

# Efficient Relational Algebra Algorithms and Data Structures for GPU

(Regular Submission)

Gregory Frederick Damos  
NVIDIA Research  
gdamos@nvidia.com

Haicheng Wu  
Georgia Institute of Technology  
hwu36@gatech.edu

Ashwin Lele  
Georgia Institute of Technology  
alele@gatech.edu

Jin Wang  
Georgia Institute of Technology  
jin.wang@gatech.edu

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

February 1, 2012

## Abstract

Relational databases remain an important application domain for organizing and analyzing the massive volume of data generated as sensor technology, retail and inventory transactions, social media, computer vision, and new fields continue to evolve. At the same time, processor architectures are beginning to shift towards hierarchical and parallel architectures employing throughput-optimized memory systems, lightweight multi-threading, and Single-Instruction Multiple-Data (SIMD) core organizations. Examples include general purpose graphics processing units (GPUs) such as NVIDIA's Fermi, Intels Sandy Bridge, and AMD's Fusion processors.

This paper explores the mapping of primitive relational algebra operations onto GPUs. In particular, we focus on algorithms and data structure design identifying 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. The design process leading to the most efficient algorithm for each stage is described, detailing alternative implementations, their performance characteristics, and an explanation of why they were ultimately abandoned.

The least efficient algorithm (JOIN) achieves 57% – 72% of peak machine performance depending on the density of the input. The most efficient algorithms (PRODUCT, PROJECT, and SELECT) achieve 86% – 92% of peak machine performance across all input data sets. To the best of our knowledge, these represent the best known published results to date for any implementations. This work lays the foundation for the development of a relational database system that achieves good scalability on a Multi-level Bulk-Synchronous-Parallel (Multi-BSP) processor architecture exemplified by modern GPUs.

**Keywords:** GPU, Database, Primitive, Data Structure

# 1 Introduction

Modern relational database systems are built on efficient implementations of relational algebra operators combined with specialized data structures, such as B-Trees, 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 such as NVIDIA Kepler [21], AMD GCN [2], and Intel Haswell [15] move towards multiple cores with fine grained SIMD parallelism and non-uniform or user-managed memory hierarchies, 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, such as JOIN or PROJECT, on relations. The data intensive nature of relations might suggest that a high degree of data parallelism could be discovered in relational algebra operations. 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. However, recent work has demonstrated that multi-stage algorithms, such as those common to sorting [19], pattern matching [27], algebraic multi-grid solvers [4], or compression [12], can be efficiently mapped onto massively parallel processors such as programmable GPUs from NVIDIA and AMD.

The core contribution of this paper is the development of hierarchical, Multi-level Bulk-Synchronous-Parallel (Multi-BSP) [26] algorithms that implement relational algebra operators. These algorithms are designed to exploit the hierarchical and data-parallel architectures of modern and future general purpose processors to achieve good scaling and near-peak performance. Our focus is on developing and demonstrating Multi-BSP algorithms for a single GPU processor comprised of hundreds of cores. 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 [20, 28]). To the best of our knowledge, the performance of the implementations described here represent the highest reported throughputs for relational algebra operators to date.

## 2 Background and Related Work

Relational Algebra (RA) consists of a set of fundamental transformations that are applied to sets of primitive elements. Primitive elements consist 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. Another way to think about tuples is as coordinates in an  $n$ -dimensional space. An unordered set of tuples of each type specifies a region in this  $n$ -dimensional space and is termed a 'relation'. 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 that is not considered by the operation. Figure 1 is an example of a relation containing 4 tuples and each tuple has 1 key attribute and 1 value attribute. A relational algebra 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. Table 1 lists the complete set of RA operators and their examples.

This paper uses NVIDIA CUDA as the target platform, but the same proposed concepts apply to other brands of GPUs. Figure 2 shows a typical NVIDIA GPU architecture and elements of the Multi-BSP execution model it implements. A typical CUDA program is composed of a series of multi-threaded kernels. As shown in Figure 2, kernels are composed of parallel work-units called *Cooperative Thread Arrays* (CTAs), that are mapped to Single Instruction Multiple Thread (SIMT) units where each thread has support for independent control flow. Appendix A.1 provides a detailed description of this execution model.

The most recent work closest to what is reported here (for other works please see Appendix A.2 ) is a GPU-based database system, GDB, developed by He et al. [13]. Their GPU implementation is based on

RA Operator	Description	Example
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.	$x = \{(3, \text{True}, a), (4, \text{True}, a), (2, \text{False}, b)\}$ PROJECT [field.0, field.2] $x \rightarrow \{(3, a), (4, a), (2, b)\}$
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.	$x = \{(3, a), (4, a)\}, y = \{(\text{True}, 2)\}$ PRODUCT $x \times y \rightarrow \{(3, a, \text{True}, 2), (4, a, \text{True}, 2)\}$
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.	$x = \{(3, \text{True}, a), (4, \text{True}, a), (2, \text{False}, b), (3, \text{False}, a)\}$ SELECT [field.1, field.0=2] $x \rightarrow \{(2, \text{False}, b)\}$
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.	$x = \{(3, a), (4, a), (2, b)\}, y = \{(0, a), (2, b), (3, c)\}$ INTERSECT $x \times y \rightarrow \{(2, b)\}$
SET UNION	A binary operator that consumes two relations to produce a new relation consisting of tuples with keys that are present in at least one of the input relations.	$x = \{(3, a), (4, a), (2, b)\}, y = \{(0, a), (2, b), (3, c)\}$ UNION $x \times y \rightarrow \{(3, a), (4, a), (2, b), (0, a), (3, c)\}$
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.	$x = \{(3, a), (4, a), (2, b)\}, y = \{(4, a), (3, a)\}$ DIFFERENCE [field.0] $x \times y \rightarrow \{(2, b)\}$
JOIN	A binary operator that intersects on the key attribute and cross product of value attributes. It consumes two relations to produce a new relation consisting of tuples with keys that are present in both of the input relations. It is different than SET INTERSECTION in that it evaluates a subset of the fields in each input relation, and thus must tolerate duplicates. Outer join is a variant that returns all tuples from either the left, right, or both input relations along with the matching tuples.	$x = \{(3, a), (4, a), (2, b)\}, y = \{(0, a), (2, f), (3, c)\}$ JOIN $x \times y \rightarrow \{(3, a, c), (2, b, f)\}$

**Table 1:** The set of relational algebra operations. In the example, the 1st attribute is the "key"

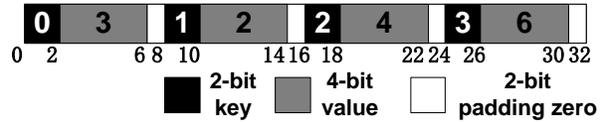
some common primitives such as map, reduce, filter and so on. They compared the performance of each RA operator comprised of these primitives with finely tuned parallel CPU counterparts. As to computation-intensive operators such as JOIN, the speedup of GPU version is 2-7x faster and the other simple operators such as SELECT can achieve 2-4x speedup. Section 4 will compare the algorithm designed in this work with GDB.

### 3 Algorithm Design

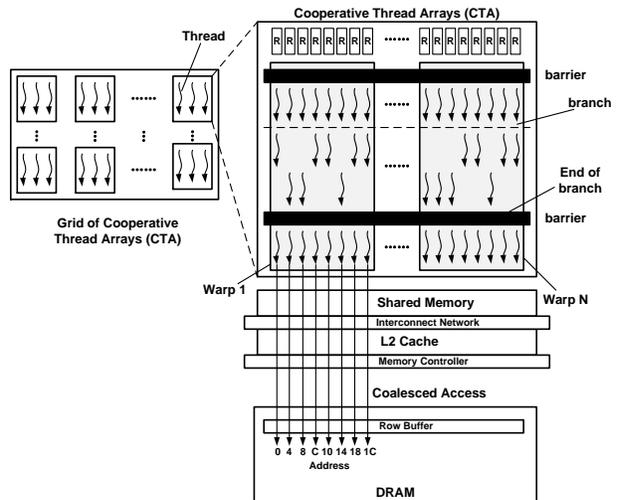
The core algorithmic structure is a sequence of stages where each stage has a Multi-BSP structure. The functionality of some stages is common across the RA operators, e.g., partitioning input sets of tuples. Thus, we are able to abstract the implementations of the RA operators into a few algorithm skeletons whereby different operators can be realized by changing one or more stages of the skeletons. An algorithm skeleton is implemented on a GPU processor. We point out that algorithm skeletons can be composed hierarchically to coordinate execution of multiple operators in a query across multiple GPU processors. However, this paper focuses on the implementation of the algorithm skeletons and the individual stages of each skeleton.

#### 3.1 Algorithm Skeletons

The RA operators are implemented using various algorithm skeletons that allow the same high level algorithm to be readily adapted to operations over complex data types. This approach is commonly used in compilers for high level domain specific languages such as Copperhead [6], Optix [22], or even in generic languages such as Haskell [8].



**Figure 1:** An example of a relation with four tuples, each compressed into 8-bits and packed into a single 32-bit word.



**Figure 2:** NVIDIA C2050 architecture and execution model as Copperhead [6], Optix [22], or even in generic languages such as Haskell [8].

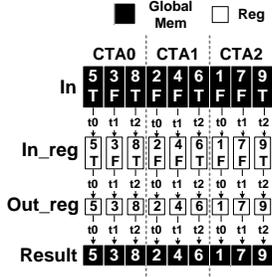


Figure 3: PROJECT

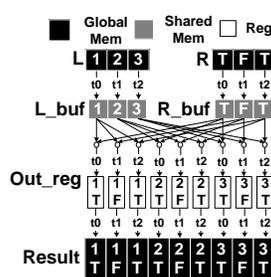


Figure 4: PRODUCT

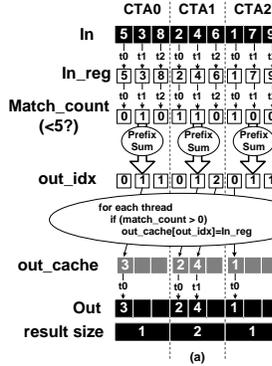


Figure 5: SELECT

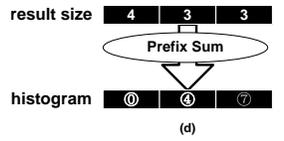
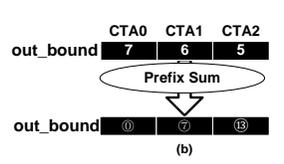
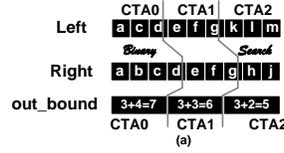
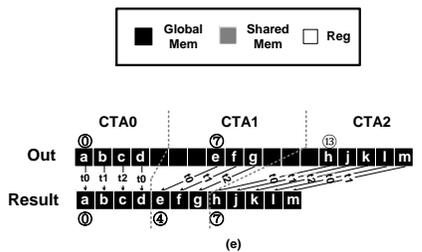
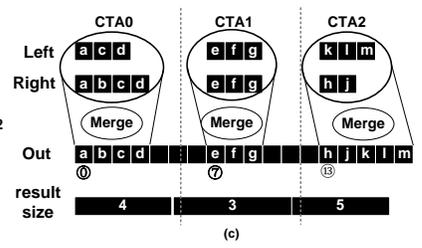


Figure 6: Example of SET Family skeleton (SET UNION)



In the following implementations and analysis, relations are stored as a densely packed array of tuples that is sorted using a comparison function that operates on tuple attributes and defines a strict weak-ordering over all tuples in the relation. For example, in Figure 1, every tuple is packed into a 8-bit byte and four tuples are ordered according to their keys. The sorted form allows for efficient array partitioning and tuple lookup operations. The algorithm skeletons are designed to simultaneously maintain this sorted property on relations as well as achieve good performance on multi-core processors. Additionally, several of the RA operators are similar enough to be implemented with a single skeleton. This is mainly applicable for the SET family operators. The individual skeletons are discussed in detail next:

**PROJECT** is relatively simple compared to the remaining skeletons. As shown in Figure 3, it only requires a transform pass over tuples in the array to decompress the tuple attributes, remove some attributes, and then compress the tuple with the reduced set of attributes. This is a completely data-parallel operation that can be trivially partitioned among a large number of threads and CTAs.

**PRODUCT** is also a simple operation (Figure 4). The number of tuples in the output relation is the product of the number of tuples in the input relations, and computing this size does not require a separate pass. Therefore, the algorithm is implemented using a single pass that iterates over each combination of elements in each of the input relations and invokes a user defined function to combine them. The input tuples is first buffered in the shared memory because they are used multiple times. The combined tuple is written out to the output relation. No buffering stage is required during output because the results are generated as contiguous, densely packed chunks.

**SELECT** uses a user-defined predicate function to examine the key of each tuple. If the comparison succeeds, the tuple needs to be copied into the output relation, otherwise it is discarded. The algorithm is implemented as two passes over the relation. Each pass requires a global synchronization operation. In CUDA, this currently forces the results to be written out to global memory and the implementation to be partitioned into three independent kernels. So, the final CUDA implementation will actually consist of a series of kernels that operate on a shared set of variables representing the relations. As shown in Figure 5(a), the first pass follows the stream compaction approach [5], but it uses shared memory and registers carefully. First, the sorted input relation is partitioned and each CTA is in charge of one partition. Every thread reads in one tuple into a register

and evaluates the tuple using the predicate function. A register called *match\_count* records the comparison result. Each CTA performs an out-of-place prefix sum on the *match\_count* registers to compute the output indices of the matched tuples. Thus, these matched tuples are buffed into the shared memory at the positions determined by the previous prefix sum step and written back to the memory using bulk transfers to improve memory efficiency. Another array, *result size*, recorded how many tuples are selected by each CTA and second prefix sum operation on this array can determine the position in the output array to be written by each CTA (Figure 5(b)). If the tuple number each CTA needs to process is larger than the number of threads, every thread must loop over several tuples and this loop can be unrolled to further reduce the computation time. The second pass is used to eliminate the gap between the result obtained by each CTA by using a coalesced memory to memory copy, a common CUDA programming pattern [18] (Figure 5(c)). The final gather stage is also used by SET family operators and JOIN in a similar way.

**SET Family** operators include SET INTERSECTION, SET UNION and SET DIFFERENCE. It is comprised of three major stages, 1) the input relations are partitioned into independent sections that can be processed in parallel (Figure 6(a)), 2) the sections are merged together using a specialized function for each RA operator generating independent sets of results (Figure 6(c)), and 3) the individual sections are gathered together back into a dense sorted array of tuples (Figure 6(e)). Each of the three stages is expected to be memory bound on the target processors. The last stage is similar as the gather stage of SELECT and the first two stages are describe in the following.

The initial partitioning stage is performed in-place and the partitions are sized as follows such that the partitioning does not consume a significant portion of the total execution time. The sorted property of the input relations is exploited to perform a recursive double-sided binary search that progressively places more restrictive bounds on the input relations, creating sections that may contain tuples with overlapping keys. The double-sided binary search is a parallel implementation of the algorithm presented by Baeza-Yates et al. [3]. It works by partitioning one of the input relations into  $N$  sections bounded by pivot elements, then using a binary search to lookup the tuples in the other input corresponding to the pivots creating a series of partitions with overlapping index ranges of tuples in the two relations. This process can be repeated recursively by swapping the inputs, and subdividing the individual partitions using the same method. In this implementation, the process is repeated until the partitions have been reduced to a predetermined size. The initial stage partitions the arrays into one partition for each CTA, and the recursive stages are handled within each CTA, allowing for maximum concurrency.

The merge operation is the most complex of the three stages. Once the inputs have been partitioned, each pair of partitions is assigned to a separate CTA to be processed independently. This stage of the algorithm implements a merge operation that is commonly used in CUDA implementations of sorting [7, 19, 23]. 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 on-chip shared memory or registers for fast access. A corresponding chunk from the second input partition is also loaded into shared memory and compared against tuples in the first chunk. The exact comparison function that is used depends on the RA operator being performed, for example, SET INTERSECTION would check for matching tuples in each chunk whereas SET DIFFERENCE would check for the presence of a tuple in one chunk but not the other. 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. When matches are found, they 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. They rely on the round-robin scheduling of warps in a CTA and therefore stripe words from the input chunk across threads in the CTA. Thus, when all memory operations in a warp are concurrently issued by the SIMD unit, these are coalesced into a single transaction. Multiple memory requests can be be coalesced into successive transactions to the same memory controller, ideally hitting the same row buffer as long as they are not interleaved with orthogonal transactions from neighboring CTAs. Additionally, the comparison operations among chunk elements are cooperatively

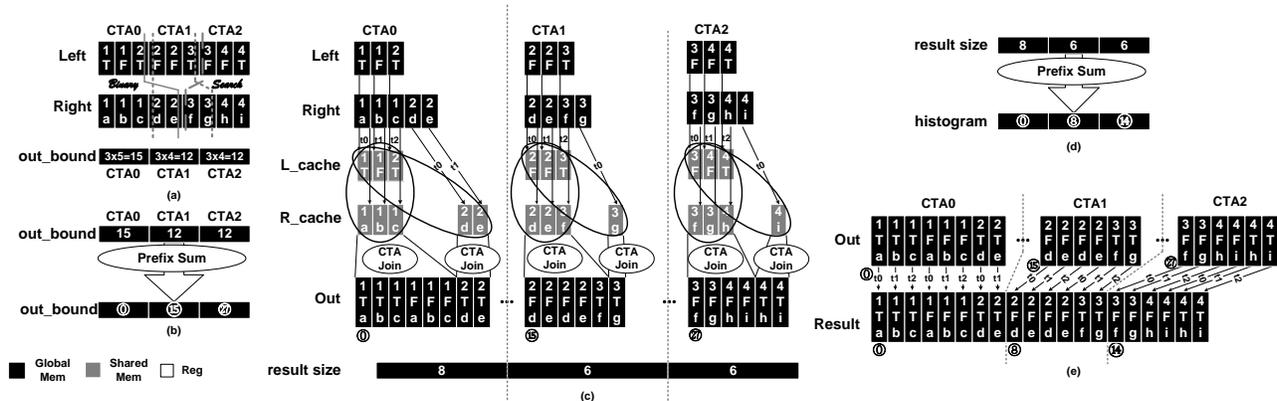


Figure 7: Example skeleton of JOIN

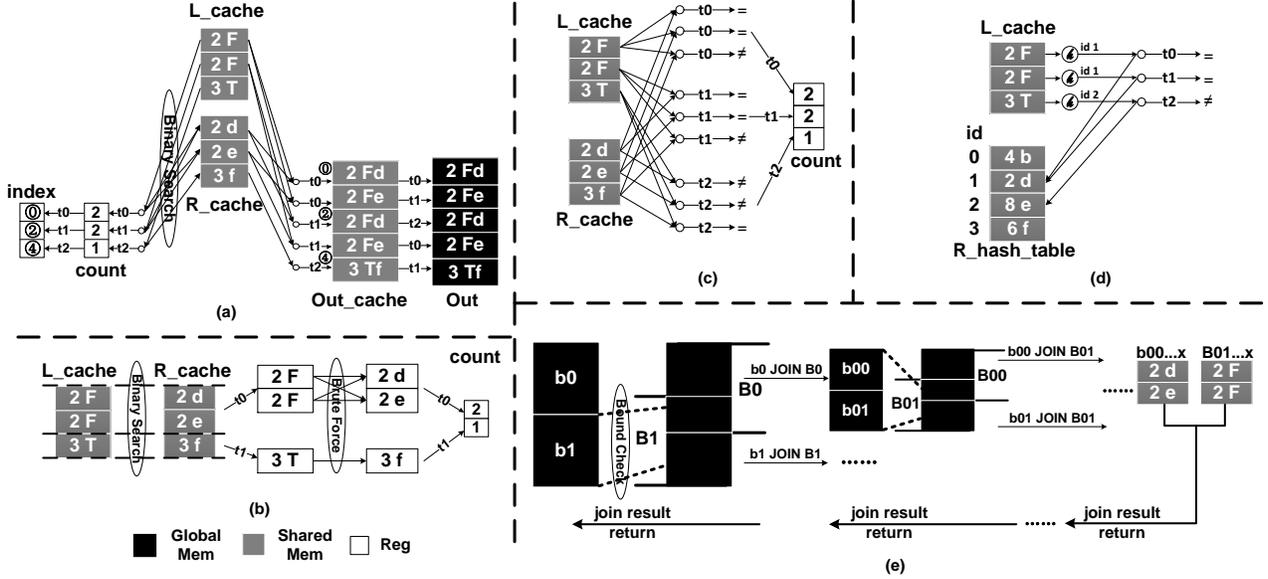
performed by multiple threads and scheduled by unrolling loops and inserting barriers at the CUDA level to eliminate sequential instruction dependences. Merge is compute bound and the most computationally intensive stage of the SET operators.

**JOIN** is easily the most complicated algorithm, sharing many characteristics with the skeleton used for the SET operations (Figure 7). However, it is further complicated by the merge stage of the algorithm, which involves identifying subsets of the partitioned relations with overlapping attributes and performing the cross product for each subset. This presents a significant problem to parallel implementations of the algorithm that eventually write to a statically allocated, dense array. Namely, that the number of tuples in each section of the output relation is not known until very late in the computation. At one extreme, there could be a single section with matching attribute values containing all the tuples in each array, in which case the JOIN would become a **PRODUCT**. At the other extreme, sections could contain at most one tuple each, in which case the JOIN would become a **SET INTERSECTION**. This problem is partially addressed by the initial partitioning stage. Sections from the input relations must always fall into the same section, so a pair of input relations, that have been partitioned into  $M$  sections (each of size  $N$ ) may only produce as many as  $M * N^2$  tuples in the output relation in the worst case, rather than  $(M * N)^2$  in the general case. However, temporary storage still needs to be allocated for  $N^2$  possible tuples rather than  $N$  tuples in the **SET INTERSECTION** case, limiting the number of partitions that can be processed independently. Section 3.2 describes several alternative implementations for the join operation within a CTA - recall in the merge stage joins are performed by each CTA. We will refer to these as CTA-Join algorithms.

### 3.2 CTA-Join Algorithms

**Binary-Search** (Figure 8(a)) is based on a parallel binary search, similar to the first stage of the complete join algorithm (Partition). 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 *lower\_bound* and the *upper\_bound* 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. All CTA join algorithms use this approach of stream compaction followed by buffering and coalesced writes to global memory. The major difference is in computing the matching elements from each input relation. So, the rest figures in Figure 8 only include the element matching part.

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. For example,  $t0$  and  $t1$  in Figure 8(a) will both access the first two tuples of *R\_cache*, but  $t2$  needs to access the third tuple. The



**Figure 8:** CTA Join Algorithms: (a) Binary-Search; (b) Register Blocked Binary-Search; (c) Brute-Force; (d) Hash-Table; (e)Join-Network.

combination of these behaviors give the binary search algorithm poor performance on the Fermi pipeline.

**Register Blocked Binary-Search** (Figure 8(b)) relies on the observation that when performing a binary search, neighboring threads will likely access the same tuples in the beginning of the search. Additionally, the last few levels of the search are the most prone to bank conflicts. This implementation extends the Binary-Search CTA-Join algorithm by combining the work of multiple threads in the original algorithm into a single thread. Instead of processing a tuple from the first relation, a thread will access a sequential range of tuples and store them directly in registers. It will then compute the *lower\_bound* of the first tuple and the *upper\_bound* of the last tuple, this defines the range in shared memory that may contain matching tuples. It then performs a brute force linear search for each tuple from the first relation against this range.

**Brute-Force** (Figure 8(c)) is the simplest implementation of CTA-join which uses a full brute-force comparison of all tuples in one relation to all tuples in the other relation. Although this algorithm has poor computational complexity, the regular nature of the computation allows for an implementation with high ILP, no pipeline hazards, and nearly optimal use of shared memory. Each thread processes several tuples from the first relation at a time, which are stored directly in registers (one tuple in Figure 8(c) for simplicity). Then, the threads sweep through all of the tuples in the second relation, utilizing the broadcast capability of shared memory.

**Hash Join** (Figure 8(d)) is usually accomplished with atomic operations on global memory, for example, with atomic exchange operations to implement cuckoo hashing [1]. These approaches are too slow to be competitive with the complete join algorithm presented here. To use with CTA-join, it is necessary to implement a hash table in on-chip shared memory. Unfortunately, NVIDIA’s Fermi GPUs do not support shared memory atomic operations in hardware. With built-in atomic operations off the table, it is necessary to consider alternatives that exploit the consistency properties of shared memory. One solution is to implement a fast hash table in shared memory by allowing collisions during insertions, and then detecting them a-posteriori. Each thread gathers a set of tuples from one of the input relations, evaluates a hash function using the tuple’s key, and attempts to write the tuple into a shared memory table at the position indicated by the hash function, along with a unique id for the thread. Once all participating threads have inserted their tuple(s), they read back the value to determine whether or not there has been a collision. If there has been, they set a flag indicating that they need to execute again at a later time, otherwise they proceed normally. Next, all threads cooperatively read tuples from the other relation and look up their keys in the hash table. Matches are stored as outputs. Finally, if there were any collisions, the process is repeated with the threads that successfully inserted into the hash table

masked off during the insertion process. It is interesting to note the similarities between this algorithm and the approach for handling bank conflicts in shared memory.

**Join-Network** (Figure 8(e)) attempts to implement the operator as a comparator network. For joins on relations without any tuples with duplicate keys, this is relatively straightforward. However, the duplicate keys make the maximum number of elements that could be generated by the join the cross product of the sizes of the input relations. Even if the common case does not produce such a large number of results, the statically defined comparisons and connections in a comparator network would result in a network with  $N * (\log N)^2$  comparators for a  $N$  JOIN  $N$  case, rather than the  $N * \log N$  comparators of a similarly sized sorting network. In order to address this problem, the join-network algorithm introduces side-exits in the static network. At each stage of the network, both of the two input relations are partitioned into two groups ( $(b0, b1)$  and  $(B0, B1)$ ). Pivot elements are chosen and compared, for example, the last tuple in  $b0$  with the first tuple in  $B1$ . Depending on the results of the comparison, each input group is routed into one of two output groups. For example, after routing, the groups may contain  $(b0, B0)$  and  $(b1, B1)$ . After this stage, each group is joined independently. In this example  $(b0$  joins  $B0)$  and  $(b1$  joins  $B1)$  are performed. In the case that the results of the pivot comparisons determine  $b0$  need to join  $B1$ , a side-exit from the join-network will be triggered and a binary search join will be invoked at that point. The algorithm only includes one branch per stage to check if any side exit was taken, limiting the number of branches to  $\log N$ . To reduce the probability of groups of tuples requiring extra checks,  $B0$  and  $B1$  are partially overlapped to enlarge their individual range.

### 3.3 A Suitable Data Structure

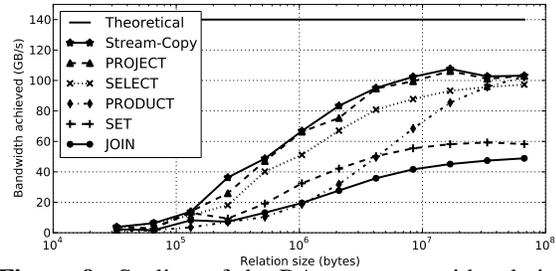
In addition to these algorithms, a data structure is needed to store relations. Many relational database systems rely on variants of B-Trees to maintain the sorted property of relations while providing efficient insert/erase operations and effectively matching the page structure of the memory and disk hierarchy. Unfortunately, using this data structure in conjunction with Multi-BSP algorithms is complicated by the need to support a large number of parallel updates during a bulk operation. Specifically, updates that potentially propagate up the tree during an insertion or deletion create dependences and serial bottlenecks when millions of updates are occurring in parallel. However, supporting fine-grained update operations is not necessary for parallel implementations of RA operators, which performs bulk-operations on entire relations at a time. This paper considers two data structures, i) dense sorted arrays and ii) sorted linked lists of large pages.

A simple dense sorted array sorts tuples by key attribute. It has the advantages that data can be easily streamed in sorted order, offering the potential for a traversal over the entire relation that achieves peak memory bandwidth. As long as updates are made during the construction of the array as bulk operations, overheads can be kept to a minimum. Additionally, efficient searches over the data structure may be accomplished with a binary search. The main disadvantage of the dense sorted array is the inability to modify it once it has been created, to update it in parallel without knowing the complete size of each update, or to even create it before its final size is known.

Dynamically Sized List of Pages attempts to address above issues by allowing several pools of data pages to be updated and resized independently, while still maintaining the efficiency of dense streaming accesses by allocating data on a large page granularity. In this data structure, each pool begins with a single or a few pages of data that is managed by a single CTA. As data is generated, it is written to the currently allocated page without any potential of collision with other CTAs. When the current page is exhausted, a new page can be allocated from a memory pool with an atomic pointer increment, and linked to the end of the current page. After an algorithm completes, the pages generated by each CTA can be linked together to form a globally sorted list of pages, avoiding the cost of the global gather operation needed by a dense sorted array. The pages should be sized appropriately such that the cost of allocating a new page and linking pages together is significantly less than the cost of writing data to the page. This data structure can also be easily converted to and from a dense sorted array with a stream-copy operation that can proceed at near full bandwidth. This may be necessary for interoperability with highly tuned implementations of SORT or other algorithms, although

CPU	AMD Phenom 9550 Quad-Core 2.2Ghz
GPU	NVIDIA Tesla C2050
Memory	8GB DDR-1333 DRAM
CPU Compiler	GCC-4.6.0
GPU Compiler	NVCC-3.2

**Table 2:** The microbenchmark test system configuration. All of the benchmarks are run only on the GPU.



**Figure 9:** Scaling of the RA operators with relation size.

it would also be straightforward to implement a modified version of these algorithms with an initial step that accessed data in this paged format. For future machines with additional levels of hierarchy, the same approach can be applied recursively to create a deeper hierarchy of pages. It is interesting to note that this approach has many similarities to the concept of INODE pages used in the UNIX file system.

## 4 Experimental Evaluation

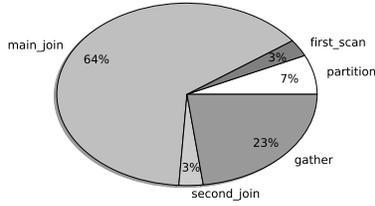
This section covers a low level performance evaluation of this system by benchmarking and analyzing the performance of each of the skeleton algorithms. The tuple attributes are compressed into 32-bit integers. It is the role of a high level compiler to change the word size used to store each tuple in cases where more attributes are required. However, tuples from many common applications can fit into 32-bit or 64-bit words, and this common case is studied here. The tuples attribute values were generated randomly, and each relation contains approximately 16 million unique tuples. The system configuration is given in Table 2. These algorithms are expected to be memory bound on the GPU used in the test system, and the results are presented in achieved bandwidth.

Figure 9 shows the throughput of each RA skeleton algorithms. Throughput is computed as the size of all tuples moved to and from DRAM divided by the total execution time of the algorithm. The theoretical peak memory bandwidth of the used GPU device is 144GB/s, the maximum achieved bandwidth using an optimized stream-copy benchmark is 107 GB/s. PROJECT and PRODUCT achieve more than 99% of the simple copy benchmark. As the algorithms become more complicated, the performance reduces, generally because of less regular memory accesses or multi-stage algorithms that require accessing each tuple multiple times. SELECT achieves the next best performance which is as high as 94.3%, as it simply requires a series of linear scans over the input relations that can be performed at close to peak bandwidth. The gather after the stream compaction also occur linearly within a CTA, leading to highly efficient operations. The SET operators closely follow the performance of the JOIN operator, which is described in the next section.

### 4.1 JOIN Stages

The performance of the complete JOIN operator is highly sensitive to the data distribution, ranging from between 57 GB/s and 72 GB/s depending on whether the data in each relation is randomly generated or perfectly aligned respectively. Figure 10 breaks down the execution time of JOIN to the time taken by each stage. The main join kernel consumes more time than any other stages. If the dynamically sized list of pages described in Section 3.3 is used instead, the final scan and gather can be avoided. This decreases the amount of data transferred by a factor of twice the generated results size.

As shown in Figure 9, unlike the previous operators, JOIN’s performance is significantly less than the peak DRAM bandwidth, indicating that it is either compute bound or not using the bandwidth effectively. Each of the C2050’s 14 SMs must issue  $107 \div 14 = 7.64$ GB/s of bandwidth in order to saturate the DRAM interface. Assuming an average IPC of 1 and a clock rate of 1.1 GHz, the SM (32 threads per warp) can tolerate  $(12 \times 32) \div (7.64 \div 1.1) = 55$  cycles for each pair of inputs and one 32-bit tuple it outputs. This includes cycles



**Figure 10:** The distribution of JOIN execution time

Algorithm	Threads	Blocking	Inputs	Outputs	Cycles	Cycles-Per-Output
Brute Force	512	2	1024	441	97461	228
Binary Search	512	1	512	217	12369	57
Blocked Binary Search	512	4	2048	856	43656	51
Hash Join	512	4	2048	856	34240	40
Join Network	512	4	2048	856	31672	37

**Table 3:** Microbenchmark results for each of the CTA-Join algorithms.

for loading input tuples, staging data in shared memory, performing the join, and writing out the results. Using the C2050’s timestamp clock register, it was determined that the data staging through shared memory requires 24 cycles, the two-finger join requires 21 cycles, which leaves 10 cycles for CTA-Join to keep the Join operator memory bound. Table 3 shows the average cycles taken by different CTA join algorithms to process each input tuple. This result presents the algorithm tuning parameters that give the best performance after a design sweep. Examples of tuning parameters include threads-per-CTA, shared memory buffer size, loop unrolling factors, and register blocking factors.

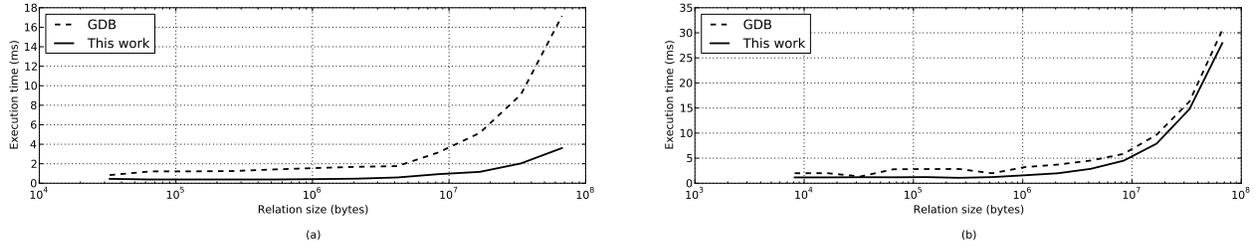
Brute Force Join performs an all-to-all comparison among all pairs of tuples from each of the two input relations. Although this implementation has high ILP, abundant TLP, and few pipeline hazards, the results in Table 3 show that the number of operations it performs makes it prohibitively expensive. The prefix sum requires 32 cycles while the join itself require 196 cycles for each tuple that is generated. It is likely that this implementation would only be effective in the case where nearly all tuples from each relation had matching keys and the JOIN degenerated into a PRODUCT.

Binary Search computes the upper and lower bound, and a prefix sum to determine the position in the output buffer to store the results. 36 cycles are spent in prefix sum and join takes 21 cycles.

Register Blocked Binary Search gives a small but noticeable performance improvement over the binary search join. The time for the join increases because the lower levels of the search tree are performed with a brute-force comparison, increasing the number of comparisons. However, the total time is reduced because each thread computes several results, and require fewer iterations in the prefix sum. These results solidify the methodology proposed by Davidson et. al. [9] that the ILP improvements offered by register packing improve the performance of algorithms that can be mapped onto reduction trees, such as binary-search.

Hash Join is one of the best performing implementations of CTA-Join at 28 cycles for the prefix sum and only 12 cycles for the actual join. The process of inserting into the table, detecting conflicts, and performing the lookup (which has a fair probability of causing bank conflicts) may be longer than a single round of the binary search. However, the hash implementation requires fewer iterations compared to the nine required by the binary search over 512 elements. The best size for the hash table was twice the size of the input relation. This provided a good balance between minimizing conflicts and maximizing the occupancy of the SM. It is quite surprising that this implementation is so fast, given the lack of shared memory atomic operations. While this is not a true hash table in the sense that tuples that collide are not inserted back into the table before the first round of comparisons, it still may be a useful technique for other applications such as pattern matching that only need to check for the existence of an element. It is also interesting to note that the hash table is quite amenable to register blocking, where each thread attempts to insert multiple tuples before checking for a collision. In this case, the best performance is achieved with a blocking factor of 4.

Join Network takes the fewest cycles among all examined algorithms (28 for the prefix sum and only 9 for the actual join) because all comparisons and move operations are determined statically for a network of a given size. This allows all loops to be fully unrolled, shared memory accesses to be scheduled to avoid bank conflicts, and the final few stages to be coalesced into operations out of registers performed by independent threads. Additionally, there are fewer comparisons performed than the binary search, because the upper levels



**Figure 11:** (a) Comparison of SELECT; (b) Comparison of JOIN

of the binary search are performed by a few threads and then broadcast to the other threads, which further subdivide the operation.

It is interesting to note that all of these algorithms are compute bound. None of them can sustain the maximum DRAM memory bandwidth of the C2050. Instead, they are mainly bound by pipeline hazards due to dependencies on on-chip memory accesses, control or dataflow dependencies.

## 4.2 Comparison with Previous Work

Last set of experiments compare the performance of SELECT and JOIN between this work and the previous work, GDB [13]. SELECT stands for efficient operators and JOIN represents less efficient operators. The implementation of GDB comes from the authors’ website without any modification. The inputs to two implementations are identical.

Figure 11 is the comparison result. In average, SELECT of this work is 3.54x faster than GDB and JOIN is 1.69x faster. As to SELECT, the advantage of this work is that GDB does not use shared memory to reduce the data access time although GDB is also based on the stream compaction approach. The performance improvement of JOIN mainly comes from our finely tuned CTA join algorithm.

## 5 Discussion and Conclusion

One limitation of this paper is that it only presents in-memory algorithms designed for a single processor. While these algorithms have the advantage that they can be easily scaled to future processors that implement a Multi-BSP execution model, current systems are limited to a fixed number of SMs per processor, and even more troubling, a fixed amount of DRAM in the 1-12GB range. Some database systems operate on relations several orders of magnitude larger than this. For existing database systems, this problem is typically mitigated in one of three ways: i) using out-of-core algorithms, ii) using in-memory algorithms with multiple nodes, and iii) using a single node with a large amount of memory (systems with 256GB or more are common). As to the first method, out-of-core algorithms can be handled by connecting a parallel processor to a disk subsystem. Commodity interfaces like PCIe offer greater bandwidth than disk interfaces, although the performance of the system may be limited by disk access time rather than by the in-memory portion of an RA operator. As to the second method, Multi-BSP processors can be replicated to form a distributed system, although these algorithms would need to be coupled to a higher level distributed algorithm. It should be noted that the high level structure of the algorithms described in this paper may scale to multiple nodes, since they are already highly parallel by design. For the last method, there are no technical reasons impeding the development of GPUs with very large DRAM subsystems similar to CPU systems, only a lack of OEMs that offer such systems.

As a conclusion, this paper presents how to use multi-stage algorithm to design efficient RA operators in GPUs for database systems. The algorithm design and their performance characteristics is discussed in length. The experiment result shows that GPU implementation of RA operators can achieve 57% – 72% peak machine performance for complex operators such as JOIN and 86% – 92% for simple operators such as PRODUCT, PROJECT and SELECT. Compared with previous work did by He et al. [13], our implementation is 1.69x-3.54x faster. This study stands as a successful example of the parallel algorithm design for a highly unstructured problem traditionally thought to be unsuitable for execution on a Multi-BSP processor.

## References

- [1] D.A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J.D. Owens, and N. Amenta. Building an efficient hash table on the gpu. *Gpu Computing Gems Jade Edition*, page 39, 2011.
- [2] AMD, 2011. [http://en.wikipedia.org/wiki/Graphics\\_Core\\_Next](http://en.wikipedia.org/wiki/Graphics_Core_Next).
- [3] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. *Lecture Notes in Computer Science*, 3109:400–408, 2004.
- [4] Nathan Bell, Steven Dalton, and Luke Olson. Exposing fine-grained parallelism in algebraic multigrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, June 2011.
- [5] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM.
- [6] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPOPP '11, pages 47–56, New York, NY, USA, 2011. ACM.
- [7] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, January 2010.
- [8] M.M.T. Chakravarty, G. Keller, S. Lee, T.L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, pages 3–14. ACM, 2011.
- [9] Andrew Davidson and John D. Owens. Register packing for cyclic reduction: a case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 4:1–4:6, New York, NY, USA, 2011. ACM.
- [10] 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*, pages 325–336. ACM, 2006.
- [11] N.K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM, 2004.
- [12] Ilya Grebnov. libbsc: A high performance data compression library. <http://libbsc.com/default.aspx>, November 2011.
- [13] B. He, M. Lu, K. Yang, R. Fang, N.K. Govindaraju, Q. Luo, and P.V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.
- [14] J. Hoberock and N. Bell. Thrust: C++ template library for cuda, 2009.
- [15] Intel, 2011. [http://en.wikipedia.org/wiki/Haswell\\_\(microarchitecture\)](http://en.wikipedia.org/wiki/Haswell_(microarchitecture)).
- [16] T. Lauer, A. Datta, Z. Khadikov, and C. Anselm. Exploring graphics processing units as parallel co-processors for online aggregation. In *Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*, pages 77–84. ACM, 2010.

- [17] M.D. 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*, pages 1111–1120. IEEE, 2008.
- [18] Wen mei W. Hwu and David Kirk. Proven algorithmic techniques for many-core processors. <http://impact.crhc.illinois.edu/gpucourses/courses/sslecture/lecture2-gather-scatter-2010.pdf>, 2011.
- [19] Duane Merrill and Andrew Grimshaw. Revisiting sorting for gpgpu stream architectures. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.
- [20] NICS, 2010. <https://keeneland.gatech.edu/>.
- [21] NVIDIA, 2011. [http://en.wikipedia.org/wiki/GeForce\\_600\\_Series](http://en.wikipedia.org/wiki/GeForce_600_Series).
- [22] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics*, 29:66:1–66:13, July 2010.
- [23] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus, gpus and intel mic architectures. Technical report, Intel Labs, 2010.
- [24] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106. Eurographics Association, 2007.
- [25] 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*, pages 117–126. ACM, 2009.
- [26] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [27] Panagiotis D Vouzis and Nikolaos V Sahinidis. Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–8, 2010.
- [28] E. Walker. Benchmarking amazon ec2 for high-performance scientific computing. *Usenix Login*, 33(5):18–23, 2008.

# A Appendix

## A.1 Programmable GPU Architecture

The performance of computing platforms has increased over time, driven by advances in manufacturing technology, system/processor architecture, and software stacks. However, the hardware/software interface has also evolved alongside these changes, requiring algorithm or data structure modifications to fully exploit the performance improvements provided by new technology. This section examines how these changes impact RA algorithm design.

### A.1.1 Memory Subsystem

RA operators are highly memory intensive, mainly consisting of simple manipulations of tuple data after routing it through the memory system. Therefore, effective utilization of the memory subsystem is critical to achieving good performance.

Main memory is composed of DRAM memory blocks organized together into an array that compose a single IC. Data is moved from the memory banks into an on-chip row buffer in a single transaction. Multiple row buffer operations can be pipelined to hide the latency of individual transfers. Once in the row buffer, data is transferred synchronously over the processor-memory interface. Transfers occur out of the row buffer at maximum bandwidth when a series of transactions read a sequential burst of data. Otherwise, additional commands are needed to modify the offset being accessed in the row buffer. Several Integrated Circuits (ICs) are packaged together into a single module with data striped across the ICs to form a 64-bit data interface that connects directly to a processor's memory controller. A processor will typically include multiple memory controllers, each connected to series of ICs. The NVIDIA C2050 includes six independent controllers.

These DRAM organizations favor deep queues of coarse-grained bulk operations on large contiguous chunks of data, so that all data that is transferred to the row buffer is returned to the processor, and that it is accessed sequentially as a long burst. Memory controllers can typically support several streams of sequential accesses by buffering requests and switching between streams on a row buffer granularity. However, a large number of requests directed to addresses that map to different row buffers will force the memory controller to switch pages and trigger premature row buffer transfers, reducing effective bandwidth. These characteristics favor algorithms with a limited number of streams that are accessed sequentially, or block-strided accesses on a granularity larger than the DRAM page size, typically 2-4KB. Purely random accesses result in frequent row buffer transfers and address traffic, which significantly reduce effective DRAM bandwidth.

### A.1.2 Network and On-chip Memory

GPU cores are referred to as streaming multi-processors (SMs). The on-chip network sits between the L2 cache banks and SMs. The memory controllers interface with the L2 cache banks, servicing misses from a single group of banks. The network interleaves traffic from multiple SMs, and the L2 cache serves as a buffer that accumulates requests to the same memory controller. For RA algorithms, the fact that the L2 cache is generally smaller than the aggregate size of the SMs shared memory banks or register files makes it useful primarily as a staging area for data that is being ferried to and from the memory controllers.

### A.1.3 Core Microarchitecture

SMs include SRAM banks used to implement register files, L1 data and instruction caches, and scratch-pad memory referred to as shared memory. The datapath follows a SIMD organization with 32 lanes, and a 24 cycle pipeline. Logical threads in the programming model are mapped onto SIMD lanes in the processor in 32-wide groups known as warps. Warps are time-multiplexed onto the SM to cover the pipeline and memory latencies. A scoreboard is used to track instruction dependences and allow the same warp to issue multiple independent instructions. Shared memory is organized into 32 banks, each one supporting an independent access per cycle. Like all processors, datapath hazards and resource constraints limit the peak instruction throughput of the SM. Although most RA operators are memory intensive, they are dependent on the SMs for access to fast on-chip memory and for orchestrating the movement of data throughout the system. SM inefficiencies can quickly become performance bottlenecks.

Threads in a warp that take different control flow paths through the program require serialization, reducing the utilization of the SIMD datapath. At least two warps are required to issue instructions concurrently. Instructions are scheduled speculatively, assuming fixed best-case latency for cache and shared memory accesses. A cache miss or shared memory bank conflict violates the schedule of any instructions from the same warp following the incorrectly-scheduled

operation, causing these instructions to be squashed and the offending instruction to be replayed. Branch prediction is not employed; hardware multi-threading is required to hide branch latency.

For RA, and most other applications, the algorithm design needs a large number of warps to hide memory and pipeline latency. Control flow within a warp should be regular to keep SIMD utilization high. Branches and instructions with back-to-back data dependences should be avoided in place of long streams of instructions with at least a moderate degree of instruction-level-parallelism (ILP).

## A.2 Previous Research

The greatest effort has been applied to the most demanding problems, leading to several effective sequential algorithms for JOIN, SORT, and the SET family of operators. In this section, we focus on the JOIN and SET operators due to a large existing body of work on SORT, and we refer interested readers to [7] for more details. Three classes of algorithms are described that can be used to implement SET or JOIN operators. They are linear merge, hash merge, and recursive partitioning.

**Linear Merge** can be used to perform SET or JOIN operations by co-iterating over each of the input relations. It is the default algorithm used for GCC's implementation of the SET family of operators. It works by maintaining two pointers, one for each of the input relations. At each step of the algorithm, tuples at each pointer are compared. Matching elements are written as outputs. If there is no match, the pointer to the tuple with the lesser key is incremented.

Although this algorithm has linear complexity and a sequential access pattern, naive implementations require several branches at each iteration to determine which pointer to increment. This is further complicated by the fact that the branch is typically data-dependent, leading to poor branch prediction rates and frequent pipeline flushes. Several optimizations can be applied to alleviate these issues, for example, branching overheads can be minimized by unrolling the inner loop and using conditional select instructions to determine which pointer to increment after each iteration. However, this still results in a chain of data-dependent memory operations, where the address of the next load is dependent on the value of the previous load. These cannot be used to hide pipeline or memory latency in a deeply-pipelined out-of-order processor.

**Hash Join** is a very popular algorithm for implementing the JOIN algorithm. The input relations are first partitioned into buckets using a hash function such that tuples with matching keys will be moved into corresponding buckets. In the next stage of the algorithm, each pair of buckets is joined by creating a hash table for the smaller buckets and scanning over the larger bucket, performing lookups into the hash table. Hash join is very amenable to parallel implementations; partitions can be created and processed independently. The main disadvantage of hash based algorithms is their random memory access pattern in the first phase of the algorithm where tuples are scattered into buckets, when the hash tables for the smaller buckets are created, and during the lookups into the hash tables.

**Recursive Partitioning** was introduced by Baeza-Yates et al. [3] as an efficient algorithm for performing set intersections. It begins by selecting a pivot tuple from one sorted relation and looking up the corresponding tuple in the other relation, creating two partitions. The algorithm is then applied recursively to each partition until matching tuples are found, or entire partitions are found to be out of range.

Although it has good computational complexity, it is poorly matched for most DRAM and cache organizations, because the early stages of the algorithm access a single element from an entire cache line or DRAM page and then move on to another element very far away. The cost of this operation can be amortized over multiple accesses to the relation by pre-computing the pivot elements and storing them consecutively as meta-data in the same cache line or DRAM page, effectively creating a B-Tree structure. However, this approach is not effective when relations are generated as intermediate values and discarded immediately. Furthermore, implementing an operation completely with recursive partitioning potentially requires a binary search for each partition, which has a worst case memory access complexity of  $O(N/2 \log N)$ . This compares favorably to other algorithms at the cost of a worse memory access pattern.

**GPGPU Algorithms:** Even before the introduction of general purposes programmable GPUs, Govindaraju et al. [10, 11] leveraged the use of OpenGL/DirectX to speedup the performance of RA operators including aggregation, SELECT and JOIN. After the introduction of CUDA and OpenCL, Trancoso et al. [25] introduced methods for mapping the JOIN operator onto GPUs. Lieberman et al. [17] provided techniques to accelerate similarity join for a spatial database. Sengupta et al. [24] implemented GPU scan primitives which are frequently used in database applications. Lauer et al. [16] used GPUs to optimize aggregation for an online analytic processing (OLAP) system. Furthermore, the CUDA Thrust library [14] provides GPU implementations of many operations that are commonly used in database systems such as aggregation, scan, and all of the set operations.