Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs

Haicheng Wu
School of Electrical and
Computer Engineering
Georgia Institute of Technology
hwu36@gatech.edu

Daniel Zinn Molham Aref
LogicBlox Inc.
{daniel.zinn,molham.aref}
@logicblox.com

Sudhakar Yalamanchili
School of Electrical and
Computer Engineering
Georgia Institute of Technology
sudha@ece.gatech.edu

ABSTRACT

Recent work has demonstrated that the use of programmable GPUs can be advantageous during relational query processing on analytical workloads. In this paper, we take a closer look at graph problems such as finding all triangles and all four-cliques of a graph. In particular, we present two different join algorithms for the GPU. The first is an implementation of Leapfrog-Triejoin (LFTJ), a recently presented worst-case optimal multi-predicate join algorithm. The second is a novel approach, inspired by the former but more suitable for GPU architectures. Our preliminary performance benchmarks show that for both approaches using GPUs is cost-effective. (the GPU implementation outperforms respective CPU variants). While the second algorithm is faster overall, it comes with increased implementation complexity and storage requirements for intermediary results. Furthermore, both our algorithms are competitive with the hand-written C++ implementation for finding triangles and four-cliques in the graph-processing system GraphLab executing on a multi-core CPU.

1. INTRODUCTION

The explosive growth of “Big Data” in the enterprise has been accompanied by a growing demand for increased productivity in developing applications for sophisticated data analysis tasks involving relational analysis over large data sets. *High performance relational computing (HPRC)* has emerged as a discipline that mirrors high performance scientific computing in importance. HPRC promises to profoundly impact the way businesses grow, the way we access services, and how we generate new knowledge. We are interested in relational analysis over data sets organized as conventional relational databases, particularly those that represent graph models in a relational form. Graph problems are essentially relational problems as examplified by this paper and it is a great advantage for a relational database system to own the capability of solving graph problems for the sake of productivity. The explosive growth of graph sizes towards billions of nodes has pushed graph analysis to the forefront of numerous data intensive applications and is the focus of this paper.

Our target systems are cloud systems comprising high performance multi-core processors accelerated with general-purpose graphics processing units (GPUs), such as those from vendors NVIDIA, AMD, and Intel. While discrete GPU accelerators provide massive fine-grained parallelism, higher raw computational throughput, and higher memory bandwidth compared to multi-core CPUs, there are significant algorithmic and engineering challenges in harnessing this performance for relational analysis over graphs.

For example, while queries appear to exhibit significant data parallelism, relationships between nodes are more often irregular and unstructured. Thus, computations over these relations tend to exhibit poor spatial and little temporal locality while modern processors and memory hierarchies are optimized for locality. Further, computations at the nodes of a graph are relatively simple. Thus, algorithms tend to exhibit low compute density, i.e., the ratio of arithmetic operations to memory accesses is lower than traditional engineering computations. Coupled with low locality, queries make poor use of memory bandwidth. Finally, computations over relations are highly data dependent. Control flow and memory access behaviors are difficult to predict while available concurrency is time varying, data dependent, and also difficult to predict. Thus, the ability to harness the tremendous compute and memory bandwidths of GPUs will require algorithmic advances to effectively harness the massive parallelism and memory bandwidths of GPUs. These algorithmic advances is the focus of this paper.

This paper proposes two, multipredicate join algorithms for GPUs. One is a GPU-optimized implementation of Veldhuizen’s Leapfrog Triejoin (LFTJ) [23]. The second algorithm is novel. Both methods effectively fuse multiple join operations at the algorithmic level thereby reducing (intermediate) data traversals through the GPU memory hierarchy and increasing GPU core utilization. The throughput improvements possible with these algorithms are demonstrated on several important graph computations.

This papers seeks to make the following contributions.

1. We describe how to obtain a GPU-efficient implementation of Leapfrog Triejoin. Our efficient implementation is amenable to CPU and GPU execution thereby supporting flexible run time scheduling for execution on the CPU or GPU. For finding triangles on pre-sorted data, our implementation executed on the CPU is 3.5-10X faster than GraphLab [15]. Running on the GPU, we obtain an additional 2-3X performance improvement.

Proceedings of the VLDB Endowment, Vol. 7, No. 14
Copyright 2014 VLDB Endowment 2150-8097/14/10.

2. We propose a second new multipredicate join algorithms for GPUs that is tailored to the GPU architecture and delivers an additional 1.5-10X improvement in throughput on the GPU version from above.

3. The nuances of data structure and algorithm design for these irregular operations on GPUs are described.

4. We apply these algorithms to compute triangles and 4-cliques over a relational representations of graphs to demonstrate the capability of using relational computation to solve graph problems. We evaluate their performance as well as memory requirements.

2. BACKGROUND

2.1 Motivation for General Purpose GPUs

The use of programmable GPUs has appeared as a potential vehicle for an order of magnitude or more performance improvement over traditional CPU-based implementations for large footprint relational query processing. This expectation is motivated by the fact that GPUs have demonstrated significant performance improvements for data intensive scientific applications and the recent emergence of GPU accelerated cloud infrastructures for small and medium enterprises such as Amazon's EC-2 with GPU instances.

The current implementation targets NVIDIA GPUs and therefore we adopt the terminology of the bulk synchronous execution model [22] underlying NVIDIA's CUDA language. Figure 1 shows an abstraction of NVIDIA's GPU architecture and execution model. A CUDA application [17] is composed of a series of multi-threaded data parallel kernels. Data-parallel kernels are composed of a grid of parallel work-units called Cooperative Thread Arrays (CTAs) which in turn consist of an array of threads that may periodically synchronize at CTA-wide barriers. In the processors, threads within a CTA are grouped into logical units known as warps that are mapped to SIMD units called stream multiprocessors (SMs). Hardware warp and thread scheduling hides memory and pipeline latencies. Global memory is used to buffer data between 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.

Performance is maximized when all of the threads in the warp take the same path through the program. However, when threads in a warp do diverge on a branch, i.e., different threads take different paths, performance suffers because the execution of two paths is serialized. This is referred to as branch divergence. Memory divergence occurs when threads in a single warp experience different memory-reference latencies and the entire warp has to wait until all memory references are satisfied.

GPU DRAM organizations favor coalesced memory accesses which are 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. 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. Purely random accesses result in frequent row buffer transfers and address traffic, which significantly reduce effective DRAM bandwidth.

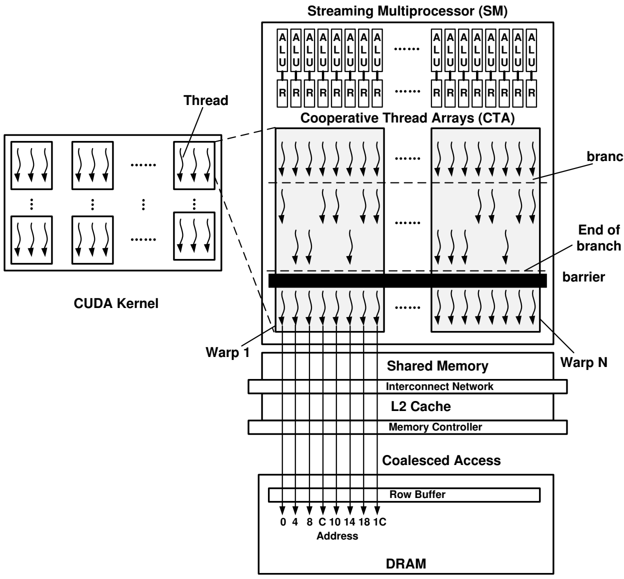


Figure 1: NVIDIA GPU Architecture and Execution Model.

Compared with a traditional multi-core CPU, GPUs have a considerably larger number of (simpler) computation cores and higher memory bandwidth. However, discrete GPUs are interconnected with the CPU via the PCIe interconnect which has relatively much smaller bandwidth compared with either the CPU or GPU memory bandwidth. Moreover, the memory capacity of modern GPU boards is not large - 12GBytes is the largest today. Thus for practical data footprints, efficient staging and management of data movement between the host and GPU memories is very important.

2.2 Clique Problems

Finding triangles is an important operation for graphs; it is used as input for various graph properties and metrics such as the graph's clustering coefficient, triangular connectivity and others [5, 14]. Formally, given a graph $G = (V, E)$ its triangles are all sets $\{x_1, x_2, x_3\}$ such that $(x_i, x_j) \in E$. As is convention, we store each undirected $\{a, b\}$ edge as two directed edges (a, b) and (b, a) in a binary relation `E(x,y)`. The unique triangles are then computed via the following relational query expressed as a Datalog rule:

```
tr(x,y,z) ← E(x,y),E(y,z),E(x,z),x<y<z.
```

Here, the first join `E(x,y),E(y,z)` "computes" all paths of length 2; these are then complemented by the third edge `E(x,z)` to form a triangle. The condition `x<y<z` is necessary since otherwise each triangle would occur 6 times. Similarly, 4-cliques are sets of four nodes such that each node is connected with each other. The Datalog rule to compute all unique 4-cliques is:

```
4cl(x,y,z,w) ← E(x,y),E(x,z),E(x,w),E(y,z),
               E(y,w),E(z,w),x<y<z<w.
```

2.3 Leapfrog Triejoin

Our first GPU join-algorithm is an implementation of leapfrog triejoin (LFTJ) [23] adapted for the GPU. This section thus reviews this join algorithm. For a more detailed exposition please see [23]. We start by describing the basic building block *leapfrog join*, which computes joins between *unary* relations. These are essentially multi-way-intersections. We then generalize to multi-way-joins over general predicates to yield leapfrog triejoin.

2.3.1 Linear Iterators

A unary input relations R is accessed via a *linear iterator* interface that presents the data in *sorted order*. It is convenient to imagine the data to be stored in a sorted array of size $|R| + 1$ where the last cell is left empty. In fact, this is how we will layout data in memory A linear iterator behaves much like a pointer into this array. It is initialized at the first element and provides methods for data access and iterator movement: (1) *bool atEnd*() returns `true` if the iterator is positioned at the last array element (which does not correspond to a data value in R). Note that for an empty relation *atEnd*() will return `true` immediately after initialization. (2) T *value*() returns the data value the iterator is positioned at. It must not be called if the iterator is *atEnd*(). (3) *void next*() moves the iterator to the next array cell. Like *value*(), this method must not be called if *atEnd*() is `true`. (4) *void seek*(x) moves the iterator to the data value x. If x is not in R then the iterator is moved to the smallest element y that is larger than x; or to the last empty cell if such a y does not exist. Must only be called when the iterator is not *atEnd*() and the current *value*() is smaller than x.

2.3.2 Leapfrog Join

The leapfrog join between a series of unary relations (e.g., R, S, and T) behaves somewhat similar to the merge-phase of merge-sort. The crucial difference is that we are only interested in values that occur in all input relations. Thus, if one relation has a large value x we can *skip forward* in the other relations to the value x. Skipping forward is done via the *seek*(.) operator. We will also always seek the iterator that presents the smallest current value to (or past) the value presented by the iterator that is most advanced. This leap-frogging motivates the name of the method.

We present the result of the join via a linear iterator itself. The implementation of the leapfrog join is given in Fig. 2. After **leapfrog_init**, the algorithm maintains the invariant that `Iter[p]` is the iterator with the smallest value while `Iter[p−1 mod k]` has the largest value; and the iterators in between are in ascending order. The core method is **leapfrog_search** where we repeatedly seek the iterator with the smallest value to the iterator with the largest value until all iterators have the same value –a join result is found– or any of the iterators is *atEnd*() indicating that no result will be found anymore.

For a fixed query, leapfrog join's runtime complexity is $O(N_{min} \log(N_{max}/N_{min}))$ where N_{min} and N_{max} are the cardinality of the smallest and largest relation, respectively [23]. This complexity bound holds if the following complexity bounds are satisfied for the linear iterator operations: *key*() and *atEnd*() need to be in $O(1)$; *next*() and *seek*(.) are required to be in $O(\log N)$ where N is the size of the relation. Furthermore, if m values are visited in ascending order, then armortized complexity of *seek*() and *next*() must not exceed $O(1 + \log(N/m))$. With an array representation, these bounds can easily be obtained as described in Section 3. The bounds can also easily be obtained when data is stored in more standard paged data structures such as B-Trees.

Leapfrog join Algorithm [23] as Linear Iterator

```
globals: Array Iter, integer p, bool atEnd

leapfrog_init():
  if any iterator is atEnd():
     atEnd := true
  else:
     sort Iter[0..k−1] by value() of each iterator
     p := 0; atEnd := false
     leapfrog_search()

leapfrog_search():
  max_value := Iter[(p−1) mod k].value()
  while true:
     min_value := Iter[p].key()
     if min_value == max_value:
        return
     else:
        Iter[p].seek(max)
        if Iter[p].atEnd():
           atEnd := true
           return
        else
           max = Iter[p].value()
           p := p + 1 mod k

leapfrog_next():
  Iter[p].next()
  if Iter[p].atEnd():
     atEnd := true
  else:
     p := p + 1 mod k
     leapfrog_search()

leapfrog_seek(seeked_value):
  Iter[p].seek(seeked_value)
  if Iter[p].atEnd():
     atEnd := true
  else:
     p := p + 1 mod k
     leapfrog_search()

leapfrog_atEnd(): return atEnd
leapfrog_value(): return Iter[0].value()
```

Figure 2: Leapfrog-Join algorithm computing the intersection of k unary predicates given as an array $\texttt{Iters}[0..k-1]$ of linear iterators.

Leapfrog Triejoin [23] as Trie-Iterator

```
globals: Array LeapFrogs, integer d

lftj_open():
  d := d + 1
  for each iter used in LeapFrogs[d]:
     iter.open()
  LeapFrogs[d].leapfrog_init()

lftj_up():
  for each iter used in LeapFrogs[d]:
     iter.up()
  d := d − 1 // backtrack to previous var

lftj_value(): return LeapFrogs[d].leapfrog_value()
lftj_next(): LeapFrogs[d].leapfrog_next()
lftj_seek(): LeapFrogs[d].leapfrog_seek()
lftj_atEnd(): LeapFrogs[d].leapfrog_atEnd()
```

Figure 3: Leapfrog Triejoin Implementation

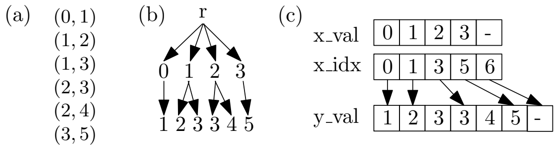


Figure 4: Sample binary relation (a), as Trie (b) and as Trie-Array (c).

2.3.3 Trie-Iterators

Again, following [23], we extend the linear iterators to allow iteration over arbitrary relations. For this, we imagine the data of a relation R to be organized as a Trie (see Fig. 4) Here, every tuple in the relation is represented as a path from the root to a leaf node. Besides the linear iterator operations we add methods that allow moving up and down in the tree: *open*() positions the iterator at the first children of the current node. This method may only be called if the iterator is not *atEnd*(); and *up*() moves the iterator back to the parent node.

An example of how the data of a relation can be examined via trie-iterator operations is provided in Fig. 2.3.3. Note that (1) all children of a node are *sorted* and *unique*, and (2) only the children of a node are represented via a linear iterator and not necessarily all elements of a certain level; for example, if the iterator is positioned at the first 3 from left in the lowest level, and *next*() is called, *atEnd*() is true. To move to the neighboring 3, one needs to call *up*(), *next*(), and *open*().

2.3.4 Leapfrog Triejoin

Leapfrog triejoin (LFTJ) is a multi-predicate-join algorithm ; that is it computes the result of Select-Project-Join queries directly without employing pair-wise joins. We explain LFTJ on the example of

```
T(x,y) ← R(x,y),S(x),T(y)
```

LFTJ is configured by an order of the variables occurring in the body of the defining rule. Furthermore, the sequence of variables in each atom must be a subsequence of the chosen variable ordering. In the example above, we can choose x, y, z because x, y in the first atom, x in the second atom, and y in the third atom are all subsequences of x, y, z.

The order of variables in join atoms can be permuted by deploying alternative indexes: $S(x, y)$ can be presented as $S'(y, x)$ with $S'(y, x) \leftarrow S(x, y)$. The chosen variable ordering, in general, influences the performance of the join evaluation. Finding a good ordering is usually deferred to a query optimizer, and is beyond the scope of this paper.

LFTJ finds result tuples by using leapfrog joins to find assignments to the variables; one leapfrog join is used for each variable. Consider the example from above: first LFTJ performs a leapfrog join between $R(x, \_)$ and $S(x)$; this is done by opening the TrieIterators for R and S and using leapfrog join at the first level of the two Tries. However, whenever a result c for x is found, we open the Trie-iterators for R and T since these have the variable y. We then perform a leapfrog join for y. Once this is finished, we back-track to the join at level x, search for the next match at level x and then descend again to level y once a match has been found.

LFTJ implementation. For each atom in the rule body (e.g., R, S, and T) a TrieIterator is instantiated. Furthermore, we maintain an array of leapfrog joins, one join for each variable according to the variable order. The leapfrog join for a variable X has a pointer to the TrieIterators of those atoms in which X appears. In the above example, the leapfrog join for x points to the TrieIterators for R and S, while the leapfrog join for y points to the TrieIterators for R and T. LFTJ itself is implemented as a TrieIterator that presents the join result; so we only need to give implementations for the TrieIterator interface. We use a variable *depth* to keep track at which variable we currently operate (corresponding to the depth of the LFTJ-Trie). The methods *next*(), *seek*(), *atEnd*(), and *value*() are delegated to the leapfrog join for the active variable. The operation *open*() and *up*() are given in Fig. 3.

To actually obtain the set of resulting tuples, we simply iterate over the Trie provided by the LFTJ.

Trie-Iterators for Non-Materialized Predicates. Notice how the leapfrog triejoin operates on the Trie-Iterator interface rather than on raw data directly. This is beneficial not only for switching out different data storage schemes, but it also allows *infinite relations* to participate in joins. These are beneficial to allow built-in functions such as comparison or arithmetic operators. For example, consider the predicate `smaller_than(x,y)`. When operating over *int*64 the content can be defined as:

$$\texttt{smaller\_than} = \{(x, y) \mid x, y \in int64 \text{ with } x < y\}$$

3. GPU-BASED LEAPFROG TRIEJOIN

Our first GPU algorithm, referred as *LFTJ-GPU*, is an implementation of leapfrog triejoin with minor modifications. These are mostly geared to avoid virtual method calls during join evaluation as virtual methods are expensive on the GPU. On the data-structure level, we store relations in an array-based data structure representing the Trie of the relation. We further implement a rather simplistic parallelization strategy that worked surprisingly well for our data-sets. In the following, we describe each of these parts in more detail.

3.1 The Trie-Array Data Structure

We restrict our attention to cases in which input data fits into the GPU memory. We store the data in sorted arrays that represent the Trie structure. The structure is inspired by the commonly used Compressed-Sparse-Row (CSR) format used for graphs and matrices. This allows us to implement the Trie-Iterator operations very efficiently. As an example, the Trie-Array for the relation A from Fig. 2.3.3 is shown in Fig. 4 on the right. For n-ary relations, we have n arrays each storing the nodes of the Trie at the corresponding depth. The length of the i-th array is the width of the Trie at the i-th level + 1. Furthermore, we have $n - 1$ *index* arrays that match the size of the value-arrays. The number `x_idx[j]` identifies the index of `y_val` that is the first child of the node `j`. Thus the children of a node `x_val[j]` are the elements in `y_val` with positions in the interval from `x_idx[j]` inclusively to `x_idx[j+1]` exclusively.

3.2 Avoiding Virtual Method Calls and Minimizing Branching

While keeping the Trie-Iterator interface as an architectural abstraction to support built-ins and different storage implementations, a conventional implementation would result in virtual method dispatch. This is (1) expensive on GPUs and (2) makes migrating data structures between

CPU and GPU more difficult. We thus designed and implemented a templated version of leapfrog triejoin that avoids virtual method calls entirely. Having compile-time fixed dispatch also allows the compiler to perform more aggressive inlining. Before we describe our templated LFTJ, we present our TrieIterator implementation.

Avoiding branching in TrieIterators. It is usually beneficial to avoid branch divergence. We thus changed the TrieIterator interface to require all operations to provide the current depth at which the operation is applied as a template parameter. This often removes code branches within the TrieIterator since different levels in TrieIterators often have different implementations - which we would need to switch to at runtime based on the depth value. Furthermore, removing branches, often leads to more aggressive inlining by the compiler and the remaining machine-code for operators is often very small. For example, *next*() operations are only a few instructions: increasing the array-index and checking whether the end has reached to set *atEnd* to true; furthermore, *up*() is often implemented as a No-Op and is completely removed from the code by the compiler.

TrieIterator Methods. The TrieIterators for predicates are conceptually easy to implement. Each TrieIterator for an n-ary relation has an $n+1$-sized boolean array where we store *atEnd* values and an n-sized array of integers that specifies the current operator position at each level in the tree. *open*<*d*>() initializes the index for position $d+1$ to the first node in the subtree; *close*() is a No-Op. All implementations for the linear iterator methods except *seek*(), which we treat separately below, are straight-forward. For example: *value*<*d*>() does a single array-lookup using the array and index at depth d. We also experimented with having *initialize*(), *next*(), and *seek*() return a boolean value instead of providing the method *atEnd*() and using a boolean variable array. Here we observed that the variant with *atEnd*() is slightly faster.

The TrieIterator for `LessThan3` or other simple builtin functions are conceptually straight-forward. We omit their details here.

3.2.1 Tuning Seek Performance

Guaranteeing algorithmic complexity bounds. We would like the *seek*() implementation to be of $O(\log N)$ worst-case complexity as well as to satisfy the amortized complexity bound of $O(1 + \log(N/m))$ for accessing m values out of the N elements.

Consider a *seek*(v) operation on an array R. If we use binary search from the current position to the end of the array to find the least-upper-bound of v in R, we do not satisfy the amortized cost requirement[1]. However, there is a simple fix: before binary searching from the middle of the array, find a value b in R that is larger than v by looking ahead $1, 2, 4, \dots$ cells first. Then, simply do binary search from the last checked value (which was still too small) to the just found value b. It can easily be shown that any probing sequence that grows exponentially would work. We found that using 8^i $(1, 8, 64, \dots)$ as a look-ahead sequence works well for triangles and fourcliques. We expect the optimal setting of this parameter to be query and data-dependent.

Performing well on modern hardware. While the above strategy satisfies the complexity bound, it requires random access into the global relation data. We experimented with an N-ary search-tree index on top of the first attribute of the edge relation, which we utilized for seeks – very similar to B-Tree structures. With this index, our CPU performance improved for large graphs (e.g., a 2.2x and 35% improvement for triangles and fourcliques, respectively, with the 100-million edge dataset described in Section 5).

For the GPU version, adding the additional index into global GPU memory did not show any major improvement. We then experimented to put the higher levels of the search-tree into the shared memory. Besides the additional complexity, this approach adds some constant cost when threads start since they need to copy the index from global GPU memory into the shared memory. In general, the index was beneficial (e.g., 26% and 3.2x improvement for triangles and fourcliques, respectively, with the 100-million edge dataset described in Section 5).

3.2.2 Compile-Time Dispatch for LFTJ

We achieve compile-time fixed dispatch by parameterizing the leapfrog triejoin with the Trie-Iterator types and a compile-time data structure that describes at each levels which iterators take part in the join. We then remove runtime-iteration over TrieIterator arrays by compile-time looping which essentially unrolls the loops, which can lead to further optimizations. We heavily use the boost metaprogramming libraries boost::mpl and boost::fusion. These provide compile-time data-structures and control-flow such as loops. We annotated the libraries to allow them to be used in GPU code. Only minor semantic changes are necessary to be performed to leapfrog triejoin: In the compile-time leapfrog triejoin, we cannot simply sort the iterators during initialization of the leapfrog join since the sort-order is data-dependent and thus not known at compile-time. We instead use an initial phase where we perform *seek*() operations conditioned on the relative ordering of the iterators at runtime. Essentially, instead of sorting an array of TrieIterators, we *seek*() them appropriately to have their respective *value*()s in sorted order. Furthermore, we remove the p integer from leapfrog join: looping over the iterators is fully unrolled, and within *next*() or *seek*(), we always move the first iterator first.

Example: Triangles. The triangle-configuration for the template based leapfrog triejoin is shown in Fig. 5. We have a "query compiler" that creates such descriptions from datalog-rule bodies like:

```
e(x,y),e(x,z),e(y,z),LessThan3(x,y,z)
```

The configurations are not designed for ease of specification but for being easily used within our LFTJ implementation. The first group `iterators_t` defines the types of the body atom TrieIterators: three times iterators for binary predicates with integer values, and one TrieIterator that implements a ternary less-than operator over the type int64_t. `boxes_t` defines the types of the variables in the join. Then, each of the `level_x` structs defines which iterators take part in the LFTJ and at which level the iterator is to be used.

3.3 Parallelization

We employ a simplistic parallelization strategy where we partition the search-space across the first variable in the key

[1] Consider an array with values $1, \dots, N$ and perform the sequence of *seek*(i) for $i = 1, \dots, N$.

```
  struct LFTJConfig_0 {
     typedef mpl::vector<
       P_ii_it, P_ii_it, P_ii_it,
       LessThanT<Box<int64_t>,3> > iterators_t;
     typedef mpl::vector<
       Box<int64_t>, Box<int64_t>, Box<int64_t> > boxes_t;
     typedef mpl::vector<
         mpl::pair<mpl::int_<0>, mpl::int_<0> >,
         mpl::pair<mpl::int_<1>, mpl::int_<0> >,
         mpl::pair<mpl::int_<3>, mpl::int_<0> > > level_x;
     typedef mpl::vector<
         mpl::pair<mpl::int_<0>, mpl::int_<1> >,
         mpl::pair<mpl::int_<2>, mpl::int_<0> >,
         mpl::pair<mpl::int_<3>, mpl::int_<1> > > level_y;
     typedef mpl::vector<
         mpl::pair<mpl::int_<1>, mpl::int_<1> >,
         mpl::pair<mpl::int_<2>, mpl::int_<1> >,
         mpl::pair<mpl::int_<3>, mpl::int_<2> > > level_z;
     typedef mpl::vector< level_x, level_y, level_z > l_maps_t;
};
```

Figure 5: Automatically generated C++ configuration for LFTJT to implement the triangle query.

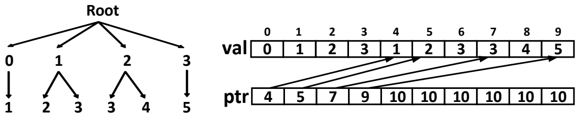
ordering. To do this, we chose a finitie predicate atom (eg., E(x,y)) that has the variable occurring first in the key-order as first attribute. We then determine the number of distinct values D the atom has in the first attribute (this is simply the size of its first value array). We then divide up the index space $[0, \ldots, D]$ into small evenly sized intervals; and assign each interval to a worker-thread. In each GPU thread, we then simply advise the TrieIterator to only consider the small interval when presenting the values for x. Choosing a good interval size generally involves a trade-off: Smaller intervals are better for load-balancing; while larger intervals minimize the overall thread initialization cost. However, we observed that (1) our thread-initialization cost is very low; that is restricting the iterator is very fast (simple computation based on the thread global index), and (2) thread launching is very fast. To the extend that choosing a single-valued interval yielded the best performance for both triangles and 4-cliques in our GPU experiments.

No explicit representation of all work-items is created during the computation. The interval that is to be considered by each thread is directly computed by uniformly mapping the total number of threads T over the interval space $[0, \ldots, D]$ at the beginning of the thread. If T is larger than D then each thread is assigned the single-value interval; otherwise threads will get larger intervals. This makes the algorithm run correctly independently of the chosen parallel launch configuration.

Skew. We can envision example queries and datasets for which this parallelization strategy is too simple and would cause load-imbalances. In general, the very small batch-size works in our favor. In all our experiments, stragglers were not an issue and the GPU was utilized evenly until the very end. Investigating alternative methods is part of our future work.

3.4 CPU-Variant

The GPU-optimized LFTJ implementation can easily be adapted to run on a regular CPU with multiple threads as well. We also implemented a CPU-based version referred as *LFTJ-CPU*. The only difference between the CPU and GPU version is how "worker-threads" are spawned and how work is distributed. In the CPU version, we launch as many threads as there are (hyper-) cores available in the machine.



	0	1	2	3	4	5	6	7	8	9
val	0	1	2	3	1	2	3	3	4	5
ptr	4	5	7	9	10	10	10	10	10	10

Figure 6: Data structure adopted by *GPU-optimized*

Each of these threads is in an endless loop in which it (1) requests a new interval to work on and then, (2) performs LFTJ on this interval, until no more work is to be done. The work-requests are received by a lock-free data structure that uses atomic additions to give out increasing intervals for $[0, \ldots, D]$. This "scheduler" can now balance giving out smaller intervals (which are better for avoiding stragglers at the end of the computation) with larger intervals that cause less load on the work-scheduler and reduce some initialization overhead. We deploy a simple strategy that initially gives out fixed-size arrays which are then shrinking when the remaining work becomes less and less. In our experiment, an initial interval size of 1000 worked well.

As with the GPU variant, this method generally avoids stragglers (i.e., all threads finish at around the same time) but is not robust against very skewed inputs; we postpone more sophisticated load-balancing to future work.

We will compare the CPU version of templated LFTJ with the GPU version. Only difference between them is the granularity of parallelism and the vastly different architecture of the underlying compute, memory, and thread scheduling.

An interesting avenue for future work is to combine CPU & GPU implementation. With our current setup, this is very easy to do: use the CPU mechanism and let the GPU be a "worker-thread" that is given rather large subsets of the remaining work. The optimal size of the subsets handed out could be dynamically adjusted.

4. GPU-OPTIMIZED APPROACH

In the previous approach, a large number of GPU threads independently run the LFTJ algorithm and vertically chase down their own subtrees to find matching nodes. The second approach, referred as *GPU-optimized*, adopts the exact same configuration (Figure 5) as the *GPU-based* approach with slightly changed data structures, but uses a carefully designed strategy that is specifically optimized for GPUs. As shown in Figure 6, the *GPU-optimized* approach changes the data structure where the value arrays and index arrays from all the levels are concatenated respectively. Correspondingly, the indices stored in the index arrays are updated to reflect the new positions of the element in the value array. Starting from the top level, all GPU threads work together to horizontally intersect one level of nodes belonging to different predicates. The algorithm continues in subsequent levels and finishes when the bottom level is processed. The *GPU-optimized* approach targets to pursue the best performance.

The algorithm uses two high performance primitives: *vectorized sorted search* and *load-balancing search* from ModernGPU library [2]. These two primitives are both designed based on the merge path [7] framework which partitions the workload among each CTA and then among each thread inside the CTAs such that workload balance is guaranteed. The computational complexity of both algorithms is $O(N)$ where N is the input size. Moreover, these two primitives

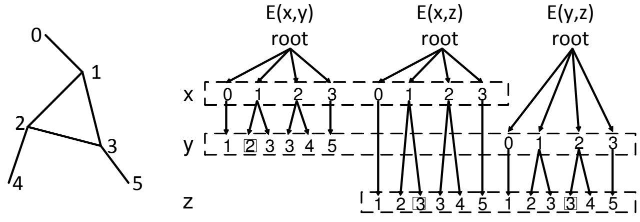


Figure 7: Processing levels by levels from top to bottom. Final results are marked by boxes

are also optimized for coalesced memory access, bank conflicts, instruction level parallelism (ILP), etc.

Vectorized Sorted Search - It reads in two sorted arrays and locates the lower/upper bounds of each element of the first array in the second array. If elements of both input arrays are unique, *vectorized sorted search* can be directly used to intersect two arrays since a simple check of the lower/upper bound with the search key can indicate if the match exists or not.

Load-Balancing Search - It is the reverse operation of exclusive scan. For example, the children count of the first level in Figure 6, is an array of {1,2,2,1}. The exclusive scan of the children count is {0,1,3,5} which corresponds to the position in the output array where the children nodes can be expanded to. Running load-balancing search over the exclusive scan result will generate another array {0,1,1,2,2,3} where each value corresponds to the parent id that can expand its children to this position. E.g., the second element has value 1 means parent 1 can expand its children to this position. Load-balancing search is used by several other primitives of ModernGPU to balance the workloads.

Every predicate has a result array to store the index of remaining elements. This array gets updated after every intersection. Elements at the same position of each index array from its corresponding predicate point to the subtrees that will be expanded and intersected in subsequent levels. These elements are referred as *associated* elements. Predicates that have been intersected are referred as *processed* while the rest are referred as *not processed*. According to the status of the two predicates to be intersected (either *not processed* or *processed*), three cases which use different intersecting procedures need to be considered. The problem of finding triangles (Figure 7) in a graph is shown below as an example to explain these three cases, followed by the description of a general algorithm.

4.1 Examples of Finding Triangles

Case 1: *Not Processed* intersects *Not Processed*

Processing the first level, level x, involves the two predicates `E(x,y)` and `E(x,z)` which are never intersected before. In fact, this case only happens when processing the highest level of two predicates because child level has to be processed after its parent. Moreover, the top level data are always stored consecutively in the array `val`. Therefore, the procedure only requires a normal pairwise intersection. Since the data from each predicate are unique, one *vectorized sorted search* can find all the results. As to the clique-problems, two predicates (e.g. `E(x,y)` and `E(x,z)`) contain the same values in level x so that the intersection results are identical to the inputs. Thus, the intersection can actually be skipped. As shown in Figure 8, in the result arrays belonging to predicates `E(x,y)` and `E(x,z)` respectively, elements in the same position point to the matched values. For example, the first elements of two result index arrays are both 0, which indicates the first elements of `E(x,y)` and `E(x,z)` are matched (the matched value is 0).

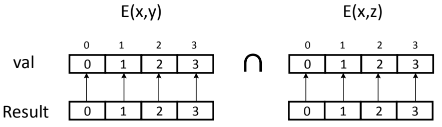


Figure 8: Processing Level x

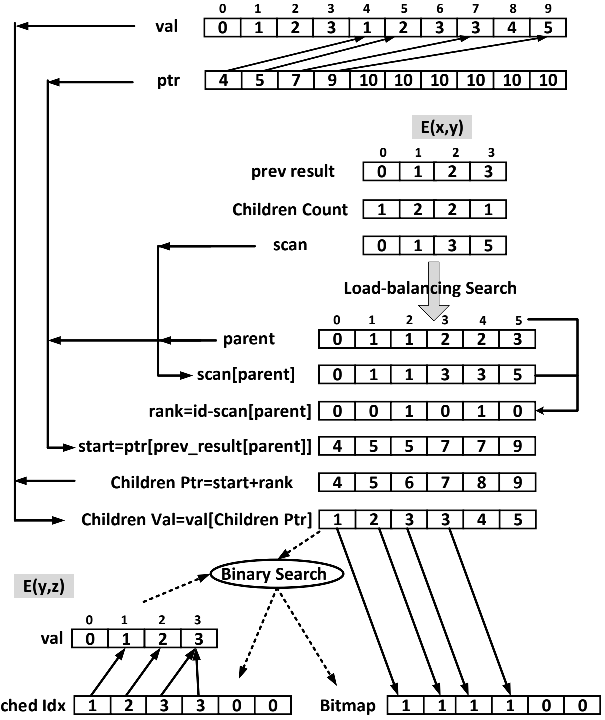


Figure 9: Children Expansion and Binary Search of Level y

Case 2: *Processed* intersects *Not Processed*

Processing the second level, level y, requires intersecting the predicate `E(x,y)` which is processed in the top level and the predicate `E(y,z)` which is a new input to the procedure. Three steps are used in this process: i) expanding the children of `E(x,y)` from level x to level y; ii) searching the matched values in `E(y,z)`; iii) generating the result index arrays for all three predicates.

The purpose of expanding children is that the following intersection step can read in the input as a consecutive array so that the intersection step can evenly partition the input and access the memory in the coalesced pattern. The overhead of expanding depends on how many parents need to be expanded and how many children each parent has. Note that the parents to be expanded may not be consecutive (consecutive in this example but not general cases) and can cause non-coalesced memory accesses. But, the memory access pattern is not completely random because children nodes of the same parent are stored together. Also note that the expanded children come from different parents so that they may not be sorted.

A most straightforward approach to expand the children of `E(x,y)` from level x to level y is i) running exclusive scan of the children count to calculate the output position; ii) mapping the parents to GPU threads such that each thread

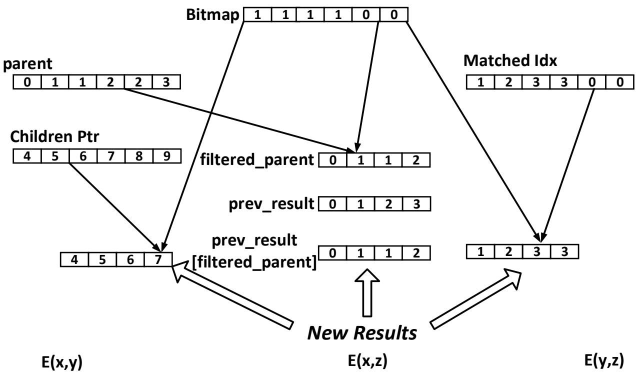


Figure 10: Generating New Results of Level y

expands one parent and stores the children to the calculated position. A major drawback of this approach is that the workload of each thread depends on its children count which varies substantially. Instead, *load-balancing search* is used to guarantee that each GPU thread expands the same amount of children. Figure 9 shows the major computations to calculate the addresses of the children to load the values. In this example, six threads independently expand six children.

Afterwards, the expanded children values search identical counterparts from the top level of `E(y,z)` which is not processed before. *Vectorized sorted search* cannot be used here because one of the inputs (the expanded children from `E(x,y)`) may not be sorted. Instead, a traditional binary search is used. The problem of binary search in GPU is that it will i) generate almost random memory accesses and ii) cause severe control divergence because different GPU threads may take different code paths. Section 5 demonstrate this problem by a set of experiments. The result of binary search is i) a bitmap that shows if a child of `E(x,y)` finds a match ii) and an array that records the matched indices in `E(y,z)`.

The last step is generating the new result index arrays for all predicates. Elements of index arrays generated from `E(x,y)` and `E(y,z)` are already *associated* and point to the same values. While not directly involved in the computation of level y, the result index array of `E(x,z)` from the previous intersection should be updated such that it is also *associated* with the newly generated index array of `E(x,y)` and `E(y,z)` Each updated index array element of `E(x,z)` is *associated* with the parent of each result index array element of `E(x,y)`. Specifically in this example, the updated index array elements of `E(x,z)` should also point to the parents of the new results of `E(x,y)`. The bitmap generated by the binary search is used to filter several different arrays to create the final new results. The detail process is shown in Figure 10. The filtering is implemented as a stream compaction process, a common practice [4] in GPGPU.

Case 3: *Processed* intersects *Processed*

Last level, level z, of the triangle example involves two predicates `E(x,z)` and `E(y,z)` that have already been processed. The computation flow is similar as processing level y, first expanding children for both predicates, then intersecting them and finally generating the results. The children expansion step is shown in Figure 11. It should be noted that some parents have been expanded twice. For example, the `prev_result` of the predicate `E(x,z)` in the figure has two indices 1s and this parent node is expanded twice. Redundant expansion consumes more memory space but eases the intersection part because it can perform better load balancing by using merge-path and more coalesced memory access. The expanded children of both predicates may not be sorted. However, the intersections only apply to the nodes whose parents are *associated* (i.e., the subtrees LFTJ is chasing down) which is indicated by the sorted parent id. Furthermore, children of the same parent are sorted and unique. So, if wrapping the value of each children values with their parent id and treating it as a single value (i.e., parent_id.child_value), the inputs to be intersected can still be considered as sorted and unique. Therefore a simple *vectorized sorted search* can find all matches. Figure 12 shows the major steps in this cases. The output of the intersection is again a bitmap and an array records all matched indices. These two outputs are again used to generate the results of this level as shown in Figure 13 which is similar as level y.

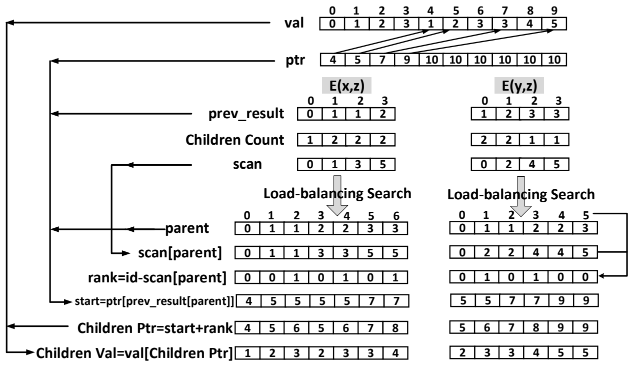


Figure 11: Children Expansion of Level z

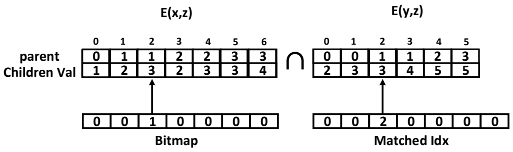


Figure 12: Intersection of Level z

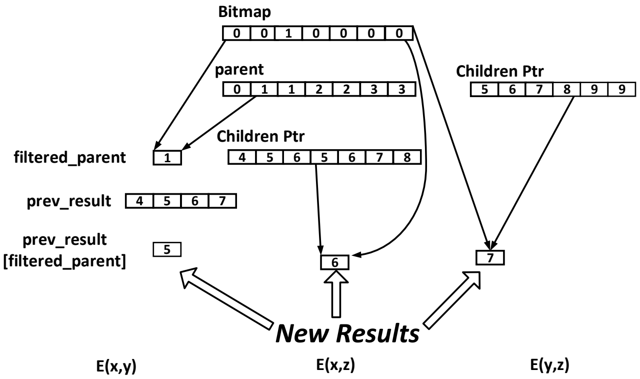


Figure 13: Generating New Results of Level z

The operations performed by the three cases involve several CUDA kernels. Kernels pass data via global memory. Since the algorithm proceeds through levels, the intermediate data size scales with the size of the level. The overall memory footprint should be larger than the *GPU-based* approach which scales with the overall input and output.

4.2 General Join Algorithm

The three levels in the example of finding triangles show how the algorithm handles three different cases. As a general

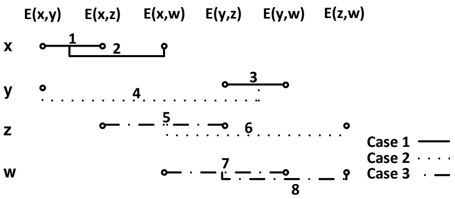


Figure 14: Intersections to solve 4-clique problem. Numbers are the orders to perform the intersection

multi-predicate join, every level may handle more than two predicates. The combination of any two predicates always falls into one of the above three cases. The general algorithm that processes one level is described as below:

1. Classify the predicates according to their status, *processed* or *not processed*.

2. For all the *not processed* predicates, run the pairwise intersection as Case 1 to find common values of all new predicates. The memory accesses are mostly coalesced.

3. For all the *processed* predicates, run the pairwise intersection as Case 3. The intersections can reduce the result size. Some of the memory accesses are index-based indirect access and may not be coalesced.

4. Use the costly binary search as in Case 2 intersect *not processed* and *processed* predicates.

The above algorithms aggressively selects predicates that can be processed by the most efficient intersections and use at most one binary search in one level. Figure 14 shows how the four clique problems can be solved. Three intersections belong to Case 1, another three belong to Case 3, and only two binary searches are used which is much fewer than the *LFTJ-GPU* approach. The pairwise intersection happens in the same level are not independent. Some intermediate results of the first intersection can be reused by the later intersections. For example, the later intersection does not need to expand nodes that has already expanded by the first intersection. Instead, it only needs to filter the existing expanded children.

Overall, GPU-Optimized translates the multi-predicate join problem into a sequence of pairwise intersections and further into three sub problems: node expansion, intersection between two sorted arrays or one sorted and one partially sorted, filtering. Node expansion and intersection are two nontrivial research problems in GPGPU and they still have a large design space to explore in the future.

GPU-Optimized follows the same programming style as ModernGPU and uses the same optimization techniques such as register blocking, texture memory, etc. Moreover, kernels that use load-balancing searches use merge-path to map array elements to GPU threads as many ModernGPU primitives do. Other kernels such as filtering kernels, array elements are evenly assigned to CTAs and then to threads.

In conclusion, compared with the *LFTJ-GPU* approach, *GPU-optimized* approach i) employs a more sophisticated algorithm to enable higher parallelism, better load balance, and more coalesced memory accesses, ii) produces larger memory footprint which restricts the problem size that can be fed into the system. Compared with other GPU algorithms designed for pairwise relational join, *GPU-optimized* approach i) replaces heavy data reorganization (sorting [2] or rebuilding hash table [12] when join keys are changed) with binary searches and ii) has relatively smaller memory footprint because of the uniqueness of the data structure. The goal of GPU-optimized is to pursue the performance even at the cost of smaller limitation of problem size as long as the runnable size is still reasonable. GPU alone cannot run large data set anyway and it has to rely on a CPU algorithm/runtime to run out-of-core data. The experiments in section 5 will discuss the tradeoffs with the performance number.

CPU	Intel i7-4771	GPU	GeForce GTX Titan
G++	4.6.3	NVCC	6.0
OS	Ubuntu 12.04	Driver	331.62
CPU Mem	32GB	GPU Mem	6GB (288.4GB/s)
PCIe	3.0 x16		

Table 1: Experimental Environment.

5. EXPERIMENTAL EVALUATION

We experimentally evaluate the performance of four approaches: LFTJ-CPU, LFTJ-GPU, GPU-optimized and regular pairwise relational joins. We here focus on the two queries that find triangles and four-cliques in randomly generated graphs. Adapting to a larger set of queries and input datasets is part of our future work. We use Red Fox [27], a compiler and runtime system that can evaluate relational queries on GPUs, to perform regular sort-merge joins. The joins performed by Red Fox uses subroutines from the ModernGPU library (as does GPU-optimized). The inputs to Red Fox are flat arrays and the other three approaches use the trie data structure. The results of all different approaches are verified with the LogicBlox platform.

Table 1 lists the characteristics of our evaluation environment. The high-end NVIDIA GeForce GPU is attached as a device on the host PCIe channel.

Graph sizes range from 10K edges to 100M edges; Edges are randomly placed between nodes. Nodes are stored as 64-bit integers. The number of nodes is numEdges$^{(C-1)/(2C-4)}$, where C is 3 for triangles or 4 for 4-cliques. So, each node in triangle problem has 2 edges on average and 4-clique has $2\times$ numEdges$^{1/4}$ edges. Note how nodes for the 4-clique dataset have increasing connected edges when the size of the graph is growing. This will stress the memory footprint of GPU-optimized approach and Red Fox. The degree-distribution is chosen that the number of found triangles or 4-cliques is very sparse, ranging from 0 to 3.

5.1 Overall Performance

In the experiments, both approaches assumes the input data are transformed into the CSR data structures and resided in the GPU memory. PCIe transfer time is not included in the reported result since the paper focuses on in-core algorithm. When extending the algorithms to support out-of-core or even a complete database system, many optimizations such as pipeline execution needs to be applied to reduce PCIe overhead.

Figure 15(a) shows the performance of finding triangles. Limited by GPU memory capacity, GPU-optimized and Red Fox can run up to 30M edges. For larger datasets, these two approaches require space larger than the available GPU memory (6GB) to store the intermediate data. For problem sizes up to 30M edges, GPU-optimized is faster than the other three approaches which demonstrates that all optimizations work well. Compared with LFTJ-GPU, GPU-optimized uses the similar data structure but can more efficiently utilize GPU resources. Compared with Red Fox, GPU-optimized works on a more compact data structure and uses binary searches to replace sorting. Averaging from 10K to 30M, GPU-optimized is 1.72x faster than LFTJ-GPU, 5.23x faster than LFTJ-CPU, and 2.75x faster than Red Fox. It is also interesting to notice that LFTJ-GPU is 3.12x faster than LFTJ-CPU averaging from 10K to 100M although they are originated from the same algorithm; which demonstrates raw GPU performance.

Figure 15(b) compares the performance for finding 4-cliques. Recall that 4-clique involves much more computation than 3-clique so that the achieved throughput for all four approaches is much lower than triangle problem. Here, the advantage of GPU-optimized is more evident. GPU-optimized can run up to 3M edges and the limit of Red Fox is 1M. Averaging from 10K to 3M, GPU-optimized is 5.35x faster than LFTJ-GPU, 2.36x faster than LFTJ-CPU, and 4.13x faster than Red Fox (10K to 300K).

As to LFTJ-GPU, it is slower than LFTJ-CPU when the problem size is small because GPU resources might not be fully utilized. For edges number larger than 1M, LFTJ-GPU surpasses its CPU counterpart since more data need to be processed. LFTJ-GPU is 1.36x faster averaging from 10K to 100M, and 2.14x faster averaging from 1M to 100M.

Most effort of GPU-Optimized is spent on improving memory access pattern and load balance between parallel threads. To verify the effectiveness of our approaches, we use *nvprof* to measure the *warp execution efficiency* and *ld/st average replay*. For comparison, Wang et al. [24] measured the above metrics for a wide range of irregular GPGPU applications including pairwise binary search based relational join. In our 30M triangle problems, *warp execution efficiency* across all kernels are as high as 95% (100% is ideal) which is as high as many regular GPGPU applications. As to *ld/st average replay* (0 is ideal), binary search kernel is 31 and node expansion kernel is 19 as expected. Excluding these kernels, the average number across all kernels is down to 0.96 from 4.62. 4-cliques problem has similar *warp execution efficiency* metrics (94% overall for 3M), but much better *ld/st average replay* number (0.63). The reason is that it only uses 2 binary searches and its node degree is much larger which is good for coalesced memory access pattern.

The peak throughput of GPU-Optimized in triangle example is around 120 MEdges/sec which equals to 3.75 GB/s input/output throughput. Similarly, the peak of 4-cliques is around 0.6 GB/s. Both of them are several times smaller than the bandwidth of PCIe-3.0. we are continuing improving the performance of the algorithm to push it to close to or surpass the PCIe bandwidth. Thus, when extending the algorithm to a larger system capable of running out-of-core data, it will only be limited by PCIe.

Comparison with GraphLab. The GraphLab distribution provides a tool for counting triangles. We used this to compare with our triangle performance (on the same hardware). This tool does not use general-purpose join algorithms to compute triangles; instead a a vertex-centric programming model is used where the triangle counting algorithm is written directly in C++. We did not compare 4-cliques performance because we did not have access to a highly tuned 4-cliques implementation for GraphLab.

As with our approaches, we only report the time for counting triangles and not the load-time. For the 30M (100M) datasets, GraphLab took 7.8s (42.0s) while LFTJ-CPU required 2.1s (9.2s). For smaller datasets LFTJ-CPU was even faster, averaging an improvement of 7X for data size from 10K to 100M. To compare relative speeds without multi-core-effects, we also ran both implementations with the configuration to use only a single-thread. Here, the runtime-numbers for the 30M (100M) dataset are: 27.4s (124s) for GraphLab and 12.8s (51.9s) for LFTJ-CPU.

It is important to note that (1) GraphLab allows using cluster-parallelism and will then scale to much larger datasets; and (2) GraphLab does not require the input data to be sorted as our approaches do. However, even with sorting LFTJ-CPU outperforms GraphLab on our single machine. Focusing on the 30M (100M) dataset: using the CUDA thrust library [3], we can sort in 0.64s (2.5s) on the GPU; we have not parallelized/optimized building the TrieArray data structure. Our serial CPU implementation here takes 0.23s (0.76s).

We did not compare with database-management systems such as LogicBlox or PostgreSQL since these systems focus on out-of-core queries and a comparison would thus not be fair; adding out-of-core functionality to our approaches is part of future work.

5.2 Memory Footprint

The memory footprint of LFTJ-GPU scales with the input and output size. Considering the fact that the tested benchmarks have very sparse output, the memory footprint only scales with the input. In addition to the original input data, the memory footprint of GPU-optimized is also determined by the largest input and output size of all the intersections. The input size, which is expanded from certain parent, can be much larger than the total edges of the benchmark because some parents are expanded more than once for the sake of the intersection performance. Take 4-clique as an example, the ratio between the largest size of the expanded children and graph edge number grows from 12.5 (10K edges) to 50 (3M). Case 3 intersection requires expanding both inputs which makes things even worse. As to Red Fox, the memory footprint is caused by the expansion of the relational join especially when nodes have a lot of connected edges like what 4-clique problem has. Figure 15(c) and Figure 15(d) show the memory footprint for triangle and 4-clique problems respectively. The figures proves that LFTJ-GPU is efficient with memory. GPU-optimized uses less memory than Red Fox mostly because the data in the trie are unique.

5.3 Performance Breakdown of GPU-optimized

One of the largest difference between GPU-optimized and LFTJ-GPU is that LFTJ-GPU only uses binary searches but GPU-optimized uses three different intersections. If only using binary searches, the GPU-optimized would be very similar as LFTJ-GPU because they have the same amount of searches to perform considering the fact they use the similar data structure and same configuration. To better understand the performance of GPU-optimized, Figure 16 shows the performance breakdown of GPU-optimized for 1M triangle and 1M 4-clique. It is also compared with the breakdown of Red Fox to provide a better understanding of the tradeoff between binary searches and sorting. Performance of GPU-

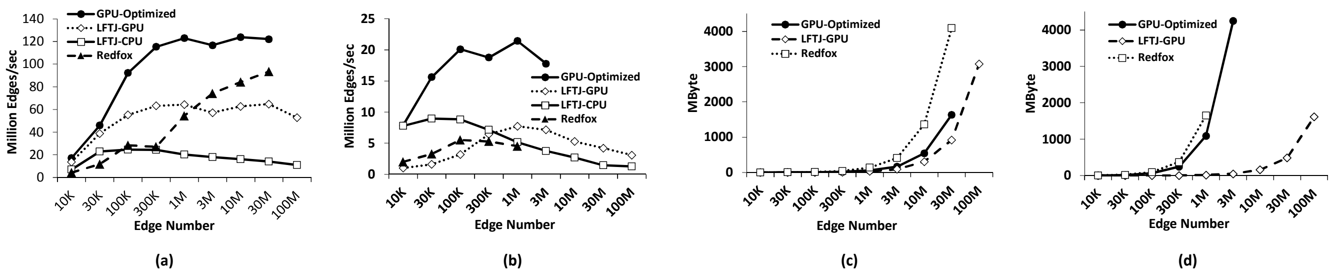


Figure 15: Experimental Results: (a) Triangle Throughput; (b) 4-Clique Throughput; (c) Triangle Memory Footprint; (d) 4-Clique Memory Footprint.

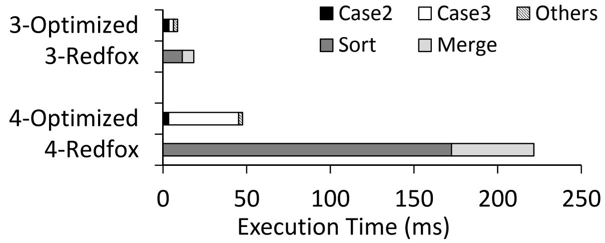


Figure 16: Performance Breakdown of GPU-optimized and Red Fox.

optimized is broken into time spent in different cases (Others include time spent in Case 1 and condition checks such as x<y<z). Meanwhile, sort-merge join performed by Red Fox is split into two parts, sort and merge. GPU-optimized and Red Fox use different algorithm and different data structure. Components belonging to different algorithms cannot be compared head to head. But they are still related since they are solving the same problem.

In Figure 16, the top two bars belong to triangle problem. As to GPU-optimized, the first level only uses Case 1, second level only uses Case 2, and the third level only uses Case 3. Levels are independent of each other. The computation of the first level, Case 1, is negligible. Time spent in Case 2 is 4x of what Case 3 takes. but the input size of Case 2 is just 2.5x larger than Case 3. In Case 3, time spent in expansion, intersection, and filtering is 61%, 31%, and 8% respectively. Expansion has lots of indirect memory accesses and the actual time spent in intersection is relatively small. Without expansion, this efficient intersection operation could not be used and the intersection would be mixed and overwhelmed by the irregular memory access pattern. As to Red Fox, it uses two merges and one sorting to find triangles. Its performance is dominated by the sorting which is even longer than the time spent by the entire GPU-optimized. In functionality, this sorting and its following merge is equivalent to the Case 2 intersection performed by GPU-optimized because they both connect two edges together. Although binary search and sorting are both intense workload in GPU, binary search here is much more efficient.

The bottom two bars in Figure 16 compares the 4-clique benchmark. The sorting used by Red Fox is again more time-consuming than any other components. As to GPU-optimized, time spent on Case 3 is larger than Case 2 because i) the algorithm largely avoid Case 2 and uses Case 3 instead; ii) Case 2 is used at the end of processing a level and its input is already filtered by the previous Case 3 if existing; iii) previous Case 3 expands the nodes for Case 2. Actually, the time spent on the first Case 2 is almost equal to the binary search time in the triangle example because they perform on the similar amount of data and the second Case 2 is negligible because the input size is very small. Performance breakdown and larger speedup against GPU-LFTJ in 4-clique prove that the goal to reduce binary searches to improve performance is achieved.

6. RELATED WORK

Previous GPU-related database research focuses on RA primitive algorithm design. He et al. [9] designed a series of join algorithms including sort-merge join, nested-loop join and hash join which achieved 2-7x speedup over their CPU baseline. Later, Trancoso et al. [21] also implemented nested-loop join and hash join in GPU. They reported an up to 21x speedup compared to a single core system. Diamos et al. [6] designed all RA algorithms for GPUs. For simple operations such as selection, they can achieve the practical maximum memory bandwidth. Their sort-merge join implementation is also based on binary searches which achieved half of peak memory bandwidth. Baxter [2] designed the ModernGPU library including the sort-merge join algorithms. Both the sort and the merge parts were based on the merge path framework [7]. His join implementation reached input/output throughput of 35GB/s for 64-bit random keys in a Titan GPU. All the above algorithms assume the data fit and reside in the GPU memory.

Kaldewey et al. [12] designed a hash join algorithm by using CUDA UVA to support out-of-core data sets. Their measurements showed that the performance of the algorithm was close to the PCIe bandwidth limit. He et al. [10] implemented a Hash Join algorithm in OpenCL that can utilize fused architectures and perform the computation on both CPU and GPU. Their experiments performed on a AMD APU showed that the implementation was 1.53x or 1.35x faster than computing on CPU or GPU only.

Wu et al. [26, 28] designed a framework, Kernel Weaver, as a module of Red Fox that can automatically fuse RA primitives such as several joins together. Using their framework, fusing two pairwise joins is 1.42x faster than running joins separately, which is worse than the GPU-optimized algorithm presented in this paper. Similar as Red Fox, there is other system level research that attempts to evaluate full database queries on GPUs [8, 1, 29, 25, 16, 19].

There is a wealth of work on how to efficiently count or list triangles (as a survey, see [14]) in main memory. While it would be interesting to compare these algorithms with ours in terms of achieved performance, our focus is on general relational processing.

Recent work in the database community has been studying the graph triangulation problem when graphs do not fit into main memory [11, 5, 13, 18]. While out-of-core processing was not the focus of this paper, it would be interesting to investigate the proposed triangle-specific methods for their applicability to (our) general join algorithms. There also has been work [20] that uses general join algorithms to solve graph problems; which, however does not focus on GPUs.

7. CONCLUSION

We have presented two multi-predicate join algorithms for the GPU. While the first is an efficient implementation of its CPU counterpart, the second is a novel algorithm designed for the GPU. We investigated their performance and memory footprint by using them to find triangles and 4-cliques in undirected graphs on synthetic datasets. Our benchmarks show that the GPU-Optimized algorithm outperforms our optimized LFTJ implementation when run on the GPU, which in turn outperforms the same implementation run on the CPU, which outperforms GraphLab on the same hardware. However, the LFTJ has the advantage that its memory requirement is essentially the sum of its input and output data, while the GPU-optimized approach uses potentially large intermediary scratch data.

8. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grant CCF 1337177, by an NVIDIA Graduate Fellowship, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). We would also like to acknowledge the detailed and constructive comments of the reviewers.

9. REFERENCES

[1] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. GPGPU 2010, p.94–103.

[2] S. Baxter. Modern gpu. `http://nvlabs.github.io/moderngpu/`, 2013.

[3] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems, Jade Edition*, pages 359–372, 2011.

[4] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide simd many-core architectures. HPG '09, pages 159–166, 2009.

[5] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *SIGKDD*, pages 672–680, 2011.

[6] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. PPoPP '13, pages 301–302, 2013.

[7] O. Green, R. McColl, and D. A. Bader. Gpu merge path: A gpu merging algorithm. ICS '12, pages 331–340, 2012.

[8] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[9] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. SIGMOD '08, pages 511–524, 2008.

[10] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[11] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. SIGMOD'13, pages 325–336, 2013.

[12] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. DaMoN '12, pages 55–62, 2012.

[13] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu. Opt: a new framework for overlapped and parallel triangulation in large-scale graphs. SIGMOD'14, pages 637–648, 2014.

[14] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1):458–473, 2008.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[16] C. A. Martınez-Angeles, I. Dutra, V. S. Costa, and J. Buenabad-Chávez. A datalog engine for gpus. *Kiel Declarative Programming Days*, pages 239–253, 2013.

[17] NVIDIA. Cuda c programming guide, 2014.

[18] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *PODS*, pages 224–233, 2014.

[19] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber. Multi-level parallel query execution framework for cpu and gpu. In *Advances in Databases and Information Systems*, volume 8133 of *LNCS*, pages 330–343. 2013.

[20] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed SociaLite: A datalog-based language for large-scale graph analysis. *VLDB*, pages 1906–1917, 2014.

[21] P. Trancoso, D. Othonos, and A. Artemiou. Data parallel acceleration of decision support queries using cell/be and gpus. CF '09, pages 117–126, 2009.

[22] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[23] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. ICDT'14, pages 96–106, 2014.

[24] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *IISWC-2014*, October 2014.

[25] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. 2014.

[26] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. MICRO, p.107–118, 2012.

[27] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. CGO '14, pages 44:44–44:54, 2014.

[28] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. IPDPSW '12, pages 2433–2442, 2012.

[29] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.