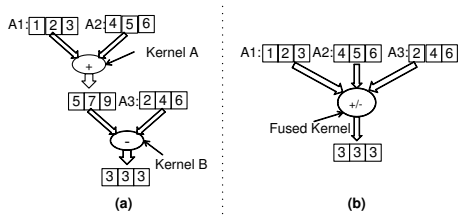

Figure 3: Example of kernel fusion.

2.2. 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 [1]. They are the building blocks of modern relational database systems. A relation is a set of tuples, each of which comprises of a list of n attributes. Some attributes are keys that are considered by the RA operator.

Table 1 lists the common RA operators and a few simple examples. In general, these operators perform simple tasks on a large amount of data. A typical data warehousing query consists of dozens of RA operators over massive data sets.

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. Further, operators such as SORT and UNIQUE are required to maintain certain order amongst data elements and thereby can introduce certain ordering constraints amongst relations.

2.3. Motivation

The idea of GPU kernel fusion comes from classic loop fusion optimization. Basically, kernel fusion reduces data flow between two kernels (via the memory system) by merging them into one larger kernel. Therefore, its benefits goes far beyond reduction in PCIe traffic. Figure 3 depicts an example of kernel fusion. Figure 3(a) shows two dependent kernels - one for addition and one for subtraction. After fusion, a single functionally equivalent new kernel (Figure 3(b)) is created. The new kernel directly reads in three inputs and produces the same result without generating any intermediate data.

Kernel Fusion has six benefits as listed below. The first four stem from creating a smaller data footprint through fusion since it is unnecessary to store temporary intermediate data in global memory after each kernel execution, while the other two relate to increasing the compiler's optimization scope.

Smaller Data Footprint results in the following benefits:

- Reduction in Memory Accesses: Fusing data dependent (producer-consumer) kernels enables storage of intermediate data in registers or GPU shared memory (or cache) instead of global memory.
- Temporal Data Locality: As in traditional loop fusion, access to common data structures across kernels expose and increase temporal data locality. For example, fusion can reduce array traversal overhead when the array is accessed in both kernels.

RA Operator	Description	Example
SET UNION	A binary operator that consumed two relations to produce a new relation consisting of tuples with keys that are present in at least one of the input relations.	x = {(2,b),(3,a),(4,a)}, y = {(0,a),(2,b)} UNION x y → {(0,a),(2,b),(3,a),(4,a)}
SET INTERSECTION	A binary operator that consumes two relations to produce a new relation consisting of tuples with keys that are present in both of the input relations.	x = {(2,b),(3,a),(4,a)}, y = {(0,a),(2,b)} INTERSECT x y → {(2,b)}
SET DIFFERENCE	A binary operator that consumes two relations to produce a new relation of tuples with keys that exist in one input relation and do not exist in the other input relation.	x = {(2,b),(3,a),(4,a)}, y = {(3,a),(4,a)} DIFFERENCE x y → {(2,b)}
CROSS PRODUCT	A binary operator that combines the attribute spaces of two relations to produce a new relation with tuples forming the set of all possible ordered sequences of attribute values from the input relations	x = {(3,a),(4,a)}, y = {(True)} PRODUCT x y → {(3,a,True),(4,a,True)}
JOIN	A binary operator that intersects on the key attribute and cross product of value attributes.	x = {(2,b),(3,a),(4,a)}, y = {(2,f),(3,c),(3,d)} JOIN x y → {(2,b,f),(3,a,c),(3,a,d)}
PROJECT	A unary operator that consumes one input relation to produce a new output relation. The output relation is formed from tuples of the input relation after removing a specific set of attributes.	x = {(2,False,b),(3,True,a),(4,True,a)} PROJECT [0,2] x → {(2,b),(3,a),(4,a)}
SELECT	A unary operator that consumes one input relation to produce a new output relation that consists of the set of tuples that satisfy a predicate equation. This equation is specified as a series of comparison operations on tuple attributes.	x = {(2,False,b),(3,True,a),(4,True,a)} SELECT (key==2) x → {(2,False,b)}

Table 1: The set of relational algebra operations. In the example, the 1st attribute is the "key". (Syntax: (x,y) – tuple of attributes; {(x1,y1),(x2,y2)} – relation; [0,2] – attribute index)

- Reduction in PCIe Traffic: Kernel fusion can cut down transfers of inter-kernel data across the PCIe interconnect.
- Larger Input Data: Since kernel fusion reduces intermediate data thereby freeing GPU memory, larger data sets can be processed on the GPU increasing throughput.

Larger Optimization Scope brings two benefits:

- Common Computation Elimination: When two kernels are fused, the common stages of computations are redundant and can be avoided.
- Improved Compiler Optimization Benefits: When two kernels are fused, the textual scope of many compiler optimizations are increased bringing greater benefits than when applied to each kernel individually.

These benefits are especially useful for data warehousing applications since RA operators are fine grained and exhibit low operation density, ops per byte transferred from memory. Fusion naturally improves operator density and hence performance. Figure 4 is a simple example comparing the GPU computation throughput of back-to-back SELECTs with and without kernel fusion. Inputs are randomly generated 32-bit integers, the x-axis is the problem size which fits GPU memory, and kernels were manually fused in this example. On average, fusing two SELECTs achieves **1.80x** larger throughput while fusing three kernels achieves **2.35x**. Fusing three SELECTs is better since more redundant data movement is avoided and larger code bodies are created for optimization.

Recently, Intel introduced the Sandy (and Ivy) Bridge architectures and AMD brought Fusion APUs to the market. Both designs put the CPU and GPU on the same die and remove the PCIe bus. In these systems, four out of the six benefits listed above still apply (excluding Reduction in PCIe Traffic and Larger Input Data). Thus, kernel fusion is still valuable.

While a programmer could perform a fusion transformation manually, database queries are typically supplied in a high level language like Datalog or SQL, from which lower-level operations are synthesized using a query planner and compiler. Automating this process

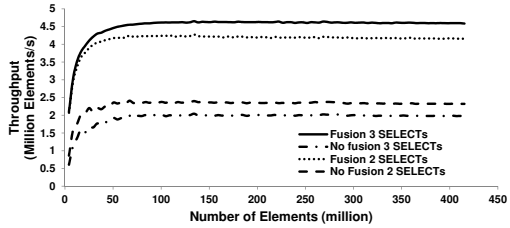

Figure 4: Performance Comparison between fused and independent SELECTs.

as a compilation transformation is necessary to make GPUs accessible and useful to the broader community of business analysts and database experts. Moreover, running kernel fusion dynamically in a just-in-time compiler (JIT) creates opportunities to leverage runtime information for more effective optimizations.

3. System Overview

Kernel Weaver is implemented as part of a domain-specific compilation and run-time system illustrated in Figure 5. The language front-end is based on the Datalog language [21]. Datalog is a declarative language used to express database queries and operations - in this case over large data sets. The output of the language front-end is a query plan that contains nodes corresponding to arithmetic, logical, and relational operators and their dependences. These are translated into an internal kernel intermediate representation which drives the Kernel Weaver transformation engine. Kernel Weaver operates on CUDA source implementations of RA operators stored in a primitive library to produce fused CUDA implementations from which *nvcc* is used to generate kernel code in NVIDIA's parallel thread execution (PTX) instruction set. The lightweight host runtime layer [10] picks up the fused PTX kernels and drives the Ocelot dynamic compilation and runtime infrastructure [11] which is responsible for the execution on the NVIDIA GPUs. Note that each RA operator may be implemented as several CUDA kernels so that fusing operators requires coordinated fusion of several CUDA kernels. While we tested all of

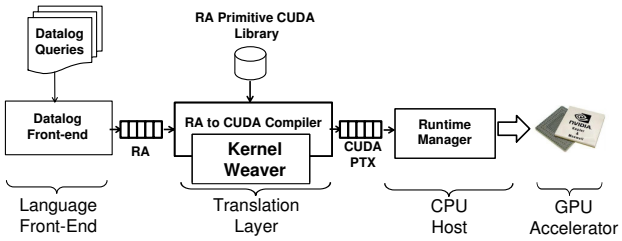

Figure 5: System diagram of Kernel Weaver

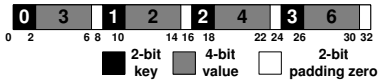

Figure 6: An example of a relation with four tuples, each compressed into 8-bits and packed into a single 32-bit word.

the examples in this paper, the language front-end needs further work before the full Datalog language can be executed on GPUs.

3.1. Kernel Representation

Kernel fusion is based on the multi-stage formulation of algorithms for the RA operators. Multi-stage algorithms are common to sorting [31], pattern matching [39], algebraic multi-grid solvers [5], or compression [17]. This formulation is popular for GPU algorithms in particular since it enables one to separate the structured components of the algorithm from the irregular or unstructured components. This can lead to good scaling and performance. Kernel fusion can now be explained as a process of weaving (mixing, fusing, and reorganizing) stages from different operators to generate new optimized operator implementations.

The high level description of the order and functionality of the stages will be referred as an *algorithm skeleton*. In this paper we use the algorithm skeletons developed by Diamos et al. [12] which have been evaluated to be 1.69-3.54x faster than those developed by He et al. [18]. Diamos et al. store relations as a densely packed array of tuples with strict weak-ordering. Figure 6 is an example of a 32-bit relation containing 4 tuples sorted according to the key attributes. The sorted form allows for efficient array partitioning and tuple lookup operations. In our compilation environment, skeletons for all of the RA operators are stored in the RA primitives library (see Figure 5) with CUDA implementations of each stage. The remainder of this section describes the basic structure, relevant details, and adaptations we have made of their implementation.

All RA operator skeletons are comprised of three major stages, partition, compute and gather. In the following we briefly describe the functionality of each stage using the implementation of a simple operator - SELECT (Figure 7) - as an example.

Partition: The input relations are partitioned into independent sections that are processed in parallel by different CTAs. For unary operators such as SELECT in Figure 7 the input relations can be evenly partitioned to balance the workload across CTAs. Binary operators such as JOIN and SET INTERSECTION are more complex in this stage since they need to partition both inputs and partitioning is based on a key value consequently producing unbalanced sizes of inputs to CTAs and resulting in unbalanced compute loads.

Compute: A function for each RA operator is applied to its partition of the inputs to generate independent results. Different RA operators are specialized to effectively utilize fine-grained data parallelism and the multi-level memory hierarchy of the GPU to maximize

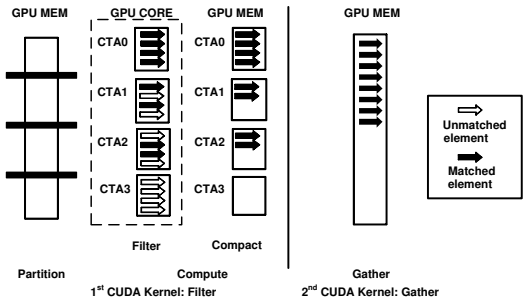

Figure 7: Example algorithm of SELECT

performance. For example, the SELECT in Figure 7 first *filters* every element in parallel and then leverages the shared memory to *compact* [6] the filtered result in preparation to create a contiguous output.

Gather: The results computed in individual partitions are gathered into a global dense sorted array of tuples by using a coalesced memory to memory copy, a common CUDA programming pattern [30].

Multi-stage RA operators are implemented as multiple CUDA kernels - typically one per stage. Kernel weaver fuses operators by interleaving stages and then fusing interleaved stages (their respective CUDA implementations) to produce a multi-stage implementation of the fused operator. Thus, fusion of CUDA kernels is necessary to realize operator fusion. A variety of alternative implementations can be used for the implementation of each stage and can be accommodated by the operator fusion process. Diamos et. al. [12], report performance results that are significantly better than any reported results in the literature. Consequently we use their multi-stage algorithms for RA operators in the demonstration of kernel weaver.

4. Automating Fusion

This section introduces the process of kernel fusion employed in Kernel Weaver. For simplicity, the initial description is based on each operator being implemented as a single data parallel kernel. Subsequently, we will describe higher performance multi-stage implementations of the RA operators. This section describes three main steps to fuse operators: (i) using compiler analysis to find all groups of operators that can be fused, (ii) selecting candidates to fuse, and (iii) performing fusion and generating code for the fused operators.

4.1. Criteria for Kernel Fusion

The simple idea is to take two kernels say with 4096 threads each, and produce a single kernel with 4096 threads, where each thread is the result of fusing two corresponding threads in the individual kernels. Clearly, the data flow between the two fused threads must be correctly preserved. The classification below can be understood from the perspective of preserving this simple model of kernel fusion. The first consideration is finding feasible combinations of data parallel kernels to fuse via compiler analysis, followed by the selection of the best options. Two types of criteria for fusion of candidate kernels are that they possess i) same kernel configuration (CTA dimensions and thread dimensions), and ii) producer-consumer dependence.

The first criteria is similar to loop fusion [23] that requires compatible loop headers (same iteration number, may need loop peeling to pre-transform the loop, etc.). Kernel fusion also requires compatibility between kernel parameters. The fused kernel will have the same kernel configuration as the candidates. The data parallel nature of RA operators make their implementation independent (with respect to correctness) of the kernel configuration. Thus, while too many

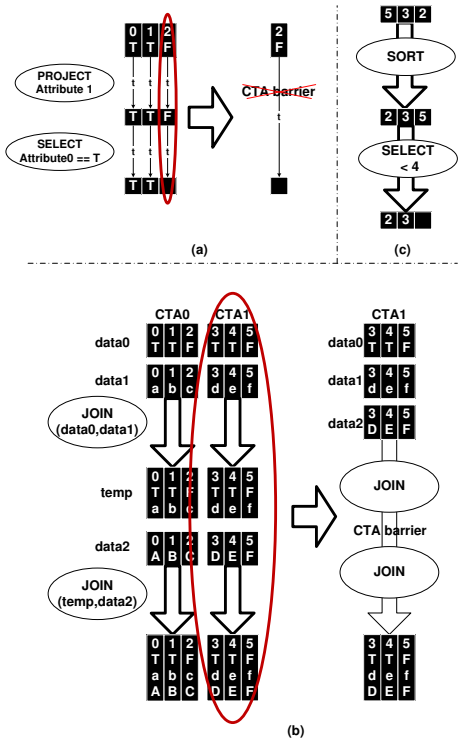

Figure 8: Example of three kinds of dependence: (a) thread dependence; (b) CTA dependence; (c) kernel dependence.

or too few CTAs or threads may lead to inefficient use of resources, fusion can be performed correctly if the kernel configurations are the same. This work tests a set of micro-benchmarks (see Section 5) with a wide range of combination of CTA dimensions and thread dimensions and picks one pair that works best in most cases.

The second criteria is due to the fact that the benefits listed in Section 2.3 are derived primarily from exploiting producer-consumer dependences. Data dependence analysis is necessary to find candidate kernels. Producer-consumer dependence between two data parallel kernels can be classified into three categories as shown in Figure 8: thread, CTA and kernel dependence.

In the first category, each thread of the consumer kernel only consumes data generated by a single thread from the producer kernel. Figure 8(a) illustrates such an example with tuples containing two attributes, e.g., (1,T). Dependences between producer and consumer kernels corresponding to unary RA operators such as SELECT and PROJECT, belong to this category because the operation on one input tuple is independent of the operation performed on any neighboring tuple. In this case, corresponding producer and consumer threads from each kernel can be fused without having to insert synchronization operations. This type of producer-consumer dependence between kernels is referred to as *thread dependence*.

The second category is wherein every CTA of the consumer kernel depends on the completion of a CTA of the producer kernel. Such dependences are referred to as *CTA dependences*. For example, this occurs between binary RA operators such as JOIN and SET INTERSECT that have a producer-consumer dependence. Consider, Figure 8(b) that illustrates a producer-consumer dependence between two JOIN operators. The first operator performs a JOIN operation across tuples from two input data sets, *data0*, and *data1*. Each CTA is provided a partition of input tuples, corresponding to some range of the key value used in the JOIN (in this example each tuple has a

Input: a list of operators *op*
Output: a list of fusion candidate groups *c*
i = 0;
length = size of list *op*;
Topologically sort *op*;
while *i* ≠ *length* **do**
    class = classify dependence between *op*[*i*] and its predecessors and successors;
    if *class* == *Kernel Dependence* **then**
        delete *op*[*i*];
    end
    i = *i* + 1;
end
c = all connected subgraphs of *op*;

Algorithm 1: Searching for fusing candidates.

unique key value which is an integer). Thus, a thread in a CTA must compare the key values of tuples it is processing with the key values of tuples being processed by every other thread in the CTA, *and only within the CTA*. While such a partitioning of input tuples across CTAs produces unbalanced loads between CTAs, data dependences between threads are confined to remain within a CTA. The producer CTA writes its tuples to shared memory where a CTA from the consumer kernel can now pick it up. A barrier synchronization is necessary after the first JOIN operation before the second JOIN operation can start. The actual implementation is more involved, but for the purposes of this paper, corresponding threads from producer-consumer CTAs can be fused with appropriately placed barriers.

The third category is wherein the consumer kernel has to wait until the completion of all threads in the kernel, i.e., kernel fusion is not feasible. A typical example is where the producer kernel is a SORT operator (Figure 8(c)) because it behaves like a global barrier. The reasons are i) it cannot be launched until all inputs arrive; ii) SORT shuffles all data and the following consumer operators need to wait for its completion before being able to start streaming data. Such dependence is referred to as *kernel dependence*.

Note the three categories of dependences are from the perspective of being able to fuse kernels by fusing corresponding threads within producer-consumer kernels. Accordingly, the dependences are implicitly associated with the level of the memory hierarchy used to pass data. Fused threads across thread dependent kernels use the register file which is allocated by the thread. Fused threads across CTA dependent kernels use shared memory which is allocated by the CTA. Finally, according to the above classification, the kernels in an dependence graph that are candidates for kernel fusion only exhibit thread or CTA dependences with other kernels, and are bounded by operators with kernel dependences. Algorithm 1 formalizes the steps to find kernel fusion candidates. Its main idea is first removing operators causing kernel dependence from the graph and then finding the rest connected operators.

The output of the language front-end consists of RA operators and their associated variables. This information is used to construct an RA dependence graph like the one shown in Figure 9(b). The nodes in the graph represent RA operators and the directional edges identify nodes with the producer-consumer dependences. The large circle bounded by SORT operators contains operators satisfying the dependence requirement and are candidates for fusion. Instances supporting recursive queries (e.g. ancestor(a,c)←parent(a,b),ancestor(b,c)) may generate a dependence graph with enclosed loops. This work only

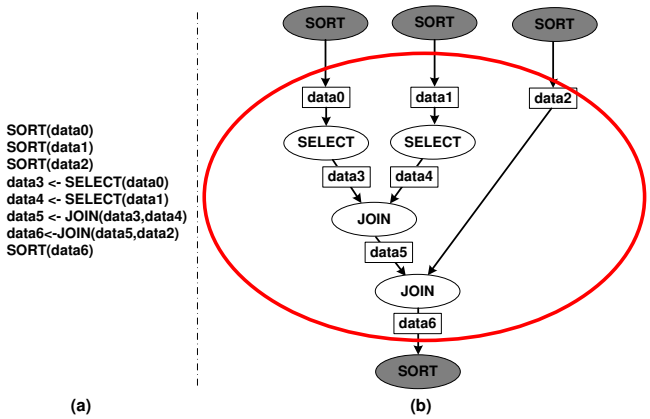

Figure 9: Example of constructing dependence graph: (a) database program; (b) dependence graph.

Input: a list of candidate operators op
Input: resource budget b
Output: a list of fusion groups f
$i = 0$;
$j = 0$;
$length$ = size of list op;
Topologically sort op;
while $i \neq length$ **do**
  add $op[i]$ to $f[j]$;
  $cost$ = resource usage estimation of $f[j]$;
  if $cost > b$ **then**
    delete $op[i]$ from $f[j]$;
    $j = j+1$;
  else
    $i = i + 1$;
  end
end

Algorithm 2: Choosing operators to fuse.

considers acyclic graphs although often loop unrolling and related known optimizations can create acyclic dependence graphs.

4.2. Choosing Operators to Fuse

Fusing all the kernels meeting above criteria may not be practical. The main constraint on fusion is resource constraints - pressure on limited registers and limited amount of shared memory available within each stream multiprocessor. Fusion choices must also be ordered based on dependences and performance impact. Accordingly we adopt the following heuristic and use Algorithm 2 to choose operators to fuse.

Figure 10 is an example that shows how the greedy heuristic of Algorithm 2 works. It starts from the candidates circled in Figure 9(b). Figure 10(a) first performs a topological sorting on the dependence graph to produce a list of operators. If operators execute in this order, all dependences will be honored. Starting from the top of the list, Figure 10(b) searches for the longest contiguous sequence of operators that can be fused, within resource constraints, i.e., fits within the shared memory and registers budgeted for each CTA (*data3* and *data4* become internal to the fused operator). In the example in Figure 10(c) the second JOIN cannot be added since the estimated shared memory resource usage is larger than the budget. The algorithm repeats the above process for the next not fused operator, the second JOIN, until no more operators can be fused. Resource

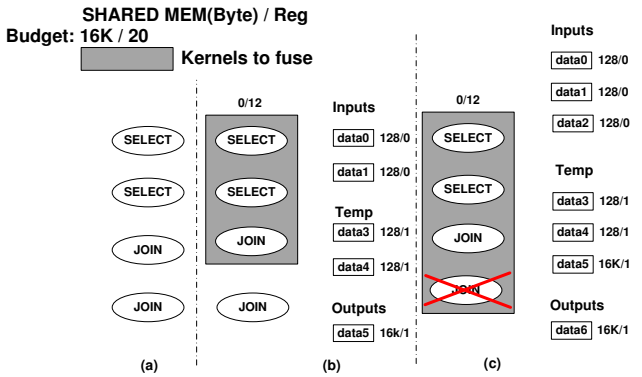

Figure 10: Example of choosing operators to fuse: (a) topologically sorted operators; (b) choose the first three operators to fuse; (c) refuse to fuse the fourth operator.

usage estimation is discussed in Section 4.3.3 after introducing code generation.

The intuition underlying the above method is that it is more important to fuse operators executed earlier than those executed later. The reason is that data warehousing applications normally process large amounts of data. After several filtering and reduction operators, the data set is reduced significantly. Resource permitting, fusing the first few operators in the dependency graph provides the most benefit.

4.3. Kernel Weaving and Fusion

Given the dependence graph and candidate operators to fuse, the final step is performing the fusion. Recall that each operator is implemented as a multi-stage algorithm with three stages - *partition*, *compute*, and *gather* - each of which is implemented as a CUDA data parallel kernel. The fused operator still has these three stages. At a high level, fusion is achieved by two main steps: (i) grouping the *partition*, *compute*, and *gather* stages of the operators together (which we also refer to as interleaving); (ii) fusing the individual stages. In other words, the *partition* stages of the candidate operators are fused together into a single data parallel kernel, which could be viewed as the *partition* stage for the newly fused operator. Similarly, the *compute* and *gather* stages are fused into a single fused *compute* and *gather* stage respectively. For example, when two operators are fused, the fused operator will have the multi-stage structure shown in Figure 11 where the two compute stages are fused into one data parallel kernel (the fused partition and fused gather stages similarly represent fusion of individual partition and gather stages). The fused computation stage performs the computation stage of the original operators in the order of their dependences. All intermediate data and data sizes are stored in the shared memory or registers. The fused operator may have multiple inputs and outputs.

The above fusion process includes code generation for the fused operators. Code generation takes as input a description of a topologically sorted set of operators to be fused and their associated variables, and produces CUDA code for the data parallel kernels that implement the fused operator. The CUDA code is generated by concatenating the instantiated algorithm skeleton code of each stage, and connecting the outputs of one stage to the inputs of the next stage. How to connect stages is discussed in the following sub sections. A variable table, which records and tracks the use of variables between stages, is needed to instantiate the skeleton. Figure 11 shows how the variable table tracks the variables that hold result data and result size of each computation stage.

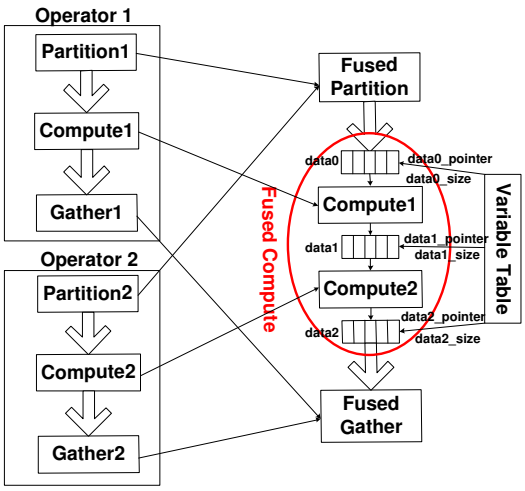

Figure 11: The structure of generated code (fusing two operators).

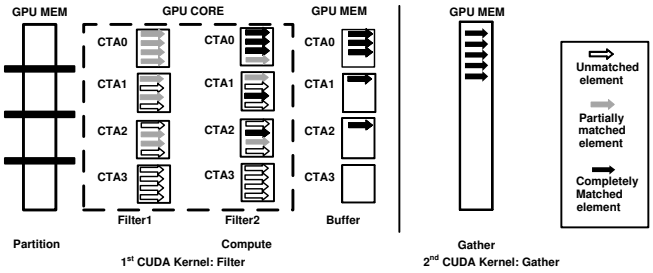

Figure 12: Example of fusing two SELECTs.

Fusing operators depends on whether thread dependence or CTA dependence exists between operators. We now describe in more detail how to fuse operators with thread and CTA dependences.

4.3.1. Fusing Thread Dependent Only Operators Unary operators SELECT and PROJECT exhibit thread dependence. The kernel configuration (threads/CTA and CTA grid dimensions) of both operators are equal. Therefore each thread in the producer operator is fused with a corresponding thread in the consumer operator.

The partition stage of the fused operator remains the same as that of the producer operator. The compute stage of the fused operator is a data parallel kernel with the same kernel configuration, where each thread is created as follows. Every thread first loads a tuple from its input partition into registers. The computation of corresponding producer and consumer threads are performed using these registers, i.e., SELECT or PROJECT in the correct order. These operators either discard data (SELECT) or discard attributes (PROJECT). The output of this sequence of operations is compacted into an output array. The gather stage accumulates all of the data from different threads in the fused compute stage into contiguous memory.

As shown in Figure 7, the computation stage of SELECT has two parts, filter and stream compaction. After kernel fusion, stream compaction is needed only when the SELECT result should be copied to GPU memory. Figure 12 is an example of fusing two back-to-back SELECTs together. Compared with Figure 7, only one filter operation is added. Moreover, the fused kernel only needs to read and write memory once rather than twice.

For PROJECT, its result tuple should be stored into a new register with a different data type since it contains less attributes. Thus, the operations after PROJECT have to use this new register instead.

4.3.2. Fusing CTA and Thread Dependent Operators Binary relational operators are CTA dependent . This change increases the

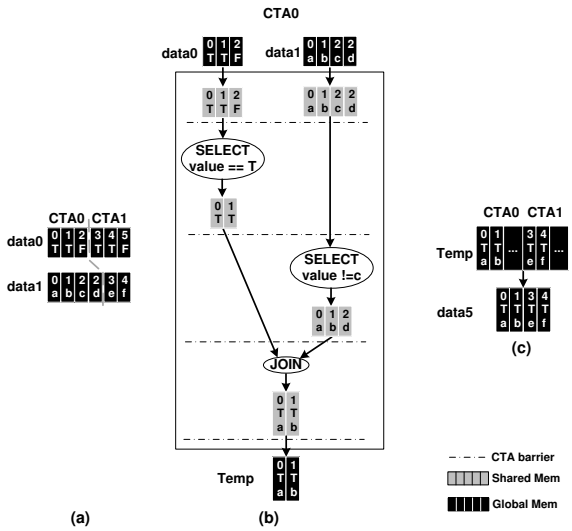

Figure 13: Example of Generated Code of Figure 10(b): (a) Partition two inputs; (b) Computation of one CTA; (c) Gather one output.

number of inputs of the fused operators and necessitates the following main distinction in code generation: (i) Use binary search to partition inputs; (ii) Use shared memory to support CTA dependence; (iii) Synchronize two operators having CTA dependence. Thus, code generation has to be extended to support the three differences. Figure 13 shows the generated code for the operators in Figure 9(b) and is used as example to explain the extensions.

Our approach is to maintain the independent operation of each CTA to be able to fuse corresponding CTAs from the producer and consumer operators. This is achieved in the *partition* stage by partitioning the input set by key values. Each CTA then receives a set of tuples corresponding to a specific range of key value pairs. This is achieved using binary search [3] and both inputs of each binary operator are partitioned across CTAs. For example, in Figure 13(a), *data0* is first evenly partitioned into two parts bounded by pivot tuples. Then, a binary search is used to lookup the tuples in *data1* corresponding to the *key* attributes of *data0* pivot tuples. The partitioned data sizes of the two inputs provided to each CTA thus may differ (e.g. *data1*). However, when fusing two binary operators (e.g., two JOIN operators), three inputs need to be partitioned and each operator may use different keys. For instance, one JOIN may use the first 2 attributes as a *key* and the other JOIN may only use the first attribute as *key*. In this case, the fused input stage will only use the first attribute as *key*. This preserves the independence of operation across CTAs.

Figure 13(b) is an example of the computation stage of one CTA. The other CTA works in exactly the same way but upon different data. In the beginning, each CTA first allocates a software controlled cache in shared memory for each input and then loads data into the cache (e.g. CTA0 loads in a portion of corresponding *data0* and *data1* divided as in Figure 13(a)) . Afterwards, a CTA-wise fused computation performs fused operations upon those cached data. Within a CTA, the generated code can perform all supported operations such as SELECT and JOIN. If two connected operators have CTA dependence (e.g. between SELECT and JOIN), the result data of the producer operator should be stored in a cache allocated in the shared memory, and the result size is stored in a register. To guarantee all threads within a CTA finish updating the cache, a CTA

barrier synchronization is needed after the producer operation. If two dependent operators only exhibit thread dependence, they only need to use register(s) to pass value(s) and no synchronization is necessary. For example, the first operator in Figure 13 to execute is SELECT and it has CTA dependence relationship with its consumer JOIN. Thus, SELECT has to store its result in shared memory rather than the register. The second SELECT is handled in the same way. Thus, the inputs of JOIN all reside in the shared memory before its execution. After JOIN, the result is dumped to GPU memory.

The gather stage (Figure 13(c)) is the same as in the thread dependent only cases which packs the useful results generated by two CTAs into an output array.

4.3.3. Resource Usage Code generation decides how many resource will be occupied. As shown in Figure 10(c), some resources are used to store input, output, and intermediate temporary data. Others are used inside the computation.

Fusing thread dependent operators stores intermediate data in the registers. The number of needed registers depends on the data type of the tuple which is provided by the database front-end. Fusing CTA dependent operators stores temporary data in the shared memory and temporary data size value in one register. Allocated shared memory size is a function of data type, input data size and operator type. For example, SET INTERSECT needs to allocate $\min(input1, input2)$ tuples for its output. The data variable and data size variable stored in registers are live until they are no longer needed.

Registers are also needed to perform partition, computation, and gather. The partition result, the beginning and the end position of all inputs, uses variables to pass to the computation stage. The liveness of the variables used inside each stage is the same as the scope of the stage. Thus, different variables of different stages can reuse the same registers. So, the register usage of a fused operator is not larger than the maximum of the register usage in each stage plus the registers used to pass values between stages. The registers used by each stage can be determined as long as the data types of all tuples are known.

4.4. Extensions

The preceding three sections discussed how code is generated for RA operators having producer-consumer dependence. This method can be extended to support other dependence or other operators.

The first extension is to support input dependence, i.e. operators shares the same inputs. The benefits of fusing these operators is that the input data shared by different operators only need to be loaded once, which is not as important as the case of producer-consumer dependence. Fusing operators having input dependence also increases the resource pressure. The modification to the above automation process is to detect input dependence when constructing the dependence graph. The code generation part can remain the same.

The second extension is to support simple arithmetic operations such as addition, subtraction, multiplication and division. These arithmetic operators are much simpler than RA operators. They have two inputs, but use even partitions to divide both inputs. The dependence between them belongs to thread dependence and can use registers to store computation results.

5. Experimental Evaluation

Table 2 shows our experimental infrastructure. We use TPC-H [9], a widely-used decision support benchmark suite, to quantify the speedups of kernel fusion in a practical situation. TPC-H comprises 22 queries with 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. Before showing results for actual TPC-H queries, we examine micro-benchmarks derived from the TPC-H queries.

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 2: Experimental Environment.

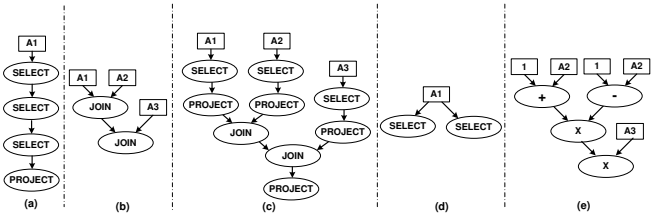

Figure 14: Common operator combinations to fuse.

5.1. Micro-benchmarks

We analyze TPC-H queries and identify commonly occurring combinations of operators that are potential candidates for fusion. From the 22 queries in TPC-H, Figure 14 illustrates some frequently occurring patterns of operators corresponding to different cases discussed in Section 4. In the figure, (a) is a sequence of back-to-back SELECT operators that perform filtering, for instance, of a date range. It only has thread dependence. (b) is a sequence of JOIN operations that creates a large table with multiple attributes, and exhibits CTA dependence. (c) corresponds to the JOIN of three small selected tables and has both thread and CTA dependence. (d) represents the case when different SELECT operators need to filter the same input data and has input dependence. (e) performs arithmetic computations such as $price \times (1 - discount) \times (1 + tax)$ which appears in several TPC-H queries. The PROJECTs in the figure discard their sources and only retain part of the result. The above patterns can be further combined to form larger patterns that can be fused. For example, (a) and (b) can be combined to form (c).

In the following experiments, the tuple used in patterns (a)–(d) are 16 bytes. (e) uses single precision floating point values.

5.1.1. Examples of Generated Code Figure 15 shows the generated fused computation stage code of Figure 14(a) (only two SELECTs shown for brevity). It performs two SELECTs and a PROJECT. The first two filters operate on the value in *data_reg*, and store the filter result in *match*, which is later used to determine if follow-up operations are needed. The result of PROJECT is written to a new register *project_reg* since its data type is smaller than *data_reg*. The last step, stream compaction, dumps the value stored in this new register to the GPU's global memory. The generated code may be less compact than manually written code, but compilers such as *nvcc* can optimize it to produce high quality binary code.

5.1.2. Small Inputs The micro-benchmarks listed in Figure 14 are first tested with small inputs that fit in GPU memory. The purpose of this is to isolate the benefits of kernel fusion from the effects of PCIe transfer. Figure 16 shows the speedup in the pure GPU execution time (no PCIe transfer) with kernel fusion. The input data are randomly

```
if(begin_input + id < end_input)
{
  data_reg = begin_input[id];                    Load Data To Reg
  {
    unsigned char key = extract(data_reg);
    if(comp(key, 64))
    match = true;
  }                                              Filter0
  {
    if(match)
    {
      unsigned char key =extract(data_reg);
      if(comp(key, 64))
      match = true;
    }
  }                                              Filter1
  {
    if(match)
    {
      project_reg = project(data_reg, 0);
    }
  }                                              PROJECT
}
{
  unsigned int max = 0;
  unsigned int output_id = exclusiveScan(match, max, 0);
  if(match)
    buffer0[output_id] = project_reg;            Stream
  __syncthreads();                               Compaction
  if(threadIdx.x < max)
    begin_output0[outputIndex0 + threadIdx.x] = buffer0[threadIdx.x];
  outputIndex0 += max;
}
```

Figure 15: Example of generated computation stage source code of Figure 14(a).

generated and then fed into the automatically generated fused code using the compilation flow of Figure 5. The baseline implementation for comparison directly uses the implementation from the primitive library without fusing. Similar to Figure 4, the performance data are averaged over a wide set of problem size (from 64 MB to 1 GB). On average, kernel fusion achieves a **2.89x** speedup. Cases (a) and (e) containing only thread dependence show the largest speedup, because they do not insert new synchronizations, and threads execute independently. Furthermore, (a) gets rid of three stream compaction stages and three gather stages after fusion. The speedup of case (d) is less than the rest because it has input dependences and can only benefit from loading fewer inputs. (b) and (c) have CTA dependences and need extra synchronizations which makes kernel fusion less beneficial than the thread dependence only cases. The speedup in case (c) is larger than (b) because (c) fuses some thread dependence operators. Considering the reported CPU and GPU computation performance difference [18, 12], the baseline GPU implementation should be 4x–40x faster than CPU and kernel fusion can further increase the GPU advantage.

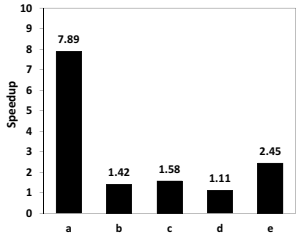

Figure 16: Speedup in execution time (Small Inputs).

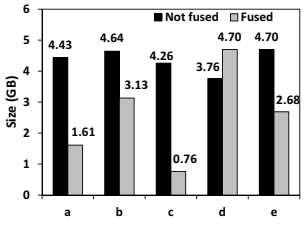

Figure 17: GPU global memory allocation reduction.

The next set of experiments examine the benefits claimed in Section 2.3, specifically the improvement in GPU global memory usage, total memory access cycles and compiler efficacy.

Figure 17 shows the GPU global memory allocated and used with and without kernel fusion. The additional memory without fusion is attributed to large intermediate results. In pattern (d) however, the fused operator uses a little more memory because the fused compute stage has to store two outputs in memory for future gather rather than one. Similarly, Figure 18 shows the data for GPU memory access cycles (collected using the *clock()* intrinsic). On average, fusion reduces the GPU global memory access time by 59%. Finally, Figure 19 quantifies the impact of the compiler. All micro-benchmarks are compiled with *-O3* and *-O0* flags, both with and without kernel fusion. The figure shows the speedups achieved by *-O3* compared to *-O0*. Clearly, kernel fusion enables the the compiler to perform better optimization.

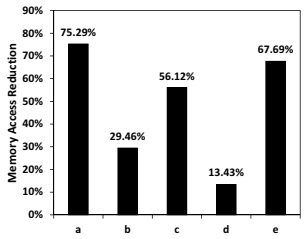

Figure 18: Reduced memory cycles with kernel fusion.

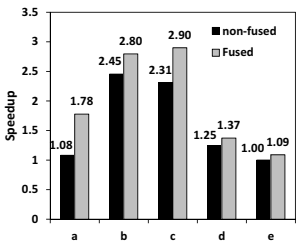

Figure 19: Comparison of compiler optimization impact.

When fusing two or three SELECTs together (e.g., pattern (a)), the second or third SELECT might have some idle threads because some data are not matched in the earlier SELECT. This might impact the overall performance. Figure 20 examines the performance sensitivity of kernel fusion to the selection ratio (percentage of data matching selection condition) with randomly generated 32-bit integers. The results shows fusing two 10% SELECTs produces (more idle threads) 1.28x speedup while fusing two 90% SELECTs (less idle threads) produces 2.01x speedup. Thus, it is fair to say that idle threads may impact the performance but do not negate the benefits of data movement reduction.

5.1.3. Large Inputs In this experiment, the program inputs are enlarged so that every operator has to move its result data back to host to make room for the next operator when kernel fusion is not used. But the problem size still fits the GPU memory when running fused kernels. This set of tests examines the effect of kernel fusion on reducing PCIe data traffic. The input data is generated on the CPU and the final results are sent back to the CPU. Figure 21 compares the execution time with and without kernel fusion over a wide range of problem sizes. In this figure, the execution time comprises two parts: GPU computation time and PCIe transfer time. On average, kernel fusion achieves **2.91x** speedup in the GPU computation time, **2.08x** speedup in PCIe data transfer, and **1.98x** speedup overall. Computation time speedup is similar to the small input case because performance scales with data size. The speedup of PCIe transfer dominates the overall speedup because it is the bottleneck for RA operators. The case (d)

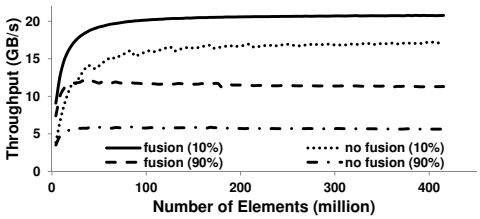

Figure 20: Sensitivity to selection ratio.

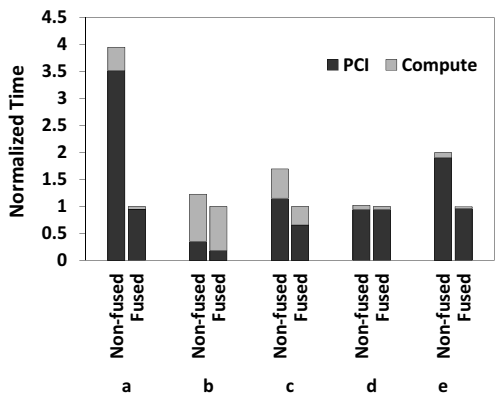

Figure 21: GPU execution time with and without kernel fusion (Large Inputs).

with input independence does not enjoy any benefit from reducing PCIe transfer because the fused version has the same data footprint as the baseline. Considering only the four producer-consumer cases, kernel fusion has **2.35x** speedup in PCIe and a **2.22x** speedup overall. If compared with CPU only systems, due to the large amount of time spent in PCIe shown in Figure 21, the computation performance gap between CPU and GPU would be reduced or GPU could even lag behind of CPU for these simple micro-benchmarks even with the help of kernel fusion. Other techniques discussed in the Related Work section are needed for a complete GPU-assisted database system.

5.1.4. Resource Usage Table 3 lists the GPU resource usage and occupancy (active warps / maximum active warps) of the individual operators and the fused patterns. Since resources are finite, utilizing too many resources per thread may decrease the occupancy. We obtain resource information from *ptxas* and occupancy from *CUDA_Occupancy_calculator*. The left four columns list the resources used by individual operators (e.g. 1 PROJECT needs 11 PTX registers and 0 byte shared memory), and the right four columns show the usage of each pattern after kernel fusion (e.g. fused pattern (a) uses 22 PTX registers and around 2.3K shared memory). The statistics indicate that kernel fusion in most cases increases the resource usage which is the same as the impact of loop fusion, and consequently may lower occupancy (pattern (b) – pattern(e)). Taking pattern (b) as an example, it requires 55 PTX registers and about 23K shared memory with fusion. However, if running two JOINs back-to-back sequentially, each JOIN only needs 47 PTX registers and 13K shared memory. Pattern (a) will use less shared memory after kernel fusion than a single SELECT because i) thread dependence does not use shared memory to store temporary results and ii) the data type of fused results array buffered in shared memory uses smaller data type since PROJECT removes some attributes.

5.2. Real Queries

In this section, we evaluate kernel fusion with two real queries from TPC-H benchmark suites, Q1 and Q21. Q1 represents arithmetic centric queries and Q21 represents relational centric queries. TPC-H queries are very complex (e.g. the 15 operators of Q1 maps to 107 kernels to execute). While the microbenchmarks were compiled and executed with the Datalog front-end, the query plans for the two TPC-H queries presented here were created manually. The additional language support required in the front-end to also automatically compile all Datalog TPC-H queries is being completed for open source distribution of the compiler.

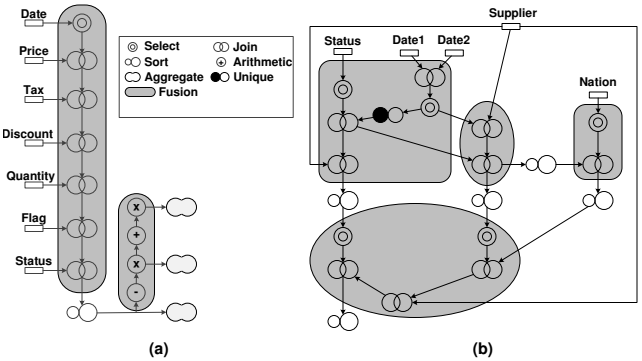

Figure 22: (a) Query plan for Q1; (b) Query plan for Q21.

Q1 calculates several price statistics for selected entries. Figure 22(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) arithmetic calculations over several fields of the table. The first part of the query including one SELECT and six JOINs can be fused into one operator. All of the arithmetic computations performed as the final part of the query can be fused as well. The SORT operator causes kernel dependence and cannot be fused because it has to wait for the completion of the JOINs and arithmetic operations have to wait for the completion of the SORT.

From the TPC-H database, query Q21 identifies suppliers who were not able to ship required parts in a timely manner. Compared with Q1, Q21 has less arithmetic computation but many more relational operations. Figure 22(b) is its simplified query plan (simple operators such as PROJECTs are omitted for clarity). Just as with Q1, SORTs form a boundary for the application of kernel fusion since they can not be fused with their producers nor their consumers.

We tested two queries with 200 MB to 1 GB data and averaged the performance. For Q1, the most time consuming part is the SORT operator which takes around 71% of the total execution time, but cannot be optimized. However, fusion dramatically speeds up the other operators and contributes an overall **1.25x** speedup. Further study shows that when SORT is excluded, the remaining operators can be fused and fusion achieves a **3.18x** speedup due to the fusing of 6 JOINS and 1 SELECTs into a single kernel. For Q21, kernel fusion realizes a **1.22x** overall speedup, which is significant given the complexity of the operators.

The fused patterns for Q1 and Q21 are built based on JOIN operators (e.g. joining several columns together into a larger table and then performing different cross-field computations). These patterns appear very frequently in all 22 queries of TPC-H so that they can all get similar speedup from kernel fusion.

6. Discussion

The proposed framework, Kernel Weaver, opens a door to a class of new optimizations that can be applied in different situations. The following discusses three possible opportunities.

Different Domain: Instead of database applications, kernel fusion can also be used in other domains such as dense linear algebra. The requirement is that the application should use multi-stage algorithms and the stages are independent of each other so that they can be weaved into a new format. The classification of dependences used in this paper is still useful as a guide to fusion candidate selections.

Different Representation: In this paper, the optimization occurs at the CUDA source code level. However, the same technology can

	PTX Reg #	Shared Mem (Byte)	Occupancy (%)		PTX Reg #	Shared Mem (Byte)	Occupancy (%)
PROJECT	11	0	100	(a)	22	2308	88
SELECT	22	3848	88	(b)	55	23560	33
JOIN	47	13580	38	(c)	62	23048	17
+/-	10	0	100	(d)	30	4612	67
Multiply	13	0	100	(e)	27	0	75

Table 3: Resource usage and occupancy of individual (left) and fused (right) operators.

be applied to different representations, such as OpenCL [24], CUDA PTX or LLVM [25], as long as each stage of the operator can be represented. Thus, kernel fusion can be implemented as a module of a static compiler or a JIT compiler that optimizes the representations online.

Different Platform: Furthermore, kernel fusion can be considered as a general cross-kernel optimization that is not only restricted to GPU devices. The benefits of smaller data footprints and larger optimization scope still applies if the CPU program is optimized using kernel fusion. Thus, if using an execution model translator such as Ocelot [11], and a runtime manager such as Harmony [10] it is possible to execute fused kernels on both the CPU and GPU to fully utilize the available computation power.

Moreover, a more complicated fusion framework can use invariant analysis to reschedule operators and to fuse those which are not originally executed back-to-back. For example, if switching the order of SORT and SELECT of Figure 9(c) does not alter the final result, the switch brings more opportunity to optimize since SELECT can thus fuse with the operators before SORT.

7. Related Work

For decades, academia and industry have invested a great deal of effort in query optimization for traditional CPU-based relational database management systems (RDMS) [22, 29]. These query optimizations originated from different perspectives, considered different factors, and made different tradeoffs. Take the CPU cache as an example - the database system can choose among techniques such as cache prefetching, cache partitioning, cache compression, and so on to minimize cache misses and miss penalties [19]. This paper mainly uses shared memory to apply kernel fusion. Compared with the CPU cache, GPU shared memory is i) completely programmable which provides more flexibility, ii) accessed by a large number of threads which forces us to keep concurrency in mind. Thus, the optimization here differs quite a bit.

The idea of kernel fusion arises from loop fusion [23], a well studied loop optimization technique, which can reduce loop traversal overhead and improve certain types of data locality. It is also used in loop parallelization since it can aggregate a large loop body.

The most similar to our work is that of Sato et al. [35], who built a system to run general primitives, map, reduce and zipwith on GPUs with kernel fusion enabled. They fused the CPU code of the primitives and then inserted CUDA runtime library calls and other CUDA required language features to turn a C program into a CUDA program. Thus, they did not exploit the advantageous characteristics of GPUs, such as the multi-level memory hierarchy, that can improve performance. There are also some domain specific kernel fusion techniques targeting GPUs. Copperhead, developed by Catanzaro et al. [7], attempted to fuse a subset of Python primitives to reduce global synchronizations when accelerating them using GPUs. They classified dependence into *local* and *global* which are similar to the thread and kernel dependence of this paper, and fuse primitives having *local* dependence. Thus, they can only fuse a few simple primitives. Chakravarty et al. [8] noticed the benefits of kernel fusion in accelerating Haskell array operations with GPUs and listed it as their future work. On the CPU side, Lee et al. [27] propose a runtime framework, Thread Tailor, which uses fusion techniques albeit at a different level of granularity. Their framework partitions an application into a large number of threads and use a 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. [18] 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 RA primitives [36, 26, 38, 28, 15, 16]. Similarly, Bakkum et al. [4] modified the virtual machine infrastructure of SQLite to use GPUs to execute SQLite opcodes (not RA primitives). All of these previous works 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 enabled by the GPUs and suggest the use of data compression techniques to reduce the amount of transfered data [13]. Our work differs in that we are seeking to discover and develop mainstream compiler passes that can automatically provide inter-kernel optimizations.

To further boost the performance of a GPU assisted database system, other techniques including but not limited to PCIe data compression [13], double buffer [41], and GPU aware query optimizer, are also important to reduce the PCIe hazard. These techniques are orthogonal to kernel fusion because they are independent of the contents transfered over PCIe and can be applied together with kernel fusion. As to larger systems having multiple GPUs or even spanning over multiple nodes, the runtime should have an intelligent scheduling module that can balance the work load of each device (CPU and GPU) and minimize the data movement over the interconnections [10].

8. Conclusion

This paper proposes a cross-kernel optimization framework, Kernel Weaver, that can apply kernel fusion optimization to improve the performance of relational algebra primitives used in data warehousing applications on GPUs. Kernel fusion aggregates larger body of code that can reuse as much data as possible. It can reduce the data traffic through the memory hierarchy caused by the I/O bound nature of database applications, and also enlarge the optimization scope.

To automate the process of kernel fusion, this paper first classifies the producer-consumer dependence between RA operators into three categories: thread, CTA and kernel dependence. Then, Kernel Weaver leverages the multi-stage algorithm design to weave stages from operators having thread and CTA dependence. The experiments shows that kernel fusion optimization brings **2.89x** speedup in GPU computation, **2.35x** speedup in PCI transfer on average across the micro-benchmarks tested. The same technique can be applied to different domain, different representation format and different devices.

Acknowledgements

This research was supported in part by the National Science Foundation under grants IIP-1032032 & CCF 0905459, by LogicBlox Corporation, and by equipment grants from NVIDIA Corporation. We also acknowledge the detailed and constructive comments of the reviewers.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley, 1995, vol. 8.

[2] 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.

[3] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," *Lecture Notes in Computer Science*, vol. 3109, pp. 400–408, 2004. Available: http://www.springerlink.com/content/yth9h90y94n10l7e

[4] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[5] N. Bell, S. Dalton, and L. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," NVIDIA Corporation, NVIDIA Technical Report NVR-2011-002, Jun. 2011.

[6] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 159–166. Available: http://doi.acm.org/10.1145/1572769.1572795

[7] 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. Available: http://doi.acm.org/10.1145/1941553.1941562

[8] M. Chakravarty *et al.*, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. ACM, 2011, pp. 3–14.

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

[10] G. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 197–200.

[11] G. Diamos *et al.*, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proceedings of PACT '10*. ACM, 2010, pp. 353–364.

[12] G. Diamos *et al.*, "Efficient relational algebra algorithms and data structures for gpu," CERCS, Georgia Institute of Technology, Tech. Rep. GIT-CERCS-12-01, Feb. 2012.

[13] 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.

[14] W. Fang *et al.*, "Frequent itemset mining on graphics processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. ACM, 2009, pp. 34–42.

[15] N. Govindaraju *et al.*, "Gputerasort: high performance graphics coprocessor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 325–336.

[16] N. Govindaraju *et al.*, "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.

[17] I. Grebnov, "libbsc: A high performance data compression library," http://libbsc.com/default.aspx, November 2011.

[18] B. He *et al.*, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 21, 2009.

[19] B. He and Q. Luo, "Cache-oblivious databases: Limitations and opportunities," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 2, p. 8, 2008.

[20] T. Hetherington *et al.*, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2012.

[21] S. Huang, T. Green, and B. Loo, "Datalog and emerging applications: an interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 1213–1216.

[22] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing surveys (CsUR)*, vol. 16, no. 2, pp. 111–152, 1984.

[23] K. Kennedy and K. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," *Languages and Compilers for Parallel Computing*, pp. 301–320, 1994.

[24] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[25] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proc. of the 2004 International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[26] T. Lauer *et al.*, "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.

[27] J. Lee *et al.*, "Thread Tailor : Dynamically Weaving Threads Together for Efficient , Adaptive Parallel Applications," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, 2010.

[28] 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.

[29] M. Mannino, P. Chu, and T. Sager, "Statistical profile estimation in database systems," *ACM Computing Surveys (CSUR)*, vol. 20, no. 3, pp. 191–221, 1988.

[30] W. mei W. Hwu and D. Kirk, "Proven algorithmic techniques for many-core processors," http://impact.crhc.illinois.edu/gpucourses/courses/sslecture/lecture2-gather-scatter-2010.pdf, 2011.

[31] D. Merrill and A. Grimshaw, "Revisiting sorting for gpgpu stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Tech. Rep. CS2010-03, 2010.

[32] 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.

[33] S. G. Parker *et al.*, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics*, vol. 29, pp. 66:1–66:13, July 2010.

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

[35] S. Sato and H. Iwasaki, "A skeletal parallel framework with fusion optimizer for gpgpu programming," *Programming Languages and Systems*, pp. 79–94, 2009.

[36] 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.

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

[38] 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.

[39] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–8, 2010. Available: http://www.ncbi.nlm.nih.gov/pubmed/21088027

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

[41] H. Wu *et al.*, "Optimizing data warehousing applications for gpus using kernel fusion/fission," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 2433–2442.