Graphulo: Server-side Matrix Multiply on Accumulo Tables

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

[1]MIT Lincoln Laboratory BeaverWorks
[2]MIT Computer Science and Artificial Intelligence Laboratory
[3]MIT Mathematics Department
[4]Sqrrl

use more specific example

***Abstract*—Server-side computation is difficult in Accumulo due to its design for distributed storage and not for general computing, yet many big data applications such as enrichment and analytics compute on Accumulo data and persist results back to Accumulo anyway. Users must subsequently implement complex clients and shuffle data between Accumulo storage and engines for compute.**

In this work we enable SpGEMM, sparse matrix multiplication, server-side on Accumulo tables by repurposing Accumulo iterators. We compare mathematics and performance of inner and outer product approaches and show how an outer product implementation scales with synthetic experiments. We offer our work as a first step for the Graphulo library that will deliver linear algebra primitives server-side on 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 running tasks such as enrichment, algorithms and analytics. These techniques often move data from a database to a computational element. The ability to compute directly in a database can lead to benefits including *selective access*, *data locality* and *infrastructure reuse*.

Consider the Apache Accumulo database whose features as a database deliver fast answers to data subsets along indexed attributes. Accumulo sits atop the physical location data is stored and cached such that computation inside Accumulo can avoid unnecessary network transfer, effectively moving "compute to data" like a stored procedure in contrast to client-server models moving "data to compute." Computing in Accumulo also reuses its distributed infrastructure such as write-ahead logging, fault-tolerant execution atop Zookeeper and horizontal scaling from a master load balancing tablets.

One family of algorithms commonly applied to large scale data is linear algebra. Researchers in the GraphBLAS Forum [2] have identified a set of kernels that form a basis for linear algebraic algorithms useful for graphs, including sparse general matrix multiplication (SpGEMM), sparse element-wise multiplication (SpEWiseX), sparse subset reference (SpRef), reduction along a dimension (Reduce), function application (Apply) and others. This article presents Graphulo, an effort to realize the GraphBLAS primitives and enable algorithms in the language of linear algebra server-side in Accumulo [3].

In this paper we focus on SpGEMM, a core kernel at the heart of the GraphBLAS. In fact, many other GraphBLAS primitives can be expressed in terms of SpGEMM through custom functions that may redefine multiplication and addition. SpGEMM usage ranges from graph search [4] to table joins [5] and plenty others described in the introduction of [6].

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

We are particularly interested in SpGEMM for queued analytics, that is, analytics on a selected table subset. Queued analytics maximally leverage Accumulo as a database by quickly accessing subsets of interest, whereas whole-table analytics usually perform better on parallel file systems such as Lustre or Hadoop. We therefore prioritize low latency over high throughput, in the best case enabling analysts to manipulate Accumulo 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 SpGEMM methods in Section II-A, ultimately settling on outer product for implementing TableMult. We show TableMult's design as Accumulo iterators in Section II-B and test its scalability with experiments in Section III. We discuss design alternatives and related work in Section IV, concluding in Section V.

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.

A. Primer: Accumulo and the Iterator Stack

Accumulo stores data in Hadoop as byte arrays decomposed into (key, value) pairs called entries. Keys decompose further into rows, column families, qualifiers, visibilities and timestamps, though we mainly consider rows and column qualifiers in this work. Entries belong to tables that Accumulo divides into tablets and assigns to tablet servers. Clients write new entries via BatchWriters and retrieve stored entries from tablets sequentially via Scanners or in parallel via BatchScanners.

Accumulo's server-side programming model runs an *iterator stack* on each tablet in range of a scan, which is a list of classes that implement the `SortedKeyValueIterator` (SKVI) interface. At a high level, an iterator stack is a set of data streams originating at Accumulo's data sources 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, all while maintaining data emission in sorted order.

Developers add custom logic for server-side computation by writing new SKVIs 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 also more powerful than Java iterators in that they can seek to a specific position in the data stream (top-level system iterators perform actual disk seeks).

A special caveat to iterator stacks is that Accumulo may destroy, re-create and re-seek them to their last emitted key between any function call. Accumulo does this when it needs to switch data sources after a compaction, when a client stops requesting data, or out of fairness to concurrent scans. Iterators have no `close` method that would grant them the lifecycle control to clean up state before Accumulo destroys them, and so the only safe way for an iterator to use state requiring cleanup (such as opening a file or starting a thread) is to clean up state before returning from any method call. Ideas discussed in [8] may lax this restriction for future Accumulo versions.

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 original data sources, not because generator iterators are impossible but 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 input matrices $\mathbf{A}$ of size $n \times m$, $\mathbf{B}$ of size $m \times p$, and operations $\oplus$ and $\otimes$ representing summation and multiplication, the matrix multiplication $\mathbf{A} \oplus.\!\otimes \mathbf{B} = \mathbf{C}$, or more shortly $\mathbf{AB} = \mathbf{C}$, 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 $\otimes$ operations where both operands are nonzero, an optimization stemming from the requirement 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 is impossible without these conditions, since in that case zero elements could generate nonzero results.

We study two well known patterns for matrix multiplication, inner and outer product, in terms of how they implement $\otimes$, deferring $\oplus$ to run on output generated from applying $\otimes$. We use Matlab notation in pseudocode for arrays and indexing.

The more common inner product method runs the following:

```
for $i = 1\!:\!n$
    for $j = 1\!:\!p$
        emit $\mathbf{A}(i,:) \cdot \mathbf{B}(:,j)$
```

where the operation $\cdot$ is inner (also called dot) product on vectors, which we may unfold as

```
for $i = 1\!:\!n$
    for $j = 1\!:\!p$
        for $k = 1\!:\!m$
            emit $\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$
```

Inner product has an advantage of generating output "in order," meaning that all partial products needed to compute a particular element $\mathbf{C}(i,j)$ are generated consecutively by the third-level loop. We may apply the $\oplus$ operation immediately after each third-level loop and obtain an element in the result matrix. This means that 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. It is also adventageous that inner product generates entries sorted by row and column, which allows inner product to be used in standard iterator stacks that require sorted output.

Despite its order-preserving advantages, we chose not to implement inner product because it requires multiple passes: the second-level loop that scans over all of input matrix $\mathbf{B}$ repeats for each row of $\mathbf{A}$ from the top-level loop iteration. Under our assumption that we cannot fit $\mathbf{B}$ entirely in memory, multiple passes over $\mathbf{B}$ translates to multiple Accumulo scans that each require a disk read. We found in our performance tests that multiple scans over $\mathbf{B}$ delivered over an order of magnitude worse performance, taking over 100 seconds to multiply SCALE 11 inputs whereas the outer product method ran in under 8 seconds.

Outer product matrix multiply runs the following:

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

```
for $k = 1 : m$
  for $i = 1 : n$
    for $j = 1 : p$
      emit $\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$
```

where the operation $\times$ is outer (also called tensor or Carteisan) product on vectors, which we may unfold as

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 full column of $\mathbf{A}$ and row of $\mathbf{B}$, we never need to read them again (i.e., we never need to restart the top-level k loop).

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 still run the algorithm by re-reading rows of $\mathbf{B}$ while streaming through columns of $\mathbf{A}$ (or vice versa by symmetry of i and k). We may implement the "no memory assumption" streaming approach in Accumulo by using `deepCopy` SKVI methods to store "pointers" to the beginning of rows from table $\mathbf{B}$ (or columns of table $\mathbf{A}$), perhaps at the cost of extra disk reads.

Because k runs along the second dimension of $\mathbf{A}$ and Accumulo uses a row-oriented data layout, 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 a result table via Accumulo's standard ingest channel that does not require sorted order. The scan itself emits no entries except for a smidgeon 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 Accumulo iterators, we enable client-side processing patterns within the Accumulo server.

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

1) RemoteSourceIterator: RemoteSourceIterator scans an Accumulo table (not necessarily in the same cluster) with the same range it is seeked to. Clients pass connection information including zookeeper hosts, timeout, username and password via iterator options. We leave more secure scan authentication

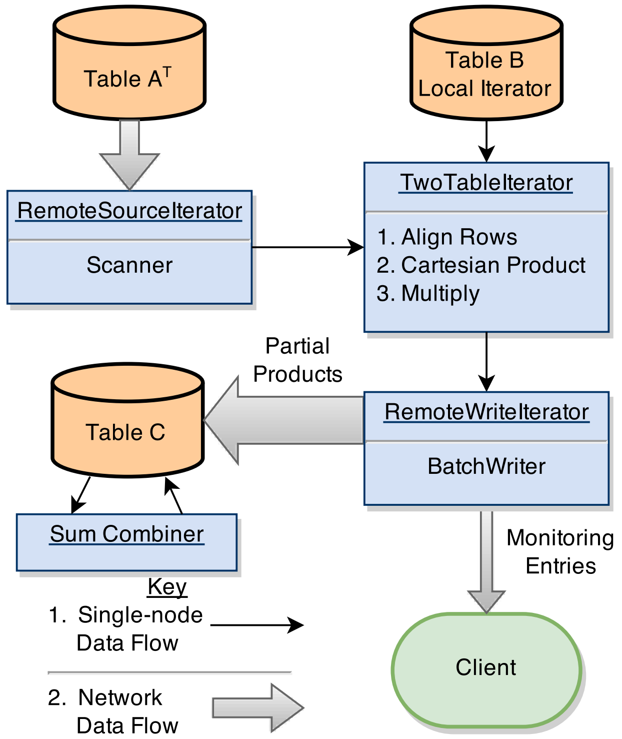

Fig. 1: Data flow through the TableMult iterator stack

methods to future work, although only users with access to the Accumulo instance may see the password in iterator options.

We also use iterator options to specify row and column subsets, encoding them in a string format similar to that in D4M [9]. 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 [10] 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' Carteisan product.
3) Multiply. Pass pairs of entries to $\otimes$ and emit results.

The user defines $\otimes$ by specifying a class that implements a provided "multiply interface" which TwoTableIterator instantiates and calls. 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 standard ingest process. Like RemoteSourceIterator, the client passes connection information for the remote table via iterator options.

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

A performance concern remains in the case of TableMult over a subset of the input tables' rows that consists of many disjoint ranges, such as 1M "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, and a known Accumulo issue could even crash the tablet server [11]. We accomodate this case by "transfering seek control" over the desired row range subset from the Accumulo tablet server to RemoteWriteIterator by passing the range objects through iterator options using the same techniques as RemoteSourceIterator, as opposed to the usual method of passing range objects to the BatchScanner on table $\mathbf{B}$. RemoteWriteIterator then traverses all ranges in the desired subset (within the tablet it runs on) within one call to seek.

We include an option to BatchWrite the result table's transpose in place of or alongside the result table. Writing the transpose facilitates chaining TableMults one after another as well as 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 perform $\oplus$ on subsets of a result element's partial products and on any ordering of them, which is chaotic by the 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 the 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 unhelpfully 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, recreates 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 gauge weak scaling by measuring rate of computation as problem size increases. We define problem size as number of nodes in random input graphs and rate of computation as number of partial products processed per second. Second we gauge strong scaling by repeating the experiment 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, written as Assocs for short. Since Assoc multplication runs fast in memory, D4M bottlenecks on reading data from Accumulo and especially on writing back results. We consequently expect TableMult to run faster than D4M because TableMult avoids transfering data out of Accumulo in order to process it.

We also expect TableMult to succeed on larger graph instances 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 the cost of 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 random graph generator [12] to create random input matrices. The generator creates graphs with a power law structure, such that the first vertex has high degree and subsequent vertices have exponentially decreasing degree. The graph generator takes a SCALE and EdgesPerVertex parameter and creates graphs with 2^{SCALE} vertices and EdgesPerVertex $\times$ 2^{SCALE} edges. 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 random graphs with different random seeds and insert them into Accumulo tables using 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 nearly evenly divides an input graph into two tablets.

3) Create an empty output table and pre-split it with the first input table's split. The split will not be optimal for the output table because the matrix product has a different degree distribution than that of the input tables, but it is close enough for the purposes of our test.
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 it into Accumulo.

We conduct the experiments on a laptop with 16GB RAM and 2 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 it would run more threads 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 a RemoteSourceIterator
3) Table $\mathbf{B}$ server-side scan/multiply thread,
    running the TableMult iterator stack
4) Table $\mathbf{B}$ client-side scan thread,
    running from the client initiating; mostly idle
5) Table $\mathbf{C}$ server-side write thread,
    running with a Combiner implementing $\oplus$
6) Table $\mathbf{C}$ client-side write thread,
    running from a RemoteWriteIterator
7) Table $\mathbf{C}$ server-side minor compaction threads

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.

In terms of weak scaling, 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. Graphulo roughly achieves weak scaling, although the two-tablet Graphulo curve shows some instability.

One reason we see a downward trend in rate at larger problem sizes is that Accumulo needs to minor compact table $\mathbf{C}$ in the middle of a TableMult. This in turn triggers the $\oplus$ Combiner to sum partial products written to table $\mathbf{C}$ so far.

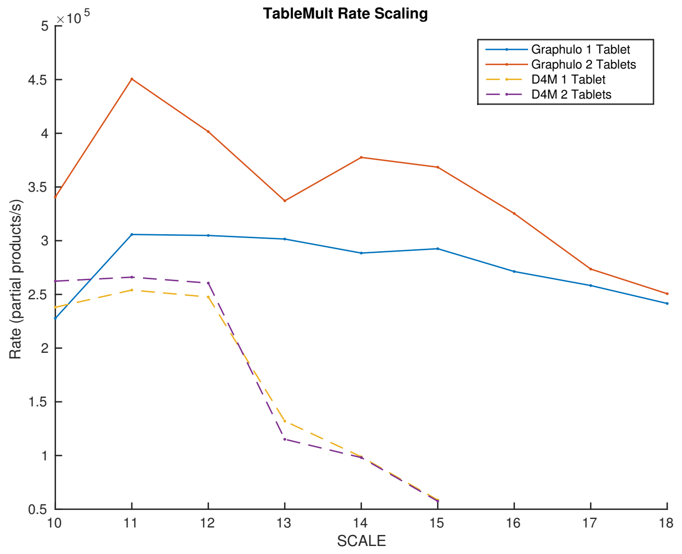

Fig. 2: TableMult processing rate vs. input table size

Thus, one explanation for the rate decrease is that our rate measurements (in partial products per second) do not include summing and disk flush operations Accumulo performs during minor compaction.

In terms of strong scaling, 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 as a result of the 14 threads that may run concurrently with two tablets; in fact, processor usage hovered near 100% for all four laptop cores throughout the two-tablet experiments. We expect better strong scaling results when we run our experiment in a larger Accumulo cluster that can handle more degrees of parallelism.

IV. Discussion

A. *TableMult Design Alternatives*

A common Accumulo pattern is to run multiple clients that scan disjoint and continuous table sections in parallel. We avoid this pattern because it increases client software complexity, whereas we aim to provide a service within Accumulo that works for any client. Perhaps more importantly, previous work has shown that table scans that do not perform significant iterator processing bottleneck on communication overhead *at the client* related to Apache Thrift serialization [13]. We gain a chance to eliminate this overhead by moving computation to the server, though we do not currently do so as we use standard Accumulo Scanners and BatchWriters.

We also find room to reconsider the inner product SpGEMM formulation from our initial design because it has an opposite performance profile as Figure 3 depicts: inner product bottlenecks on scanning whereas outer product bottlenecks on writing. At the expense of multiple passes over input matrices, inner product emits entries more efficiently in that emitted

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	804,989	269,404	3.53	227,700.33	3.38	238,023.69	2.36	340,721.66	3.06	262,328.46
11	2,361,580	814,644	7.72	305,789.27	9.29	254,040.68	5.24	450,554.23	8.87	266,050.05
12	6,816,962	2,430,381	22.36	304,868.98	27.53	247,568.13	16.97	401,529.20	26.16	260,553.36
13	19,111,689	7,037,007	63.37	301,550.36	144.73	132,047.17	56.68	337,136.98	166.01	115,121.57
14	52,656,204	20,029,427	182.52	288,486.51	532.91	98,808.44	139.46	377,562.34	536.79	98,094.46
15	147,104,084	58,288,789	502.86	292,532.77	2,510.38	58,598.13	399.24	368,452.34	2,559.24	57,479.52
16	400,380,031	163,481,262	1,475.81	271,294.29			1,230.81	325,297.94		
17	1,086,789,275	459,198,683	4,208.24	258,252.42			3,972.13	273,603.14		
18	2,937,549,526	1,280,878,452	12,161.74	241,540.15			11,720.44	250,634.66		

entries are in order and partial products can sum immediately, 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 partial products, instead writing individual partial products to the result table. Table I quantifies the additional number of entries outer product writes for power law inputs as 2.5 to 3 times that of inner product. In the worst case, multiplying a fully dense $n \times m$ with an $m \times p$ matrix, outer product would emit m times more entries than inner product.

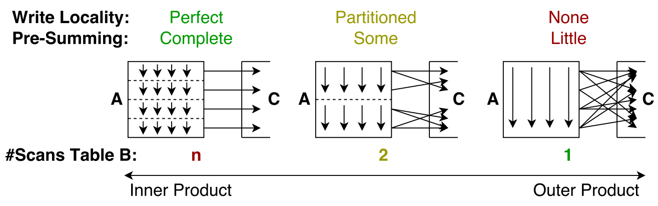

Fig. 3: Tradeoffs between Inner and Outer Product

Is it possible to blend inner and outer product algorithms and achieve better performance than either method alone? A definitive answer is future research, though we sketch one possible bridge in Figure 3. Suppose we partition the rows of $\mathbf{A}$ (columns of $\mathbf{A}^\intercal$) into two halves and run outer product on each separately. Each outer product requires one scan over $\mathbf{B}$ for a total of two passes. In exchange we increase write locality: the top half outer product only writes to the top half of $\mathbf{C}$ and the bottom half outer product only writes to the bottom half of $\mathbf{C}$. Selecting the number of partitions along rows of $\mathbf{A}$ determines balance between inner product (partitions between every row) and outer product (zero partitions).

A challenge for any hybrid algorithm is mapping it to Accumulo infrastructure. We chose outer product because it more naturally fits into Accumulo, using iterators for one-pass streaming computation, BatchWriters to handle unsorted entry emission and Combiners to defer summation. We may realize greater performance by considering data placement among tablet servers, but such a consideration would require accessing and perhaps manipulating Accumulo's internal state and non-public API. We suggest this paper's approach as a balance between top performance and implementation stability.

B. 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 a round trip to disk if we could mark tables $\mathbf{E}$ 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 downstream computations. Outer product pipelining is difficult because it cannot guarantee all partial products for any particular element are written to table $\mathbf{C}$ until it finishes. Inner product's write locality makes it easier to pipeline. More ambitiously, a *loop fusion* optimization merges iterator stacks for two computations into one.

Optimizing computation on NoSQL databases is challenging in the general case because NoSQL databases mostly exclude 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 [14]. We aspire to make a small step for Accumulo in the direction of NewSQL with current Graphulo research.

C. Related Work

Buluç and Gilbert studied message passing algorithms for SpGEMM such as Sparse SUMMA, most of which use 2D block decompositions [15]. 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 no tablet server communication other than shuffling partial products to tablets of $\mathbf{C}$ via BatchWriters.

Our outer product method could have been implemented in MapReduce on Hadoop or its successor YARN [16]. In fact, there is a natural analogy from how we process data in Accumulo 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 the correct tablet of $\mathbf{C}$ via BatchWriters; the reduce phase sums partial products using Accumulo Combiners. Examining the conditions on which MapReduce outperforms an Accumulo-only solution is worthy future work.

V. Conclusions

In this work we showcase the design of TableMult, a Graphulo implementation of the SpGEMM GraphBLAS linear algebra kernal server-side on Accumulo tables. We compare inner and outer approaches for SpGEMM and show how outer product better suits the Accumulo iterator environment. Performance experiments show good weak scaling and hint at strong scaling, although repeating our experiments on a larger cluster is necessary to confirm.

Current research is to implement the remaining GraphBLAS kernels and develop algorithms calling them, ultimately delivering a Graphulo linear algebra library as a pattern for server-side computation 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 *Big Data (BigData Congress), 2013 IEEE International Congress on*. 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] D. Hutchison. (2015) Iterator redesign. [Online]. Available: https://issues.apache.org/jira/browse/ACCUMULO-3751

[9] 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.

[10] 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.

[11] D. Hutchison. (2015) Scanning with many singleton ranges crashes tserver. [Online]. Available: https://issues.apache.org/jira/browse/ACCUMULO-3710

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.