

Low Thread-count Gustavson: A multithreaded algorithm for sparse matrix-matrix multiplication using perfect hashing

James J. Elliott and Christopher M. Siefert
Center for Computing Research
Computer Science Research Institute
Sandia National Laboratories
Albuquerque, NM

Abstract—Sparse matrix-matrix multiplication is a critical kernel for several scientific computing applications, especially the setup phase of algebraic multigrid. The MPI+X programming model, which is growing in popularity, requires that such kernels be implemented in a way that exploits on-node parallelism. We present a single-pass OpenMP variant of Gustavson’s sparse matrix-matrix multiplication algorithm designed for architectures (e.g. CPU or Intel Xeon Phi) with reasonably large memory and modest thread counts (tens of threads, not thousands). These assumptions allow us to exploit perfect hashing and dynamic memory allocation to achieve performance improvements of up to 2x over third-party kernels for matrices derived from algebraic multigrid setup.

I. INTRODUCTION

The solution of large, sparse linear systems is ubiquitous in the solution of partial differential equations (PDEs) using implicit timestepping. The solution of such systems is often done using preconditioned Krylov methods. Not infrequently, algebraic multigrid (AMG) algorithms [1], are used to precondition these Krylov methods, as they are theoretical mathematically scalable (e.g., the solution time scales linearly in the number of unknowns). These algorithms generate a series of coarse grids using sparse general matrix-matrix multiplication (SpGEMM), taking an initial matrix A_1 , a coarse-to-fine prolongator P and a fine-to-coarse restrictor R and forming the coarse matrix $A_2 = RA_1P$.

The formation of the coarse matrix is by far the most time-consuming portion of the setup phase of AMG. As the setup phase needs to be redone when the matrix changes, this calculation is often done at least once per timestep of a transient PDE code, and potentially more often if the PDE is nonlinear. Thus, the efficiency and scalability of the SpGEMM kernel is of great interest to the multigrid community as well as those who are users of multigrid preconditioners to help solve their linear systems.

A substantial trend in current thought on high-performance computing (HPC) is the “MPI+X model,” which posits the use of MPI between nodes on a HPC machine, and some other parallel programming technique on a node. One of the important candidates for “X” on a number of non-GPU current and future HPC architectures is threading via OpenMP [2].

This paper is motivated by the desire to accelerate AMG setup, coupled with the requirement that our kernels be able to use the MPI+X model. The use case we consider is one where multiple MPI processes are used on a physical node, e.g., 4 or 8 per Xeon Phi socket or 1-2 per Haswell socket. This motivates investigating kernels that are performant at low thread counts, e.g. one to eight threads, yet capable of using a modest thread count, e.g., 16, effectively. This is in contrast to kernels designed for very high concurrency, e.g., 272 threads on a Xeon Phi.

We present a new multithreaded algorithm for SpGEMM, that is competitive for general sparse matrix multiplication, as well as highly performant for AMG hierarchy construction. This paper is organized as follows: We introduce SpGEMM and the canonical approach for implementing it. We introduce our algorithm and explain how we obtain performance while using only a single pass through data. Finally, we present a performance study conducted on a Cray XC40 Haswell partition.

We present the following contributions:

- Low Thread-count Gustavson: a single pass, multi-threaded SpGEMM algorithm;
- Strong thread scaling that consistently outperforms two third party kernels for AMG setup.
- A qualitative study showing our algorithm can handle the more general problem of matrix squaring and still be performant.

II. BACKGROUND

This paper presents a multithreaded variant of a classic sparse matrix-matrix multiplication algorithm. To understand our contribution, some knowledge of how sparse matrices are stored and how this storage relates to SpGEMM algorithms is necessary. Following, we present a brief introduction to the CSR format, the canonical SpGEMM algorithm from 1978, how matrix storage impacts SpGEMM algorithm design, and finally how SpGEMM relates to scientific computing.

A. Introduction to SpGEMM

Compressed sparse row (CSR) storage is a common form of storage for sparse matrices. It consists of two integer-valued arrays, *rowptr*, and *colind*, and one real-valued array, *vals*. The *rowptr* array is of length $N + 1$, where N is the number of rows in the matrix. The *colind* and *vals* arrays are of a length equal to the number of nonzeros in the matrix. The entry *rowptr*[i] is the first entry in the *colind* and *vals* arrays which contains a nonzero in row i . The entry *rowptr*[$i + 1$] is one entry past the last entry in the *colind* and *vals* arrays containing nonzeros in row i . This storage format allows row-wise operations to be computed quickly, since rows have contiguous storage. However column-wise operations are very slow. This storage format has a direct impact on the choice of algorithm for SpGEMM, which is why we detail it here.

Serial matrix-matrix multiplication involves three nested loops, which can be ordered any of six different ways. The most common loop ordering for CSR matrices is the *row-wise* ordering, which resolves rows of the result matrix C sequentially through a series of partial updates of the entries in that row. No entry in that row is guaranteed to be correct until all partial updates are finished, in contrast with the textbook *inner-product* ordering, which completes each entry in order. A row-wise algorithm for sparse matrix-matrix multiplication is shown in Algorithm 1 to illustrate the loop ordering.

Algorithm 1: Row-wise sparse matrix-matrix multiplication

```

input :  $A \in \mathbb{R}^{l \times m}$  and  $B \in \mathbb{R}^{m \times n}$ 
output:  $C \in \mathbb{R}^{l \times n}$ 
for  $i = 1, \dots, l$  do
  foreach  $k$  s.t.  $a_{i,k}$  is nonzero do
    foreach  $j$  s.t.  $b_{k,j}$  is nonzero do
      if  $c_{i,j}$  has not been stored yet then
         $c_{i,j} = 0$ ;
      end
       $c_{i,j} += a_{i,k} \cdot b_{k,j}$ ;
    end
  end
end

```

In the SpGEMM algorithm, the structure of the resulting C matrix is not known in advance. One way of handling this is a two-pass algorithm, where the first pass counts the memory required and the second pass computes the value. A second approach relies on getting a decent estimate of the memory use and doing dynamic memory allocation during a single pass. Depending on how this algorithm is implemented, a single pass algorithm could require a final “copy out” phase if some kind of temporary storage is used. This algorithmic choice will be discussed in more detail in Section IV-A.

B. Row-wise SpGEMM Algorithms

Row-wise SpGEMM algorithms require an efficient method for accumulating partially computing entries and allowing new

partial sums to be added. Following the terminology of Deveci et al., we refer to these as accumulators [3]. The three basic categories of accumulator algorithm for row-wise SpGEMM are sort/merge based, heap based hash based, where the use of dense accumulators can be viewed as perfect hashing.

The classic serial SpGEMM is found work of Gustavson [4], which outlines the canonical dense accumulator perfect hashing algorithm. Algorithms in this class use a temporary array of length equal to the number of columns present in the matrix B . This array is used to index into the final result vector. If a new entry is encountered, the temporary array is then set to point to the first vacant entry of the result array, dynamically allocating more memory if needed. If the dynamic memory allocation can be amortized, this allows accumulation to be done in $O(1)$ time per entry. MPI parallel implementations of dense accumulator algorithms can also be found in software packages such as Trilinos [5].

Non-dense hashing algorithms take a different approach, keeping a hash table to store and lookup the partially accumulated entries in the row. If rows are far from dense, this approach of taking substantially less memory than a dense accumulator (e.g. the size of the hash table required is much less than the full number of columns present in the matrix B). Like the dense accumulator case, a properly designed hash table should cost $O(1)$ per entry, however the constant is likely to be much larger. Hashing-based algorithms include the SpGEMM routines found in AmgX [6], KokkosKernels [3], ViennaCL’s Cuda kernel [7] and the work of Nagasaka et al. [8], [9].

Heap-based algorithms, such as the work of Nagasaka et al. [9] use a heap (or priority queue) to store the partially accumulated entries in the row. In the case where entries in B are sorted by column index within each row, these can be adapted to use even less memory than the hash-based algorithms. These methods also have the benefit of producing sorted output from sorted inputs.

The last major family are sort/merge algorithms. These work by using a parallel sort on component entries (if needed) and then merging them. These require more memory than a hash table and are potentially limited by the scalability of the sorting step. Nonetheless, they remain a popular choice and sort and/or merge based routines can be found in ViennaCL’s OpenMP kernel [7] and CUSP [10].

C. Algebraic Multigrid Setup

Algebraic multigrid generates a series of course approximations (or grids) to the original fine-scale problem. This is done via the generation of a coarse-to-fine grid prolongator matrix P , and a corresponding fine-to-coarse grid restrictor matrix R . For symmetric problems, $R = P^T$, but this is not a general requirement. The exact structure of the P and R matrices are chosen via the myriad of algorithmic options and parameters which guide AMG, but they do possess certain characteristics in general. P is always tall and skinny, R is always short and fat and in both cases, the number of non-zeros per row is relatively small. The choice of a relatively low

number of nonzeros per row is to keep density coarse matrix to a minimum (though some amount of density increase over the fine matrix is inevitable). Each coarse matrix, A_{n+1} is generated from the matrix at the next finer level, A_n via a matrix-matrix multiplication, specifically, $A_{n+1} = R(A_n P)$. This is done multiple times until a matrix small enough for a fast direct solver is generated, at which point the setup terminates.

These matrix-matrix multiplications are by far the most expensive components of multigrid setup (often taking 80+% of the total setup time). As AMG setup may occur every time step (for transient problems) or even multiple times per time step (for nonlinear problems), AMG setup (and thus SpGEMM) can be a substantial component of the overall code run time, which makes optimization of SpGEMM potentially impactful.

III. MOTIVATION

Our primary motivation for this work is the fact that prolongation matrices (P), discussed in Section II-C, are tall and skinny. Hence, the resulting product, $C = A_{N \times N} P_{N \times M}$, will contain rows with at most M nonzeros. This motivates a fundamental design decision. While many algorithms focus on hash or heap approaches for mapping column indices in P to locations in C , we note that that Gustavson’s original approach actually used *perfect hashing* via a dense array. For small M this works, but with large, distributed memory matrices, even M may be large, and P very sparse. To use a dense lookup table efficiently, there must be a mapping that restricts the columns to those owned by a specific MPI process. Thankfully, the framework we develop in is designed around arbitrary partitioning of sparse matrices. The ‘Petra’ model used in Trilinos [5] expresses both the rows and columns that are local.

Furthermore, all of the codes we work with are MPI-parallel, but not all kernels used in a given code are designed to thread-scale to very high counts. This is not always a programming deficiency, rather it is imposed by the algorithms used. For example, some algorithms may operate on a very large piece of a domain, yet yield a very small linear system that will be solved. Moreover, some codes find that their science performs best with modest thread counts, i.e., 4-8 threads per process. Therefore, these apps will run multiple MPI processes per node (or perhaps even multiple processes per socket). These facts motivate us to find algorithms that are suitable at low thread counts, which we consider to be 1-8 threads.

IV. LOW THREAD-COUNT GUSTAVSON

The fact that we have these ‘maps’ in Trilinos enables us to know exactly how many columns can possibly have a nonzero. It is this knowledge, coupled with Gustavson’s original algorithm using perfect hashing that leads to the Low Thread-count Gustavson (LTG) algorithm. Because we know exactly how many columns we will operate on, we are able to create a dense lookup table, that will have size at most M .

We then realize that we can partition Gustavson’s algorithm in such a way that threads can operate independently, with the penalty of having to allocate memory dynamically and then joining these contributions into the final resulting matrix structure at the end. In other words, we are exploring using more thread-local memory, while reducing our data movement of our inputs. We next digress into two general schemes for constructing a sparse matrix-matrix algorithm, and then explain how we thread parallelize our algorithm.

A. Two Passes Versus One

Typically, algorithms will require one or two passes over the input matrices. Two pass algorithms (such as KokkosKernels [3]) perform an initial pass that discovers the memory requirements of the resulting matrix. Then, the method allocates memory. Finally it performs a second pass which populates the matrix with output structure and values. Thus, two pass algorithms can be considered a count-allocate-compute approach. For some algorithms, the count phase can be very inexpensive, but for SpGEMM, counting the memory required requires that the number of nonzeros of each row in C must be discovered during this phase. Doing so requires the method to accumulate the nonzeros (but not their values) using one of the methods described in Section II-B. This means that such approaches may need to iterate through the input matrices twice. For example, both MKL Inspector and KokkosKernels expose these two phases through their API. Discovering the locations of the nonzeros in a row is the predominant expense of an SpGEMM algorithm (the actual floating-point operations are cheap), so having to do this twice represents a significant cost.

One pass algorithms must dynamically allocate memory as needed, since the number of nonzeros in the resulting matrix is not known. Furthermore, if the algorithm is thread-parallel, and the threads are using thread-local storage, then it must also have a means to aggregate all the independent storage created into the result matrix. The latter we refer to as “copy out.” Both this copy out and the dynamic memory allocation have a potential to be a significant cost for one pass algorithms.

In summary, two pass algorithms perform a pass that discovers the number of nonzeros per row and allocate memory before attempting computation. Single pass algorithms perform memory allocations as needed while performing the computation itself. In terms of computational cost, a single pass is desirable, but dynamic memory allocations are typically expensive. Moreover, scalable memory allocation with threads is an active area of research, e.g., tcmalloc [11] and jemalloc [12].

B. Thread Parallel Gustavson

While the nonzero structure of C may be unknown, we do know exactly how many rows C will have. In terms of CRS, this means we can preallocate storage for *crowptr*. We then partition the rows of the left-hand side matrix (A) into chunks. Each thread then performs Gustavson on their chunk. Because we have a *map* that tells us how many local nonzeros there are

Algorithm 2: Low Thread-Count Gustavson

```
input :  $A \in \mathbb{R}^{l \times m}$  and  $B \in \mathbb{R}^{m \times n}$ 
output: CRS arrays for  $C$ : crowptr, ccolind and cvals
Compute partitioning of rows of  $A$ ;
Allocate crowptr array and array for thread total nnz;
begin parallel region
  Allocate thread-local status array of size  $n$ ;
  Allocate thread-local colind and vals arrays page aligned;
  foreach row  $i$  in my thread's chunk do
    if thread-local array capacity  $< \max \text{nnz}/\text{row}$  then
      | Grow vals and colind by a hugepage;
    end
    Compute all entries  $c_{i,*}$  into vals and colind;
  end
  thread_total_nnz[thread_id] = thread nnz;
  if Final thread then
    | Allocate final cvals and ccolind arrays;
  end
  /* Combine thread-local storage */
  Correct crowptr offsets using thread total nnzs;
  Copy colind and vals into ccolind and cvals;
end
```

in our right-hand side matrix, we can allocate a dense lookup array per thread, even if this right-hand side matrix is large and distributed. Practically, these dense lookup arrays are very small, because the AMG algorithm constructs coarser levels and thus P is tall and skinny.

Effectively, LTG parallelizes the outer most loop in Algorithm 1. This is shown in detail in Algorithm 2. Each thread constructs an estimate for the number of nonzeros in a row in the output matrix. We can easily construct a crude upper bound, e.g., assume the row is dense. Instead, we approach this problem with a little systems insight. Since we have a gross upper bound for a row, we allocate this upper bound. That is, we allocate storage sufficient for a single row, but we page-align this memory, and always grab memory in terms of pages. This has a few benefits: First, it lets our code overallocate in a smart way, but since our overallocation is effectively the remainder of a page, this actually gives us some performance benefits. Second, by allocating in terms of page size, we observe that realloc becomes extremely fast. E.g., the underlying system is effectively appending a page(s) to our existing allocation record. This same behavior can be coded manually using the Linux calls `mmap` and `mremap`. Our code checks at the start of a row if our storage is sufficient to store the worst-case number of nonzeros. If our *capacity* is sufficient we can skip reallocating new memory. Furthermore, since we have threads calling `posix_memalign` and `realloc` in parallel, we investigated several memory allocators designed with threads in mind. The `jemalloc` library gave a boost to all kernels evaluated, so we use this allocator.

In terms of CRS storage, we can preallocate the *crowptr* array and have threads index into this shared memory. In this case, threads can store local offsets, rather than the correct global offset. Each thread must allocate (and realloc/grow) its *vals* and *colind* arrays. The values stored in the *colind* array are correct. That is, the thread knows which resulting column it is forming. This means that at the end of the kernel, we have two additional steps to perform: First, we must correct the *crowptr* array. Second, we must join all of the threads pieces of *vals* and *colind* together into one contiguous *vals* and *colind* arrays. Again, this operation is parallelizable. To correct the *crowptr* array a thread only needs to know how many nonzeros all threads who owned rows before it discovered. This is a *prefix sum*. E.g., if three threads independently computed 5, 6, and 7 nonzeros. Then thread 0 needs to apply no correction. Thread 1 applies a correction of 5, and thread 2 applies a correction 5+6. Which yields a *rowptr* array with values in the range $[0, 5+6+7+1]$. Applying the correction is simple, with each thread adding their correction to their portion of the *crowptr* array. The second operation is to unify all *vals* and *colind* arrays into the output arrays *cvals* and *ccolind* array. This can be done by having one thread allocate the required storage, then each thread copies their arrays into the correct offset of each array. These final steps entail a single thread barrier, and can be implemented inside the same parallel region. That is, we only need to execute one `pragma omp parallel`. These steps take place in the last 6 lines of Algorithm 2.

In summary, we partition the rows of the right-hand side matrix, and then perform Gustavson locally using thread local storage for *vals* and *colind*, while constructing the *crowptr* in place. Our dense lookup array remains small because we have knowledge of how many nonzeros are owned by this specific process (if the matrix is parallel). We made memory allocations fast by using page-aligned allocations, and by tracking used size and capacity separately. We require a single synchronization point to aggregate the total number of nonzeros computed by all threads, and then a single memory allocation using this information. After this, each thread is able to continue in parallel, copying their data into the final structures. We do perform some optimizations in our copy out phase. For example, we enforce a coloring on thread writes to specific arrays. This helps avoid false sharing if two threads attempt to copy into the same page of memory. We also use the low level *memcpy* routine, since the *vals* and *colind* arrays only need to be copied (not corrected in anyway). We experimented with each thread manually performing the copy via assignment in a loop. We found *memcpy* outperforms the loop-based copying by a large margin, which is not surprising given how much optimization has gone into that routine.

V. EXPERIMENTAL EVALUATION

We implemented our algorithm in the Trilinos [5] framework and evaluated the code on a Cray XC40 system using the Haswell partition. Our configuration and runtime parameters are described in Table I. Of note, we always bind our program

TABLE I: System and runtime configuration

CPU	Each node has two sockets, each socket is populated with a 16-core Intel Xeon Processor E5-2698 v3 ("Haswell") at 2.3 GHz
Memory	Two NUMA nodes per node, and 64GB DDR4 memory per NUMA domain.
Compiler	Cray PrgEnv-Gnu, GCC 7.2.0
Libraries	Intel MKL 17.0.098 ViennaCL 1.7.1
OpenMP ENV	OMP_PLACES=cores, OMP_PROC_BIND=spread
SLURM	-cpu-freq=High -cpu_bind=cores -mem_bind=local -c 16 -hint=nomultithread -m block:block

to a single socket, and we chose not to bind to hardware threads (HT). We do not present results with two hardware threads per core used (they generally slowed down all runs). The resulting CPUMASK on a dual socket Haswell node was 0xFFFF, and we restricted our driver to the NUMA domain local to the socket.

A. Methodology

We developed a driver code for performing arbitrary sparse matrix-matrix products, and instrumented this code using the C++11 `std::chrono::steady_clock`. Our driver effectively constructs two matrices and provides timers around calls into various libraries (e.g., MKL, ViennaCL, and our own). We observed a median timing overhead of approximately 17 ns. Given that a single SpGEMM trial was always on the order of 1.e7 ns, we consider timing overhead to be negligible.

We perform 10,000 randomized trials for each of the kernel/thread/matrix combinations presented. From our timing data, we then compute descriptive statistics as well as a confidence interval on our median value. All thread scaling figures use box plots, with the median annotated using a symbol. For our qualitative study, we present the median value only.

We must point out that not all kernels use the same input formats. For example, MKL requires the matrices be in their CRS datatype. **When we time a kernel, we do not count the time it takes to copy in/out data from the third party data structures.** We feel this is fair to the external libraries. In other words, our timings are taken only around the actual call to the library's SpGEMM. The reasoning is that if MKL or ViennaCL were sufficiently fast, then we would be motivated to use their data structures natively. In this work, we do not want to penalize the libraries for copyin/out work. In all cases, we expect the libraries to take as input two matrices, and return a solution matrix. We then verify the correctness of the solution obtained using the Frobenius norm.

B. Thread Strong Scaling

The MueLu [13], [14] algebraic multigrid package in Trilinos [5] can generate two kinds of prolongators, one of which we will refer to as "smoothed" and the other as "unsmoothed." For any given problem, both prolongators have the same number of columns, but the smoothed prolongator contains a higher number of nonzeros.

We consider strong thread scaling on two different size matrices extracted from a real MueLu setup. The matrix we performed setup on is called "Brick3D", which is a discretization on a 3D grid resulting in approximately 27 nonzeros per row. The two sizes chosen are a matrix formed from a 50^3 grid, and a 100^3 grid. This yields matrices with approximately 120k and 1MM rows respectively. Using this input matrix, we then extract two pieces that are used to form the P (prolongator) matrix in RAP . The pieces in this case are the smoothed and unsmoothed versions of P .

By changing the size of the input matrix (A), we are effectively increasing the number of rows and columns in both A and P , but we are not changing the nonzero count per row in the right-hand side matrix P . Since a size change increases the number of columns in P , then our dense lookup must also be larger.

In summary, The size of A dictates the number of rows and columns in both variants of P ; The variants of P have one nonzero per row (unsmoothed) and approximately eight nonzeros per row (smoothed). Therefore, the product with unsmoothed P has fewer flops and reduced data movement, while the product with smoothed P has substantially more flops and increased data movement.

Figure 1 shows strong scaling on two sizes of Brick3D when computing the unsmoothed prolongator. In this case, the number of nonzeros per row in the right-hand side matrix is one. We observe very good performance, with LTG often outperforming competing libraries by 2x. Moreover, we often observe that our slowest runtime is often faster than our competitors best runtime. We believe our performance on unsmoothed prolongation is superior for two reasons. First, the number of nonzeros in the right-hand side matrix is very small (one). Hash tables and heaps incur some setup overhead, while our kernel uses perfect hashing and requires no overhead (reinitializing or invaliding) between iterations. Second, our kernel only makes a single pass over the left-hand side matrix. Therefore, we have less data movement. Our penalty, is that our dense lookup table requires storage of atmost M , while the hash based approaches may use much less storage. Our general approach asl requires that we allocate memory twice the size of our resulting matrix. That is, we must store our in-progress resulting matrix, then must allocate and copy out into the final contiguous structure (*cvals* and *ecolind* in Algorithm 2).

Next, we consider the smoothed prolongator construction in Figure 2. In this case, the right-hand side matrix contains approximately 8 nonzeros per row, which means the innermost loop has a non-trivial amount of work, and that the lookup tables will actually be populated. Again, we observe very good performance. At high thread counts, the gap between LTG and MKL is small, but statistically significant. Table II shows the 1% and 99% percentiles, 99% confidence interval for the median, and arithmetic mean for MKL and LTG. We find that our kernel is prone to larger extreme values, while having a statistically significant median timing (w.r.t. MKL). While we do have longer 99th percentile runs, the arithmetic average is smaller in all cases except 16 threads. We feel it is fair

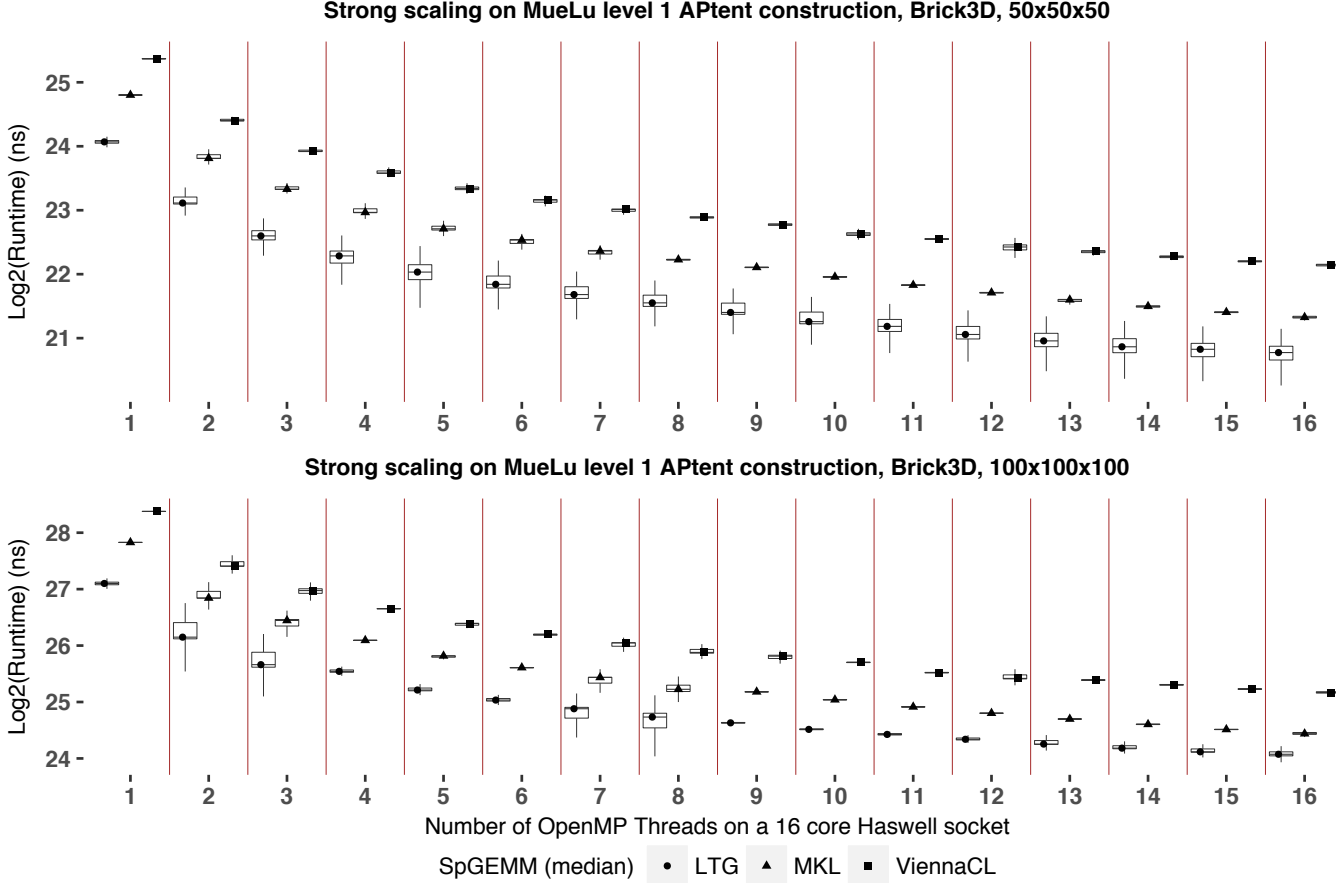


Fig. 1: MueLu A times unsmoothed P using the Brick3D matrix with size 50^3 and 100^3 .

to say our algorithm is competitive. Our kernel must perform some dynamic memory allocations, and these are expensive (even with page aligned allocations). We also require a the copy out routine, which further penalizes our performance, yet even with this overhead we still perform well.

A general trend we observe for both smoothed and unsmoothed prolongation calculation, is that while our kernel is very fast, it seems prone to more performance variation. Still, our non-outlier values, i.e., the whiskers on our box plots are very competitive. From Table II we show that with respect to the median and arithmetic mean, we have proposed an algorithm that deserves further research.

C. Sparse Matrix Squaring Survey

We next consider a sweep of 72 matrices from the Spars-eSuite [15] that have dimension between 10k and 120k. We intentionally sought matrices from many classes (as classified by the Suite Sparse maintainers). Table III summarizes the matrix classes and number chosen. The purpose of the survey is qualitative. We are not asserting that our algorithm is the best algorithm for matrix squaring, but we feel the survey supports our claim that our method is competitive in this case. Figure 3 plots the relative difference in the median runtime

between LTG and MKL. For each matrix and thread count we conducted 10,000 randomized trials.

We observe that our algorithm performs very well with lower thread counts, but remains competitive even at 16 threads. Matrix squaring is particularly hard for our algorithm, because we perform dynamic memory allocations as we iterate. Matrix squaring generates a non-trivial amount of fill-in, which means that our algorithm typically allocates memory more often. Even with the added cost of memory allocation, we still observe superior performance in some cases. In general, the majority of runtimes are roughly the same as MKL's. While there are matrices we perform significantly better on, there are also matrices we perform significantly worse on, e.g., $\pm 50\%$ of MKL performance. We could improve our algorithm if we anticipate higher fill. As-is, squaring triggers more reallocs, which we could mitigate by using a larger estimate for matrix fill. Moreover, our performance is the worst with higher thread counts. With low thread counts, we have very competitive performance the majority of the time.

VI. CONCLUSION

We have presented a thread-parallel version of Gustavson's row-wise SpGEMM algorithm that is a single pass algorithm.

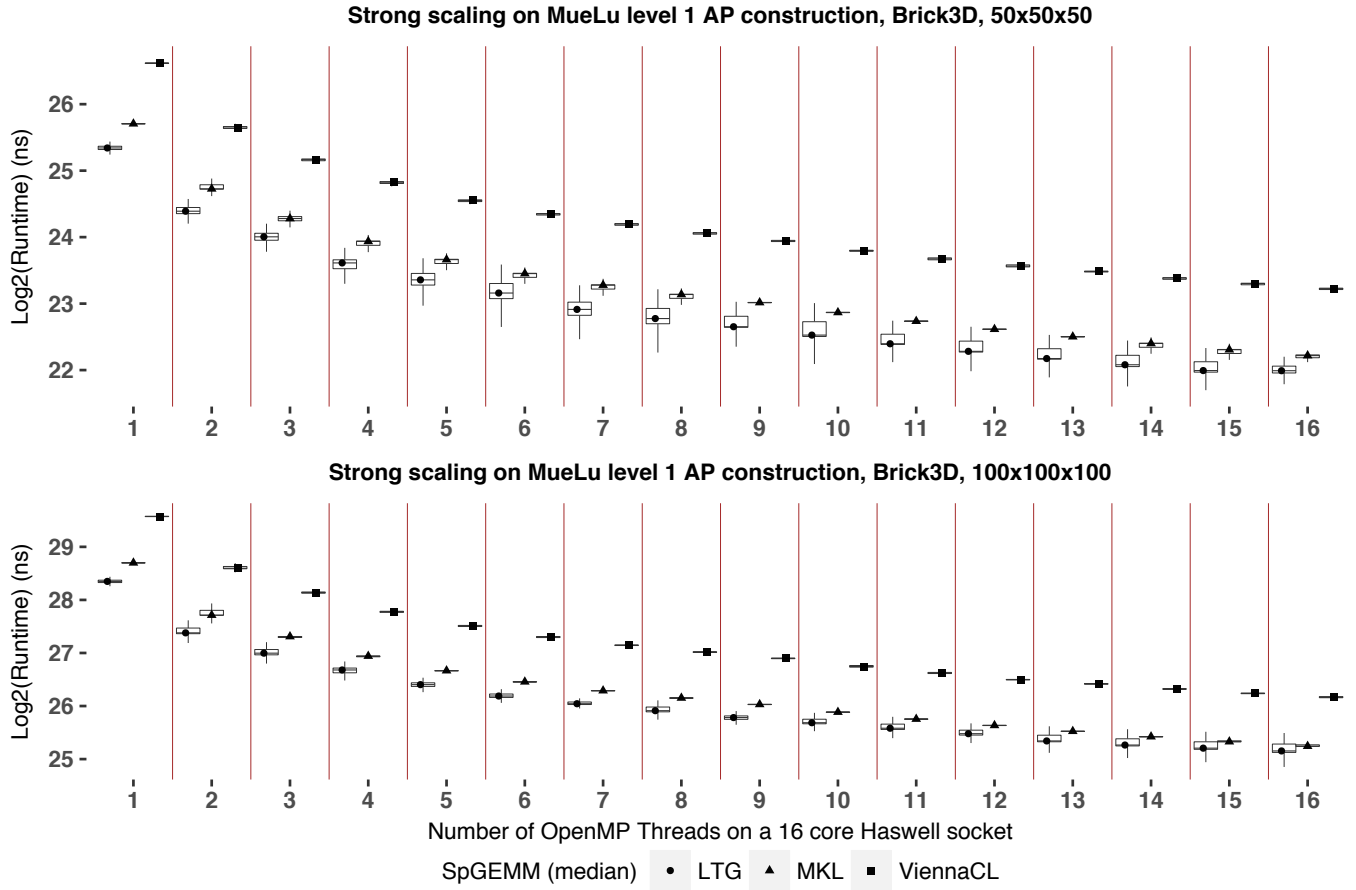


Fig. 2: MueLu A times smoothed P using the Brick3D matrix with size 50^3 and 100^3 .

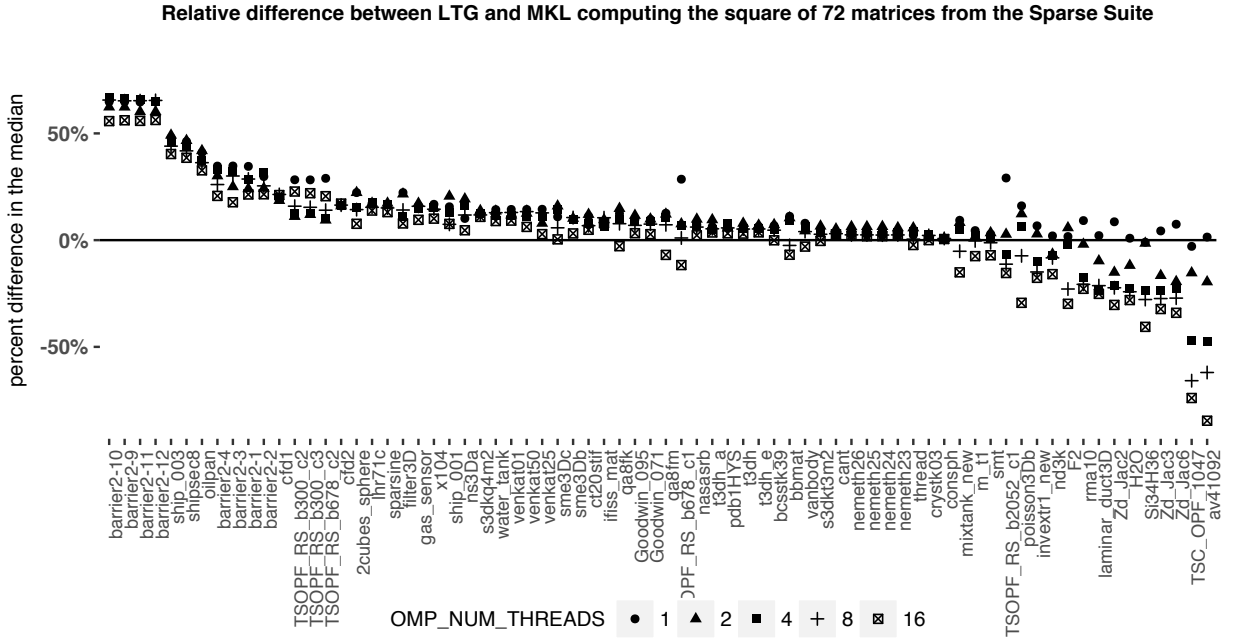


Fig. 3: Survey of performance difference with respect to MKL for matrices from different classes of problems. Computing $C = A^2$, using various thread counts and taking the median timing from 10,000 randomized trials

TABLE II: Statistics for smoothed AP from Brick3D 100³.

		1% Perc.	Lower CI	Time in μ s Upper CI	Mean	99% Perc.
1	LTG	330,155	341,666	342,052	362,611	650,240
	MKL	435,179	435,638	435,657	438,397	517,229
2	LTG	166,555	174,388	174,713	188,879	351,011
	MKL	219,294	219,939	220,076	226,830	278,808
3	LTG	121,701	133,776	134,027	145,709	294,920
	MKL	156,054	166,290	166,321	165,980	200,268
4	LTG	95,701	107,387	107,589	115,838	228,086
	MKL	120,789	128,848	128,859	128,803	159,382
5	LTG	80,492	88,667	88,833	96,676	197,074
	MKL	99,876	106,515	106,527	106,945	133,095
6	LTG	69,524	76,397	76,569	82,938	161,213
	MKL	86,138	92,078	92,089	92,663	119,282
7	LTG	63,392	69,136	69,212	75,031	159,042
	MKL	76,547	82,016	82,026	82,914	105,953
8	LTG	57,353	63,086	63,220	68,784	144,769
	MKL	69,501	74,546	74,556	75,438	100,451
9	LTG	51,946	57,560	57,883	62,568	138,034
	MKL	63,874	68,539	68,545	69,353	92,074
10	LTG	49,222	53,937	53,996	58,576	122,891
	MKL	57,662	61,974	61,980	63,042	88,166
11	LTG	45,601	50,201	50,236	54,479	117,610
	MKL	52,607	56,595	56,599	57,593	82,147
12	LTG	42,374	46,731	46,790	50,864	109,626
	MKL	48,381	52,105	52,109	53,113	76,926
13	LTG	38,828	42,507	42,557	46,935	112,232
	MKL	44,775	48,268	48,273	49,416	71,519
14	LTG	36,966	40,289	40,351	44,570	86,942
	MKL	41,666	44,965	44,970	46,172	69,513
15	LTG	35,307	38,670	38,756	43,213	105,991
	MKL	39,008	42,127	42,137	43,458	65,753
16	LTG	33,937	37,287	37,354	41,526	89,287
	MKL	36,690	39,707	39,744	41,256	64,995

TABLE III: Matrix classes and counts used in SpGEMM survey

Class	Count	Class	Count
chemical process simulation	4	2D/3D	4
computational fluid dynamics	16	acoustics	2
semiconductor device	8	materials	1
theoretical/quantum chemistry	6	structural	18
weighted undirected graph	1	model reduction	5
electromagnetics	1	power network	6

We devised a scalable strategy for joining each thread’s contribution back into a single CRS data structure, and are able to compute our CRS *rowptr* in place. Our algorithm is motivated by characteristics of the matrices arising in AMG setup, and for these types of matrices, we observed superior, scalable performance. We observed performance improvements of nearly 2x over competing kernels at low thread counts, and even at higher thread counts our algorithm is competitive. We also conducted a qualitative survey of squaring arbitrary matrices from the Sparse Suite, and presented findings that our kernel

performs reasonable well over a large spread of matrices.

We leave to future work several open questions: First, we believe we can remove the need to copyout our resulting matrix. Second, it may be beneficial to introduce some blocking into our thread’s local SpGEMM loops. Third, the ideas underpinning our algorithm can be applied to other AMG operations, and potentially introduced into performance-portability libraries such as Kokkos [16].

ACKNOWLEDGMENT

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

- [1] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*. San Diego: Academic Press, 2001.
- [2] L. Dagum and R. Menon, “Openmp: an industry standard API for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.
- [3] M. Deveci, C. Trott, and S. Rajamanickam, “Performance-portable sparse matrix-matrix multiplication for many-core architectures,” in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2017, pp. 693–792.
- [4] F. Gustavson, “Two fast algorithms for sparse matrices: multiplication and permuted transposition,” *ACM Transactions on Mathematical Software*, vol. 4, pp. 250–269, September 1978.
- [5] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, “An overview of Trilinos,” Sandia National Laboratories, Tech. Rep. SAND2003-2927, 2003.
- [6] J. Demouth, “Sparse matrix-matrix multiplication on the GPU,” in *Proceedings of the GPU technology conference*, 2012.
- [7] K. Rupp, F. Rudolf, and J. Weinbub, “Viennacl - A high level linear algebra library for GPUs and CPUs,” in *International Workshop on GPUs and scientific applications*, 2010, pp. 51–56.
- [8] Y. Nagasaka, A. Nukada, and S. Matsuoka, “High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal GPU,” in *2017 46th International Conference on Parallel Processing (ICPP)*, vol. 00, Aug. 2017, pp. 101–110.
- [9] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, “High-performance sparse matrix-matrix products on intel KNL and multicore architectures,” *CoRR*, vol. abs/1804.01698, 2018.
- [10] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrix-matrix multiplication for the GPU,” p. 25, 2015.
- [11] S. Ghemawat and P. Menage, “Tcmalloc: Thread-caching malloc,”
- [12] J. Evans, “Scalable memory allocation using jemalloc,” 2011.
- [13] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro, “Design considerations for a flexible multigrid preconditioning library,” *Sci. Prog.*, vol. 20, no. 2, 2012.
- [14] J. J. Hu, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner, “MueLu multigrid framework,” 2014.
- [15] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.
- [16] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.