Relational Algorithms for Multi-Bulk-Synchronous Processors

Gregory Diamos
Nvidia Research
Santa Clara, CA, USA
gdiamos@nvidia.com

Haicheng Wu
Georgia Institute of Technology
Atlanta, GA, USA
hwu36@gatech.edu

Jin Wang
Georgia Institute of Technology
Atlanta, GA, USA
jin.wang@gatech.edu

Ashwin Lele
Georgia Institute of Technology
Atlanta, GA, USA
alele3@gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, GA, USA
sudha@ece.gatech.edu

Abstract

Relational databases remain an important application infrastructure for organizing and analyzing massive volumes of data. At the same time, processor architectures are increasingly gravitating towards Multi-Bulk-Synchronous processor (Multi-BSP) architectures employing throughput-optimized memory systems, lightweight multithreading, and Single-Instruction Multiple-Data (SIMD) core organizations. This paper explores the mapping of primitive relational algebra operations onto such architectures to improve the throughput of data warehousing applications built on relational databases.

Categories and Subject Descriptors H.2.4 [*Database Management*]: System—Relational Database, Query Processing

Keywords Relational Algebra, GPGPU

1. Introduction

Modern relational database systems and languages are built on efficient implementations of relational algebra (RA) operators combined with specialized data structures that are used to store relations. These systems have been deployed with success on single-core processors and clusters. However, as power-constrained processor architectures move towards multiple cores with fine grained SIMD parallelism and non-uniform or user-managed memory hierarchies (e.g. modern GPGPUs), new algorithms are needed that can harness the massive parallelism provided by these processors.

Relational operations capture the high level semantics of an application in terms of a series of bulk operations on relations. The data intensive nature of relations might suggest that a high degree of data parallelism could be discovered in RA operations. Unfortunately, this parallelism is generally more unstructured and irregular than other domain specific operations complicating the design of efficient parallel implementations. In particular, we identify a fundamental conflict between the structure of algorithms with good computational complexity and that of algorithms with memory access patterns and instruction schedules that achieve peak machine utilization. To reconcile this conflict, our design space exploration converges on a hybrid multi-stage algorithm that devotes a small amount of the total runtime to prune input data sets using an irregular algorithm with good computational complexity. The partial results are then fed into a regular algorithm that achieves near peak machine utilization. These algorithms can be used directly to implement a relational database system in a single node using a single GPU blade, or as building blocks in higher level distributed algorithms that scale to multiple processors within a node or across multiple nodes.

2. Data Structure and Algorithm Design

RA consists of a set of fundamental transformations that are applied to relations. A relation consists of n-ary tuples that map attributes (or dimensions) to values. Each attribute consists of a finite set of possible values and an n-ary tuple is a list of n values, one for each attribute. Each transformation included in RA performs an operation on a relation, producing a new relation. Many operators divide the tuple attributes into key attributes and value attributes. In these operations, the key attributes are considered by the operator and the value attributes are treated as payload data which are not considered. In this work, relation is stored as a weakly ordered densely packed array of tuples for efficient access in GPUs.

A relational database application is specified as a dataflow graph of operators, making for a natural mapping to a variety of parallel execution models, for example, by mapping operators to Multi-BSP kernels and relations to data structures. RA operators include SET family (*UNION*, *INTERSECTION*, *DIFFERENCE*), *CROSS PRODUCT*, *JOIN*, *SELECT*, and *PROJECT*. All operators are designed to have the same sequence of stages (**partition**, **compute**, and **gather**) where each stage has a Multi-BSP structure, which eases further cross-operator optimization [1]. The philosophy of the algorithm design is to increase core utilizations (achievable throughput) until the computation becomes memory bound, and then achieve near peak utilization of the memory interface. This paper only introduces the implementation of the most complex operator, *JOIN*. The implementation of the other operators can be found in a technical report [2].

Figure 1 shows the three stages of a *JOIN*. The initial **partition** stage (Figure 1(a)–(b)) operates on a sorted list and is performed in-place and the partitions are sized as follows. One of the input relations is partitioned into N parts bounded by pivot elements. Each partition will be processed by a single cooperative thread array (CTA). A binary search is used to lookup the tuples in the other input corresponding to the pivots creating a corresponding series of partitions in the second array. The partitions in the two inputs now have overlapping index ranges of tuples. This stage is critical for sparse data sets because it quickly discards large segments of the input relations that do not overlap.

The **compute** stage (Figure 1(c)) is the most complex stages of the three. It identifies subsets of the partitioned relations with overlapping attributes and performs the cross product for each subset.

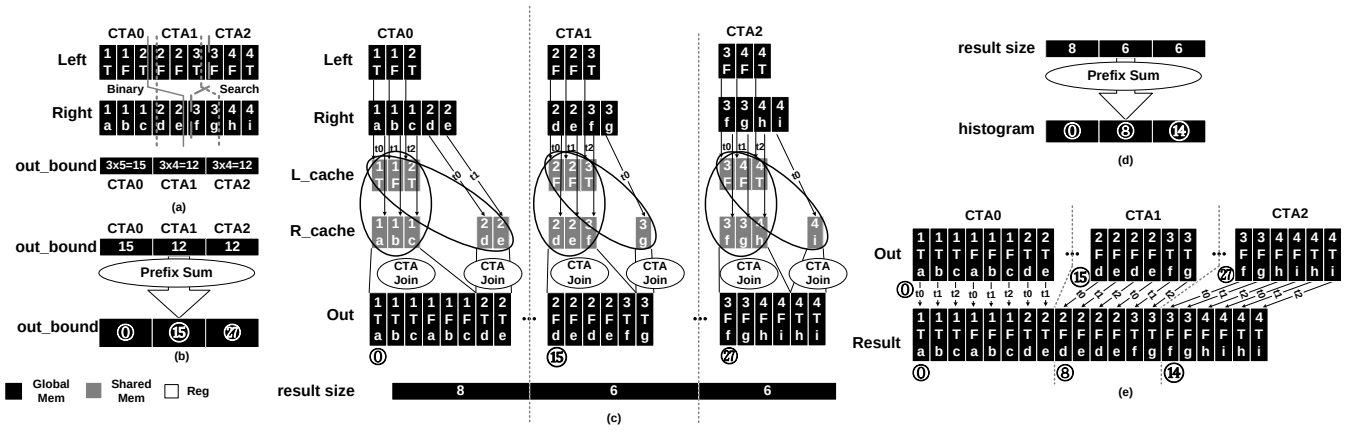

Figure 1. Example of the *JOIN*.

This presents a significant problem to parallel implementations of the algorithm that eventually write to a statically allocated dense array. Namely, the number of tuples in each partition of the output relation is unknown until very late in the computation. The algorithm is that once the inputs have been partitioned, each pair of partitions is assigned to a separate CTA to perform the merge operation independently. The merge operation is implemented by scanning one of the input partitions one *chunk* of tuples at a time where a chunk is a number of tuples that can be processed by a fixed number of threads in a CTA. This chunk is loaded into shared memory for fast access. A corresponding chunk from the second input partition is also loaded into shared memory and a CTA-JOIN is performed upon two chunks within a CTA. Chunks from the second input partition are scanned until they go out of range of the first chunk, at which time a new chunk is loaded from the first partition and the process is repeated. The results of CTA-JOIN are gathered into shared memory until a threshold is reached and eventually written out to a preallocated temporary array. The chunk copy operations into and out of shared memory are carefully designed such that they maximize DRAM bandwidth. The default CTA-JOIN algorithm is referred as Binary-Search CTA-JOIN. Other alternative implementations are described in the technical report [2].

Binary-Search CTA-JOIN is based on a parallel binary search, similar to the **partition** stage of the complete join algorithm. Each thread accesses a tuple from one of the input relations and computes the upper bound and lower bound of that tuple's key in the other relation. The elements between the two bounds match the tuple's key and are joined together. Results generated are aggregated using the stream compaction algorithm and buffered until a threshold number of tuples is reached. At this time, the buffer is written out completely to global memory. Even though this implementation has good algorithmic complexity, it suffers in terms of work-efficiency and processor utilization. It includes a chain of data-dependent loads to shared memory and control-dependent branches. Furthermore, the binary search result of different threads may overlap presenting an opportunity for shared memory bank conflicts and instruction replays when combining two tuples. The other CTA-JOIN algorithms make different trade-off decisions to address these problems.

The final **gather** stage requires first computing the position of each partition of the result in the final array. This is performed by updating a histogram during the **compute** stage, followed by an out-of-place scan operation over the histogram buckets (Figure 1(d)). Again, the number of partitions is sized such that this operation is relatively inexpensive compared to the **compute** phase. Once the position of each section in the output relation is determined, elements need to be copied from a temporary buffer for each section into the final array (Figure 1(e)).

3. Performance

We ran a set of experiments on Tesla C2050 GPU to examine the performance of each RA operator algorithm. The size of each input relation is swept from 8192 to 16 million tuples. The tuple attributes are randomly generated 32-bit integers. These algorithms are expected to be memory bound, and the results are presented in achieved bandwidth. The most efficient algorithms (*PRODUCT*, *PROJECT*, and *SELECT*) achieve $86\% - 92\%$ of peak machine performance (achieved bandwidth of an optimized stream-copy benchmark) across all input data sets. The least efficient algorithm (*JOIN*) achieves $57\% - 72\%$ of peak machine performance depending on the density of the input. The detailed analysis is in the technical report [2]. To the best of our knowledge, our algorithms represent the best known published results to date for any implementations.

Acknowledgments

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] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 '12, pages 107–118, 2012.

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