Graphulo Implementation of Server-Side Sparse Matrix Multiply in the Accumulo Database

Dylan Hutchison,[1] Jeremy Kepner,[1,2,3] Vijay Gadepally,[1,2] Adam Fuchs[4]

[1]MIT Lincoln Laboratory, [2]MIT Computer Science & AI Laboratory,
[3]MIT Mathematics Department, [4]Sqrrl, Inc.

***Abstract*—The Apache Accumulo database excels at distributed storage and indexing and is ideally suited for storing graph data. Many big data graph analytics compute on graph data and persist the results back to the database. These graph calculations are often best performed inside the database server. The GraphBLAS standard provides a compact and efficient basis for a wide range of graph applications through a small number of sparse matrix operations. In this article, We implement GraphBLAS sparse matrix multiplication server-side by leveraging Accumulo's native, high-performance iterators. We compare the mathematics and performance of inner and outer product implementations, and show how an outer product implementation achieves optimal performance near Accumulo's peak write rate. We offer our work as a core component to the Graphulo library that will deliver matrix math primitives for graph analytics within Accumulo.**

I. Introduction

NoSQL databases such as Apache Accumulo concentrate on high performance ingest and scans [1]. While fast ingest and scans solve some big data problems, more complex scenarios involve performing tasks such as data enrichment, graph algorithms, and clustering analytics. These techniques often move data from a database to a compute node. The ability to compute directly in a database can lead to benefits including *selective access*, *data locality*, and *infrastructure reuse*.

Apache Accumulo is designed to deliver fast answers to queries for data subsets along indexed attributes. The Accumulo server processes typically run on the physical nodes where data is stored and cached. Performing computations inside Accumulo can avoid unnecessary network transfer, effectively moving "compute to data" in contrast to client-server models that move "data to compute." Performing computations in the Accumulo server also reuses its distributed infrastructure such as write-ahead logging, fault-tolerant execution, and parallel load balancing of data.

There are a variety of ways to store graphs in Accumulo. One common schema is to store graphs as sparse matrices. Researchers in the GraphBLAS forum [2] have identified a small set of kernels that form a basis for matrix algorithms useful for graphs when represented as sparse matrices. This article presents Graphulo, an effort to realize the GraphBLAS primitives that enable algorithms using matrix mathematics directly in Accumulo servers [3].

In this paper we focus on Sparse Generalized Matrix Multiply (SpGEMM), the core kernel at the heart of GraphBLAS. Many GraphBLAS primitives can be expressed in terms of SpGEMM via the GraphBLAS user-defined element-wise multiplication and addition functions. SpGEMM can be used to implement a wide range of algorithms from graph search [4] to table joins [5] and many others (see introduction of [6]).

We call our implementation of SpGEMM in Accumulo TableMult, short for multiplication of Accumulo tables. Accumulo tables have many similarities to sparse matrices, though a more precise mathematical definition is Associative Arrays [7]. For this work, we concentrate on large distributed tables that may not fit in memory and use a streaming approach that leverages Accumulo's builtin distributed infrastructure.

We are particularly interested in Graphulo for queued analytics, that is, analytics on selected table subsets. Queued analytics maximally leverage databases by quickly accessing subsets of interest, whereas whole-table analytics may perform better on parallel file systems such as Lustre or Hadoop. We therefore prioritize smaller problems that require low latency to enable analysts to explore graph data interactively.

We review Accumulo and its model for server-side computation, iterator stacks, in Section I-A. We formally define matrix multiplication and compare inner and outer product methods in Section II-A, settling on outer product for implementing TableMult. We show TableMult's design as Accumulo iterators in Section II-B and test its scalability with Section III's experiments. We discuss related work, design alternatives and optimizations in Section IV and conclude in Section V.

A. Primer: Accumulo and its Iterator Stack

Accumulo stores data in Hadoop RFiles as byte arrays indexed by key using (key, value) pairs called entries. Keys decompose further into 5-tuples consisting of a row, column family, column qualifier, visibility and timestamp. For simplicity, we focus on a 2-tuple key consisting of a row and column qualifier. Entries belong to tables, which Accumulo divides into tablets and assigns to tablet servers. Client applications write new entries via BatchWriters and retrieve entries sequentially via Scanners or in parallel via BatchScanners.

Accumulo's server-side programming model runs an *iterator stack* on tablets in range of a scan. An iterator stack is a set of data streams originating at Accumulo's data sources

Dylan Hutchison is the corresponding author, reachable at dhutchis@mit.edu.

This material is based upon work supported by the National Science Foundation under Grant No. DMS-1312831. Opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

for a specific tablet (Hadoop RFiles and cached in-memory maps), converging together in merge-sorts, flowing through each iterator in the stack and at the end, sending entries to the client. Iterators themselves are Java classes implementing the SortedKeyValueIterator (SKVI) interface.

Developers add custom logic for server-side computation by writing new iterators and plugging them into the iterator stack. In return for fitting their computation in the SKVI paradigm, developers gain distributed parallelism for free as Accumulo runs their iterators on relevant tablets simultaneously.

SKVIs are reminiscent of built-in Java iterators in that they hold state and emit one entry at a time until finished iterating. However, they are more powerful than Java iterators in that they can seek to arbitrary positions in the data stream. They also have two constraints: the end of the iterator stack should emit entries in sorted order, and iterators must not maintain volatile state such as threads, open files or sockets, because Accumulo may destroy, re-create and re-seek an iterator stack between function calls without allowing time to clean up.

Iterators are most commonly used for "reduction" operations that transform or eliminate entries passing through. The Accumulo community generally discourages "generator" iterators that emit new entries not present in data sources because they are easy to misuse and violate SKVI constraints by emitting entries out of order or relying on volatile state. In this work, we suggest a new pattern for iterator usage as a conduit for client write operations that achieves the benefits of generator iterators while avoiding their constraints.

II. TableMult Design

A. Matrix Multiplication

Given matrices $\mathbf{A}$ of size $N \times M$, $\mathbf{B}$ of size $M \times L$, and operations $\oplus$ and $\otimes$ for element-wise addition and multiplication, the matrix product $\mathbf{C} = \mathbf{A} \oplus . \otimes \mathbf{B}$, or more shortly $\mathbf{C} = \mathbf{AB}$, defines entries of result matrix $\mathbf{C}$ as

$$\mathbf{C}(i,j) = \bigoplus_{k=1}^{M} \mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$$

We call intermediary results of $\otimes$ operations *partial products*.

In the case of sparse matrices, we only perform $\oplus$ and $\otimes$ where both operands are nonzero, an optimization stemming from requiring that 0 is an additive identity of $\oplus$ such that $a \oplus 0 = 0 \oplus a = a$, and that 0 is a multiplicative annhilator of $\otimes$ such that $a \otimes 0 = 0 \otimes a = 0$. Sparse arithmetic requires these conditions, because otherwise zero operands could generate nonzero results.

We study two well known patterns for computing matrix multiplication: inner product and outer product [8]. They differ in the order in which they perform the $\otimes$ and $\oplus$ operations. The more common inner product approach runs the following:

for $i = 1 : N$
  for $j = 1 : L$
    emit $\mathbf{A}(i,:)\mathbf{B}(:,j)$

performing inner product on vectors. For easier comparison, we rewrite the above approach with summation deferred as:

for $i = 1 : N$
  for $j = 1 : L$
    for $k = 1 : M$
      emit $\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$

Inner product has the advantage of generating entries in sorted order: the third-level loop generates all partial products needed to compute a particular element $\mathbf{C}(i,j)$ consecutively. The $\oplus$ applies immediately after each third-level loop to obtain an element in $\mathbf{C}$. The inner-product is easy to "pre-sum," an Accumulo term for applying a Combiner locally before sending entries to a remote but globally-aware table combiner. Emitting sorted entries also permits inner product use in standard iterator stacks.

Despite inner product's order-preserving advantages, outer product performs better for sparse matrices because it passes through $\mathbf{A}$ and $\mathbf{B}$ only once [9] [10]. Inner product's second-level loop repeats scans over all of $\mathbf{B}$ for each row of $\mathbf{A}$. Under our assumption that we cannot fit $\mathbf{B}$ entirely in memory, multiple passes over $\mathbf{B}$ translate to multiple Accumulo scans that each require a disk read. We found in performance tests that an outer-product approach performed an order of magnitude better than an inner-product approach.

The outer product approach runs the following:

for $k = 1 : M$
  emit $\mathbf{A}(:,k)\mathbf{B}(k,:)$

performing outer product on vectors, which unfolds as:

for $k = 1 : M$
  for $i = 1 : N$
    for $j = 1 : L$
      emit $\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$

Outer product emits partial products in unsorted order. This is due to moving the i and j loops that determine partial product position below the top-level k loop.

On the other hand, outer product only requires a single pass over both input matrices. This is because the top-level k loop fixes a dimension of both $\mathbf{A}$ and $\mathbf{B}$. Once we finish processing a column of $\mathbf{A}$ and row of $\mathbf{B}$, we never need read them again.

In terms of memory usage, outer product works best when either the matching row or column fits in memory. If neither fits, then we could run the algorithm with a "no memory assumption" streaming approach by re-reading $\mathbf{B}$'s rows while streaming through $\mathbf{A}$'s columns (or vice versa by symmetry of i and j), perhaps at the cost of extra disk reads.

Because k runs along $\mathbf{A}$'s second dimension and Accumulo uses row-oriented data layouts, we implement TableMult to operate on $\mathbf{A}$'s transpose $\mathbf{A}^\intercal$.

B. TableMult Iterators

We implement SpGEMM with three iterators placed on a BatchScan of table $\mathbf{B}$: RemoteSourceIterator, TwoTableIterator and RemoteWriteIterator. The BatchScanner directs Accumulo to run the iterators on tablets of $\mathbf{B}$ in parallel.

The key idea behind the TableMult iterators is that they divert normal dataflow by opening a BatchWriter, redirecting entries out-of-band to $\mathbf{C}$ via Accumulo's ingest channel that

does not require sorted order. The scan itself emits no entries except for a small number of "monitoring entries" that inform the client about TableMult progress. We enable multi-table iterator dataflow by opening Scanners that read remote Accumulo tables out-of-band. Scanners and BatchWriters are standard tools for Accumulo clients; by creating them inside iterators, we enable client-side processing patterns within tablet servers.

We illustrate TableMult's data flow in Figure 1, placing a Scanner on table $\mathbf{A}^\intercal$ and a BatchWriter on result table $\mathbf{C}$.

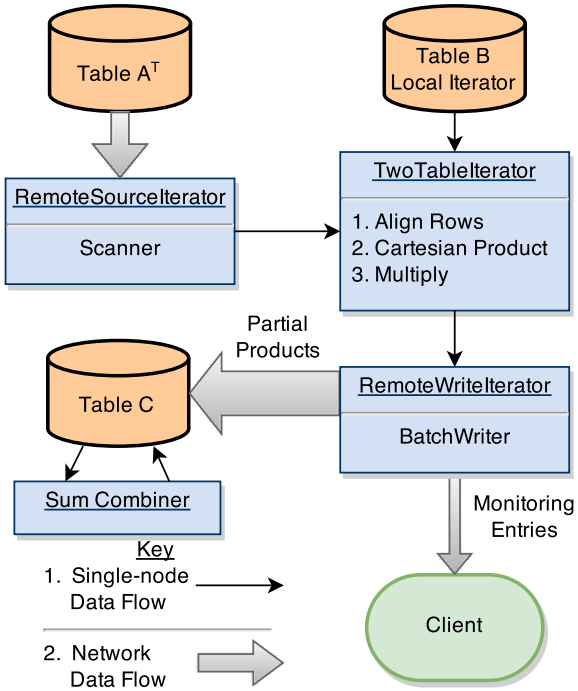

Fig. 1: Data flow through the TableMult iterator stack

1) RemoteSourceIterator: RemoteSourceIterator scans an Accumulo table (not necessarily in the same cluster) using credentials passed from the client through iterator options.

We also use iterator options to specify row and column subsets, encoding them in a string format similar to that in D4M [11]. Row subsets are straightforward since Accumulo uses row-oriented indexing. Column subsets can be implemented with filter iterators but do not improve performance since Accumulo must read every column from disk. We encourage users to maintain a tranpose table using strategies similiar to the D4M Schema [12] for cases requiring column indexing.

Multiplying table subsets is crucial for queued analytics on selected rows. However for simpler performance evaluation, our experiments in Section III multiply whole tables.

2) TwoTableIterator: TwoTableIterator reads from two iterator sources, one for $\mathbf{A}^\intercal$ and one for $\mathbf{B}$, and performs the core operations of the outer product algorithm in three phases:

1) Align Rows. Read entries from $\mathbf{A}^\intercal$ and $\mathbf{B}$ until they advance to a matching row or one runs out of entries. We skip non-matching rows since they would multiply with an all-zero row that, by Section II-A's assumptions, generate all zero patrial products.
2) Cartesian product. Read both matching rows into an in-memory data structure. Initialize an iterator that emits pairs of entries from the rows' Cartesian product.
3) Multiply. Pass pairs of entries to $\otimes$ and emit results.

A client defines $\otimes$ by specifying a class that implements a multiply interface. For our experiments we implement $\otimes$ as java.math.BigDecimal multiplication which guarantees correctness under large or precise real numbers. BigDecimal decoding did not noticeably impact performance.

3) RemoteWriteIterator: RemoteWriteIterator writes entries to a remote Accumulo table using a BatchWriter. Entries do not have to be in sorted order because Accumulo sorts incoming entries as part of its ingest process.

Barring extreme events such as exceptions in the iterator stack or thread death, we designed RemoteWriteIterator to maintain correctness, such that entries generated from its source write to the remote table once. We accomplish this by performing all BatchWriter operations within a single function call before ceding thread control back to the tablet server.

A performance concern remains when multiplying a subset of the input tables' rows that consists of many disjoint ranges, such as one million "singleton" ranges spanning one row each. It is inefficient to flush the BatchWriter before returning from each seek call, which happens once per disjoint scan range. We accomodate this case by "transferring seek control" from the tablet server to RemoteWriteIterator via the same strategy used in RemoteSourceIterator for seeking within an iterator.

We include an option to BatchWrite $\mathbf{C}$'s transpose $\mathbf{C}^\intercal$ in place of or alongside $\mathbf{C}$. Writing $\mathbf{C}^\intercal$ facilitates chaining TableMults together and maintenance of transpose indexing.

4) Lazy $\oplus$: We lazily sum partial products by placing a Combiner subclass implementing BigDecimal addition on table $\mathbf{C}$ at scan, minor and major compaction scopes. Thus, $\oplus$ occurs sometime after RemoteWriteIterator writes partial products to $\mathbf{C}$ yet necessarily before entries from $\mathbf{C}$ may be seen such that we always achieve correctness. Summation could happen when Accumulo flushes table $\mathbf{C}$'s entries cached in memory to a new RFile, when Accumulo compacts RFiles together, or when a client scans $\mathbf{C}$.

The key algebraic requirement for implementing $\oplus$ inside a Combiner is that $\oplus$ must be associative and commutative. These properties allow us to apply $\oplus$ to subsets of a result element's partial products and to any ordering of them, which is chaotic by outer product's nature. If we truly had an $\oplus$ operation that required seeing all partial products at once, we would have to either gather partial products at the client or initiate a full major compaction.

5) Monitoring: RemoteWriteIterator never emits entries to the client by default. One downside of this approach is that clients cannot precisely track progress of a TableMult operation, which may frustrate users expecting a more interactive computing experience. Clients could query the Accumulo monitor for read/write rates or prematurely scan partial products written to $\mathbf{C}$, but both approaches are too coarse.

We therefore implement a monitoring option that emits a value containing the number of entries TwoTableIterator

processed at a client-set frequency. RemoteWriteIterator emits monitoring entries at "safe" points, that is, points at which we can recover the iterator stack's state if Accumulo destroys, re-creates and re-seeks it. Stopping after emitting the last value in the outer product of two rows is safe because we place the last value's row key in the monitoring key and know, in the event of an iterator stack rebuild, to proceed to the next matching row. We may succeed in stopping during an outer product by encoding more information in the monitoring key.

III. Performance

We evaluate TableMult with two variants of an experiment. First we measure the rate of computation as problem size increases. We define problem size as number of rows in random input graphs represented as adjacency tables and rate of computation as number of partial products processed per second. Second we repeat the experiment for a fixed size problem with all tables split into two tablets, allowing Accumulo to scan and write to them in parallel.

We compare Graphulo TableMult performance to D4M as a baseline because a user with one client machine's best alternative is reading input graphs from Accumulo, multiplying them at the client, and inserting the result back into Accumulo.

D4M stores tables as Associative Array objects in Matlab. Because Assoc Array multplication runs fast in memory, D4M bottlenecks on reading data from Accumulo and especially on writing back results. We consequently expect TableMult to outperform D4M because TableMult avoids transferring data out of Accumulo for processing.

We also expect TableMult to succeed on larger graph sizes than D4M because TableMult uses a streaming outer product algorithm that does not store input tables in memory. An alternative D4M implementation would mirror TableMult's streaming outer product algorithm, enabling D4M to run on larger problem sizes at potentially worse performance. We therefore imagine the whole-table D4M algorithm as an upper bound on the best performance achievable when multiplying Accumulo tables outside Accumulo's infrastructure.

We use the Graph500 unpermuted power law graph generator [13] to create random input tables, such that the first row has high degree (number of columns) and subsequent rows exponentially decrease in degree. The generator takes SCALE and EdgesPerVertex parameters, creating graphs with 2^{SCALE} rows and EdgesPerVertex $\times$ 2^{SCALE} entries. We fix EdgesPerVertex to 16 and use SCALE to vary problem size.

The following procedure outlines our performance experiment for a given SCALE and either one or two tablets.

1) Generate two graphs with different random seeds and insert them into Accumulo as adjacency tables via D4M.
2) In the case of two tablets, identify an optimal split point for each input graph and set the input graphs' table splits equal to that point. "Optimal" here means a split point that evenly divides an input graph into two tablets.
3) Create an empty output table. For two tablets, pre-split it with an optimal input split position recorded from a previous multiplication run.
4) Compact the input and output tables so that Accumulo redistributes the tables' entries into the assigned tablets.
5) Run and time Graphulo TableMult multiplying the transpose of the first input table with the second.
6) Create, pre-split and compact a new result table for the D4M comparison as in step 3 and 4.
7) Run and time the D4M equivalent of TableMult:
   a) Scan both input tables into D4M Associative Array objects in Matlab memory.
   b) Convert the string values from Accumulo into numeric values for each Assoc.
   c) Multiply the transpose of the first Assoc with the second Assoc.
   d) Convert the result Assoc back to String values and insert them into Accumulo.

We conduct the experiments on a laptop with 16GB RAM and two dual-core Intel i7 processors running Ubuntu 14.04 linux. We use single-instance Accumulo 1.6.1, Hadoop 2.6.0 and ZooKeeper 3.4.6. We allocate 2GB of memory to the Accumulo tablet server initially (allowing growth in 500MB steps), 1GB for native in-memory maps and 256MB for data and index caches.

We chose not to use more than two tablets per table because more threads would run than the laptop could handle. Each additional tablet can potentially add the following threads:

1) Table $\mathbf{A}^\intercal$ server-side scan thread
2) Table $\mathbf{A}^\intercal$ client-side scan thread,
   running from RemoteSourceIterator
3) Table $\mathbf{B}$ server-side scan/multiply thread,
   running a TableMult iterator stack
4) Table $\mathbf{B}$ client-side scan thread,
   running from the initiating client; mostly idle
5) Table $\mathbf{C}$ server-side write thread
6) Table $\mathbf{C}$ client-side write thread,
   running from RemoteWriteIterator
7) Table $\mathbf{C}$ server-side minor compaction threads,
   running with a Combiner implementing $\oplus$

We show table $\mathbf{C}$ sizes and experiment timings in Table I and plot them in Figure 2. We could not run the D4M comparison past SCALE 15 because $\mathbf{C}$ does not fit in memory.

For the scaled problem, the best results we could achieve are flat horizontal lines, indicating that we maintain the same level of operations per second as problem size increases.

One reason we see a downward rate trend at larger problem sizes is that Accumulo needs to minor compact table $\mathbf{C}$ in the middle of a TableMult. The compactions trigger flushes to disk along with the $\oplus$ Combiner that sums partial products written to $\mathbf{C}$ so far, neither of which we include in rate measurements.

For the fixed size problem, the best results we could achieve are two-tablet rates double the one-tablet rates for every problem size. Our experiment shows that Graphulo two-tablet multiply rates perform up to 1.5x better than one-tablet rates with degraded performance at higher SCALEs. We attribute TableMult's shortfall to high processor contention for the four cores as a result of the 14 threads that may

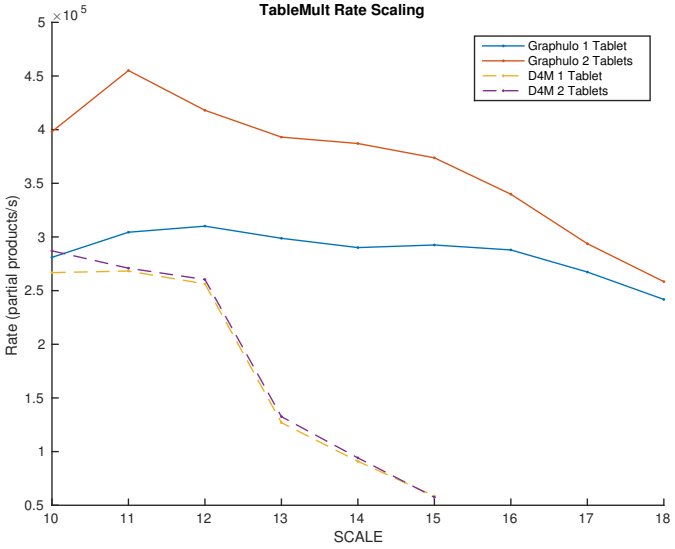

Fig. 2: TableMult Processing Rate vs. Input Table Size

run concurrently with two tablets on; in fact, processor usage hovered near 100% for all four laptop cores throughout the two-tablet experiments. We expect better scaling when running our experiment in a larger Accumulo cluster that can handle more degrees of parallelism.

IV. Discussion

A. Related Work

Buluç and Gilbert studied message passing algorithms for SpGEMM such as Sparse SUMMA, most of which use 2D block decompositions [14]. Unfortunately, 2D decompositions are difficult in Accumulo and message passing even more so. In this work, we use Accumulo's native 1D decomposition along rows and do not rely on tablet server communication apart from shuffling partial products of $\mathbf{C}$ via BatchWriters.

Our outer product method could have been implemented in MapReduce on Hadoop or its successor YARN [15]. There is a natural analogy from TableMult to MapReduce: the map phase scans rows from $\mathbf{A}^\intercal$ and $\mathbf{B}$ and generates a list of partial products from TwoTableIterator; the shuffle phase sends partial products to correct tablets of $\mathbf{C}$ via BatchWriters; the reduce phase sums partial products using Combiners. Examining the conditions on which MapReduce reading from and writing to Accumulo's RFiles directly can outperform Accumulo-only solutions is worthy future work.

A common Accumulo pattern is to scan and write from multiple clients in parallel; in fact, researchers obtained considerably high insert rates using parallel client strategies [16]. We chose to build Graphulo as a service within Accumulo instead of assuming a multiple client capability, such that Graphulo is as accessible as possible to diverse client environments.

The strategy in [16] also used tablet location information to determine where clients could write locally. Knowing tablet-to-tablet-server assignment could likewise aid Graphulo, not only to minimize network traffic but also to partly eliminate Apache Thrift RPC serialization, which prior work has shown is a bottleneck for scans when iterator processing is light [17]. Such an enhancement would access a local tablet server by a method call in place of Scanners and BatchWriters.

B. Design Alternative: Inner-Outer Product Hybrid

It is worth reconsidering the inner product method from our initial design because it has an opposite performance profile as Figure 3's left and right depict: inner product bottlenecks on scanning whereas outer product bottlenecks on writing. At the expense of multiple passes over input matrices, inner product emits partial products in order and immediately pre-summable, reducing the number of entries written to Accumulo to the minimum possible. Outer product reads inputs in a single pass but emits entries out of order and has little chance to pre-sum, instead writing individual partial products to $\mathbf{C}$. Table I quantifies that outer product writes 2.5 to 3 times more entries than inner product for power law inputs. In the worst case, multiplying a fully dense $N \times M$ with an $M \times L$ matrix, outer product emits m times more entries than inner product.

Is it possible to blend inner and outer product SpGEMM methods, choosing a middle point in Figure 3 with equal read and write bottlenecks for overall greater performance? In the following generalization, partition parameter P varies behavior between inner product at $P = N$ and outer product at $P = 1$:

for $p = 1 : P$
  for $k = 1 : M$
    for $i = \left(\left\lfloor \frac{(p-1)N}{P} \right\rfloor + 1 \right) : \left\lfloor \frac{pN}{P} \right\rfloor$
      for $j = 1 : L$
        emit $\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$

The hybrid algorithm runs P passes through $\mathbf{B}$, each of which has write locality to a vertical partition of $\mathbf{C}$ of size $N/P \times L$. Pre-summing ability likewise varies inversely with P, though actual pre-summing depends on $\mathbf{A}$ and $\mathbf{B}$'s sparsity distribution as well as how many positions of $\mathbf{C}$ the TableMult iterators cache. Figure 3's center depicts the $P = 2$ case.

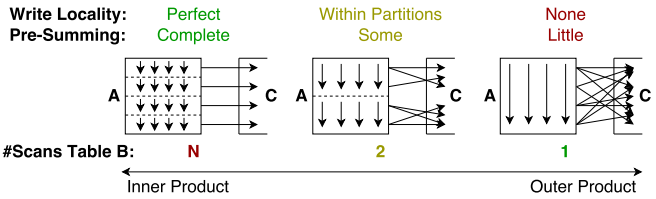

Fig. 3: Tradeoffs between Inner and Outer Product

A challenge for any hybrid algorithm is mapping it to Accumulo infrastructure. We chose outer product because it more naturally fits Accumulo, using iterators for one-pass streaming computation, BatchWriters to handle unsorted entry emission and Combiners to defer summation. The above hybrid algorithm resembles 2D block decompositions, and so maximizing its performance may be challenging given limited data layout control and unknown data distribution.

TABLE I: Output Table $\mathbf{C}$ Sizes and Experiment Timings

	Entries in Table $\mathbf{C}$		Graphulo 1 Tablet		D4M 1 Tablet		Graphulo 2 Tablets		D4M 2 Tablets	
SCALE	PartialProducts	AfterSum	Time (s)	Rate (pp/s)	Time (s)	Rate (pp/s)	Time (s)	Rate (pp/s)	Time (s)	Rate (pp/s)
10	8.05×10^5	2.69×10^5	2.87	2.81×10^5	3.02	2.67×10^5	2.02	3.98×10^5	2.80	2.87×10^5
11	2.36×10^6	8.15×10^5	7.76	3.04×10^5	8.80	2.68×10^5	5.19	4.55×10^5	8.72	2.71×10^5
12	6.82×10^6	2.43×10^6	2.20×10^1	3.10×10^5	2.66×10^1	2.56×10^5	1.63×10^1	4.18×10^5	2.62×10^1	2.60×10^5
13	1.91×10^7	7.04×10^6	6.40×10^1	2.99×10^5	1.50×10^2	1.27×10^5	4.86×10^1	3.93×10^5	1.44×10^2	1.33×10^5
14	5.27×10^7	2.00×10^7	1.82×10^2	2.90×10^5	5.79×10^2	9.09×10^4	1.36×10^2	3.87×10^5	5.59×10^2	9.42×10^4
15	1.47×10^8	5.83×10^7	5.03×10^2	2.93×10^5	2.51×10^3	5.86×10^4	3.94×10^2	3.74×10^5	2.56×10^3	5.75×10^4
16	4.00×10^8	1.63×10^8	1.39×10^3	2.88×10^5			1.18×10^3	3.40×10^5		
17	1.09×10^9	4.59×10^8	4.06×10^3	2.67×10^5			3.70×10^3	2.94×10^5		
18		1.28×10^9	1.21×10^4	2.42×10^5			1.14×10^4	2.58×10^5		

Nevertheless, possible design criteria are to select a small P to minimize passes through $\mathbf{B}$, while also choosing P large enough so that $\lceil NL/P \rceil$ entries fit in memory (dense matrix worst case), which guarantees complete pre-summing. The latter criterion may be relaxed with decreasing matrix density.

C. TableMult in Algorithms

Several optimization opportunities exist for TableMult as a primitive in larger algorithms. Suppose we have a program $\mathbf{E} = \mathbf{AB}; \mathbf{F} = \mathbf{CD}; \mathbf{G} = \mathbf{EF}$. We would save two round trips to disk if we could mark $\mathbf{E}$ and $\mathbf{F}$ as "temporary tables," i.e. tables intermediate to an algorithm that should be held in memory and not written to Hadoop if possible.

A *pipelining* optimization streams entries from a TableMult to computations taking its result as input. Outer product pipelining is difficult because it cannot guarantee writing every partial product for a particular element to $\mathbf{C}$ until it finishes, whereas inner product's complete pre-summing emits elements ready for downstream use. More ambitiously, *loop fusion* merges iterator stacks for successive computations into one.

Optimizing computation on NoSQL databases is challenging in the general case because NoSQL databases typically avoid query planner features customary of SQL databases in exchange for raw performance. NewSQL databases aim in part to achieve the best of both worlds—performance and query planning [18]. We aspire to make a small step for Accumulo in the direction of NewSQL with current Graphulo research.

V. Conclusions

In this work we showcase the design of TableMult, a Graphulo server-side implementation of the SpGEMM GraphBLAS matrix math kernel in the Accumulo database. We compare inner and outer approaches and show how outer product better fits Accumulo's iterator model. The implementation shows excellent single node performance, achieving rates near 400,000 per second, which is the consistent with the single node peak write rate for Accumulo [16]. Performance experiments show good scaling for scaled problem sizes and suggest good scaling for fixed size problems, but these require addditonal experiments on a larger cluster to confirm.

In addition to topics from Section IV's discussion, future research efforts include implementing the remaining GraphBLAS kernels, developing graph algorithms that use the Graphulo, and delivering to the Accumulo community.

References

[1] R. Sen, A. Farris, and P. Guerra, "Benchmarking apache accumulo bigdata distributed table store using its continuous test suite," in *International Congress on Big Data, 2013 IEEE*. IEEE, 2013, pp. 334–341.

[2] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson *et al.*, "Standards for graph algorithm primitives," *arXiv preprint arXiv:1408.0393*, 2014.

[3] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015 IEEE International*. IEEE, 2015.

[4] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[5] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "Mad skills: new analysis practices for big data," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1481–1492, 2009.

[6] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," 2010.

[7] J. Kepner and V. Gadepally, "Adjacency matrices, incidence matrices, database schemas, and associative arrays," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014.

[8] C. P. Kruskal, L. Rudolph, and M. Snir, "Techniques for parallel manipulation of sparse matrices," *Theoretical Computer Science*, vol. 64, no. 2, pp. 135–157, 1989.

[9] P. Burkhardt and C. Waring, "An nsa big graph experiment," *US National Security Agency Technical report NSA-RD-2013-056001v1*, 2013.

[10] P. Burkhardt, "Asking hard graph questions," *US National Security Agency Technical report NSA-RD-2014-050001v1*, 2014.

[11] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz *et al.*, "Dynamic distributed dimensional data model (d4m) database and computation system," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE, 2012, pp. 5349–5352.

[12] J. Kepner, C. Anderson, W. Arcand, D. Bestor, B. Bergeron, C. Byun, M. Hubbell, P. Michaleas, J. Mullen, D. O'Gwynn *et al.*, "D4m 2.0 schema: A general purpose high performance schema for the accumulo database," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

[13] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *Cyberinfrastructure Technology Watch*, vol. 2, pp. 1–10, 2006.

[14] A. Buluc and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[16] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout *et al.*, "Achieving 100,000,000 database inserts per second using accumulo and d4m," *IEEE High Performance Extreme Computing*, 2014.

[17] S. M. Sawyer, B. D. O'Gwynn, A. Tran, and T. Yu, "Understanding query performance in accumulo," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

[18] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: Nosql and newsql data stores," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, no. 1, p. 22, 2013.