SAND2016-12069C

Performance Portable Sparse Matrix-Matrix Multiplication on Intel Knights Landing and NVIDIA GPUs

Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam
Sandia National Laboratories, Albuquerque, NM
{mndevec,crtrott,srajama}@sandia.gov

We consider the problem of writing performance portable kernels targeting different architectures with the sparse matrix-sparse matrix multiplication kernel (SpGEMM) as our benchmark. We approach the SpGEMM from the perspectives of algorithm design and implementation, its practical usage and a theoretical model for memory access. We design this kernel to be portable and perform well on two recent, but different massively-threaded architectures, namely Intel's Knights Landing processors (KNLs) and NVIDIA's Graphic Processing Units (GPUs). First, we design a hierarchical memory-efficient SpGEMM algorithm and implement thread scalable data structures that enable us to develop a portable SpGEMM algorithm. We demonstrate the decent performance of the performance-portable implementation compared to vendor provided, specialized implementations on both architectures. Second, we study the practical usage aspects of SPGEMM. We focus on the reuse of the structure of input matrices, and show that our kernel is significantly faster than specialized vendor kernel on the KNLs when the structure of the matrices are reused. We also study the performances of the algorithms on two different types of memory in KNL. Furthermore, we implement three other hierarchical SPGEMM methods using larger-memory required data structures, and we demonstrate the superiority of the memory-efficient method. Third, we develop a theoretical hypergraph model to study the memory accesses of these four variants, and study the performances on two different memory spaces on KNL for various scenarios. We show that the model serves as a guide to understand the performances of the algorithms on various memory spaces.

I. Introduction

With modern supercomputer architectures moving simultaneously in two different paths, namely the Intel's Knights line of processors (most recently the Knights Landing or KNLs) and NVIDIA's Graphic Processing Units (GPUs), it has become important to design algorithms and implementations that can perform well on both platforms. The current focus on this problem has been mostly around programming models that will allow users to implement an algorithm for multiple architectures [1]. We approach this problem from an algorithmic perspective to design a "performance-portable algorithm" – an algorithm and its implementation that perform well on multiple architectures. Another options is to write different, architecture specific ("native") implementations for every key kernel. The question then becomes how much performance will be sacrificed for performance-portablity ?

It is important to study this problem with a kernel that is reasonably difficult, uses standard patterns that allows us to reason about designing other kernels based on the knowledge gained. We choose the sparse matrix-matrix multiply (SpGEMM) kernel as our benchmark for this study. SpGEMM is a fundamental kernel that is used in various applications such as graph analytics and in scientific computing especially in the setup phase of multigrid solvers. The kernel has been studied extensively in the contexts of sequential [2], shared memory parallel [3], [4] and GPUs [5], [6], [7], [8]. There are native kernels available in different architectures [4], [5], [6], [8], [9] providing us good comparison points.

We try to answer the following questions for the SpGEMM kernel from the perspectives of algorithm design and implementation, practical usage of the kernel, and theoretical model for memory accesses.

- What are the design choices that are crucial for the performance of SpGEMM in different architectures and how do these choices affect/enable an algorithm to map to the differences in both architectures (thousands of threads vs hundreds of threads, streaming multiprocessors vs traditional but lightweight cores, small shared memory vs on-package High Bandwidth Memory) ?
- How will the kernel serve the practical needs of real applications. When applications has a reuse of the symbolic structure, or when the architecture has limited high bandwidth memory (HBM)?
- Can we design a theoretical model for memory accesses to understand the performance of the kernel in different architectures on various memory spaces?

In addressing these questions we make the following contributions

- We design thread scalable data structures to handle multilevel hashmap, memory pool. We design a graph compression technique to speedup symbolic phase SPGEMM.
- We design and implement a hierarchical, memory-efficient SpGEMM algorithm, using the thread-scalable data structures and show its performance in comparison to both vendor provided and third party native implemn-

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.

tations.

- We also implement three other variants of the SpGEMM algorithm with different design trade-offs and compare their performance to the memory-efficient version on various scenarios.
- We develop a theoretical hypergraph model for measuring memory accesses in both architectures and study its relationship to measured performance.
- We study practical usage of this algorithm when the HBM space is limited or when the structure can be reused compared to other implementations

The rest of the paper is organized as follows. Section II covers the background for SpGEMM, especially the important differences between the different native implementations. Our algorithmic variants and implementation of the data structures are described in Section III. The theoretical model for memory accesses is given in Section IV. Finally, the performance comparisons that demonstrate the efficacy of our approach is given in Section V.

II. Background

In this work, we study scalable and memory efficient SpGEMM methods for multi-core and many-core architectures. This kernel is widely used in the literature for various graph analytic problems. Multigrid solvers uses triple products in the setup phase, which is in the form of $A_{coarse} = R \times A_{fine} \times P$ ($R = P^T$ if A_{fine} is symmetric), to coarsen the matrices.

In the literature, most parallel SPMM methods follow Gustavson's algorithm [2] given in Algorithm 1. This algorithm schedules the computations in 1D row-wise fashion (line 1), and multiplication results for all entries in a row are found simultaneously. Each iteration of the second loop (line 2) accumulates the intermediate values for multiple columns within of the row. Different data structures can be used as accumulators. The result for the whole row i is found at the end of this loop.

Algorithm 1 1D rowwise SPGEMM for $C = A \times B$.

```
Require: Matrices $A$, $B$
1: for $i \leftarrow 0$ to $numRows(A) - 1$ do
2:     for $j \in A(i,:)$ do
3:         $C(i:,) \leftarrow C(i,:) + A(i,j) \times B(j,:)$
```

There are various memory constraints with SPGEMM. For example, the size and structure of C is unknown at the beginning of SPMM operation. Finding the structure is usually as expensive as performing actual floating point operations. Although, there exist studies to predict the size of C [10], they still do not provide a robust upper bound for the memory requirements. In the literature, this problem is addressed using various approaches. One approach is to calculate the upper bound for the size of C, and allocate this memory prior to computation. However, the upper bound becomes the number of floating-point operations (FLOPS), which can be significantly larger than the actual size of C. Another approach is to dynamically reallocate the size of C as the computation progresses. However, this approach might also be problematic in the modern architectures with massive number of threads, as memory re-allocations can become bottlenecks in parallel regions. In addition, such re-allocations are not feasible for GPUs. Another approach is to perform a symbolic SPMM operation before the numeric computations to compute the accurate size of C [5]. Although, this approach doubles the overall number of operations, it allows SPMM to run with minimal memory usage. Symbolic phase needs to be run only once for matrices in which only the numeric values change while the symbolic structures are preserved. In this work, we follow this approach, and we aim to speedup the symbolic phase by performing matrix compression.

Another memory constraint with the parallel Gustavson algorithm is the data structure to use for the accumulators. The sequential algorithm uses dense data structures that have the size of the number of columns in B ($numCols(B)$), in order to accumulate the result rows. However, having such thread private arrays is costly on massively threaded architectures. Moreover, random accesses to such large arrays might harm the overall performance because of the memory bandwidth issues. Therefore, sparse accumulators such as heaps or hashmaps are usually preferred in parallel implementations. In this work, we use multi-level hashmaps as sparse accumulators.

There are various works that consider distributed parallization of SPGEMM. Tpetra package of Trilinos [11] performs $1D$ row-wise partitioning that bases on the Gustavson's algorithm. In a $1D$ algorithm the result matrix is $1D$ partitioned and each row is calculated by a single computation unit. On the other hand, SUMMA algorithm that partitions the result matrix into $2D$ space is adopted in Combinatorial BLAS [12]. This algorithm is extended to $3D$, via parallelization over calculation of single entry in [13]. Hypergraph models and algorithms that use hypergraph partitioning for sparse matrix-matrix outer-product multiplications and $3D$ computation partitioning are studied in [14] and in [15], respectively.

Most of the sparse matrix matrix multiplication studies for multi-threaded architectures bases on Gustavson's algorithm. They study efficient parallelization of Gustavson's method, and usually differ in the data structure used fo row accumulation. One option is to use dense accumulators [3], which have the size of columns of the result matrix. Another approach is the use of sparse accumulators. Azad et. al. [13] uses heaps as accumulators in their shared memory implemention. This bases on the assumption that the rows of B have sorted column indices. Similarly, ViennaCL [9] provides SPGEMM implementations written with CUDA, OpenCL and OpenMP, and it implements row merges [7] that bases on merge sort of the columns of B. MKL [4] also provides a multi-threaded implementation that uses sparse accumulators without any assumption on the order of the columns of B. As a different approach than all, Patwary et. al. [3] also studies $2D$ partitioning of the result matrix. At their preprocessing step, they partition B based on the columns, however the gain only amortizes the cost of preprocessing on certain conditions, for which they have a heuristic for partitioning decision.

Extensive parallelism of GPUs are exploited for SPGEMM in the literature, and most of the algorithms uses multiple levels of partitioning in the literature. CUSP [8] has a $3D$ algorithm that bases on expand and sort methods (ESC). Each multiplication is performed by a single thread and they are accumulated at the end with a sort operation. However, accumulation is performed globabaly, therefore the memory requirements of this method is usually high. cuSPARSE [5] follows a **h**ierarchical Gustavson algorithm. In a hierarchical algorithm, rows are first assigned to first level of parallelism (blocks or warps), and the calculations within the row are calculated using the lower level parallelisms. In cuSPARSE's case, each row is calculated by a single warp, and multiplications within a row is done by different threads of the warp. They use 2 level hash map accumulators, and do not make any assumption base on order of the column indices. On the other hand, row merge algorithm [7] and its implementation in ViennaCL [9] uses merge sorts for accumulations of the sorted rows. Similarly, bhSPARSE [6] exploits this assumption on GPUs. It has methods to predict the size of the result matrix. It performs binning based on the size of the result rows, and performs accumulation for the multiplication of the row using heap accumulators, ESC or row merge based on the size of the row.

Table I lists the summary of the literature. The last line list our library, KokkosKernels (KK), and the decisions we follow in this work. Among these algorithms, the ones that uses merge sorts and heaps, requires the columns of B to be sorted. ViennaCL and cuSPARSE (and KK) follows 2-phase approach where they first calculate the input structure, and then perform the actual flops. bhSPARSE follows somewhat similar approach, however, it estimates the size of the result matrix, while CUSP over allocates the memory. Other multi-threaded methods usually follow 1-phase approach with by dynamically adjusting the result matrix size.

A different approach to SPGEMM method is studied by Mccourt et. al [16]. In the symbolic phase, they find the structure of the matrix. Using this structure they perform a distance-2 graph coloring on the C to find the rows that do not share any column. In the numeric phase, then in the numeric phase, the rows with the same color are compressed to dense matrix, and sparse matrix-dense matrix multiplication is performed. The result is projected back to the sparse matrix. However, this method requires the conversion from sparse matrix to dense matrix for all numeric steps. In Section III we also study distance-2 graph coloring, however we use this to share dense accumulators by multiple threads.

A. Kokkos

Kokkos [1] is a C++ library that provides an abstract thread parallel programming environment and enables performance portability for common multi- and many-core architectures. It provides a single programming interface but enables different

TABLE I: Summary of the SPGEMM work in the literature that bases on the partitioning scheme, paralellism and accumulator they use. HM and MS denotes Hashmap and Merge Sort, respectively.

	Partitioning	Parallelism	Accumulator
Gustavson	1D	Sequential	Dense
Trilinos [11]	1D	Dist.	Dense
ComBlass [12]	2D	Dist.	Heap
Azad et. al. [13]	3D	Dist./Multicore	Heap
MKL [4]	1D	Multicore	
Patwary et. al. [3]	1D/2D	Multicore	Dense
ViennaCL - OMP [9]	1D	Multicore	MS
CUSP [8]	3D	GPU	ESC
cuSPARSE [5]	Hier.	GPU	HM
ViennaCL-Cuda [9] Gremse et. al. [7]	Hier.	GPU	MS
bhSPARSE [6]	Hier.	GPU	Heap/MS/ESC
KokkosKernels	Hier.	Multicore/GPU	HM

TABLE II: Kokkos Hierarchy mapping to GPUs and CPUs

	GPUs	CPUs
Team	Block	Work assigned to group of hyperthreads
Kokkos Thread	(half, quarter...) Warp	Work assigned to a single Thread
Vector Lane	Threads within a warp	Vectorization Units

optimizations for backends such as OpenMP and CUDA. We used only a subset of Kokkos' features, mainly parallel_for, parallel_scan, atomics, and views (arrays), as well as the Kokkos thread hierarchy. Using Kokkos allows us to run the same code on Xeon Phi and GPU just with different compile options.

The kokkos-parallel hierarchy consists of teams, threads and vector lanes. Table II shows how we map these terms to computation units on GPUs and CPUs. A team in kokkos refers to independent bunch of work that is assigned to group of threads that can share resources. This refers to thread blocks on GPUs, where threads have shared memory spaces. On the other hand on CPUs, a team can be more flexible. It can consists of all threads sharing the DDR memory, or the threads sharing L2 or L1 cache. To our experience, the best performance with teams is achieved by using the hyperthreads that share L1 cache. Work-sharing of the threads within the team is achived by assignment of the consecutive indices to consecutive threads. This allows coalesced memory accesses on GPUs, while on CPUs, this allows better cache reuse if threads within the team are chosen as hyperthreads that run on the same core. Note that, if a team on CPU consists of threads that shares DDR memory, this causes false sharing. Note that, there is not one-to-one mapping from kokkos-teams to the used number of computational units. That is the number of teams on CPU are much higher than the number of computation units. A computation unit is assigned various number of teams. A kokkos-thread within a team consists of vector lanes. It can map to a warp (half, quarter warp, or so on.) on GPUs, while it on the CPU side it corresponds to vectorization units. We will use the terms, teams, kokkos-threads and vector lanes for the rest of the paper, which refers to the explained computational units on CPUs and GPUs.

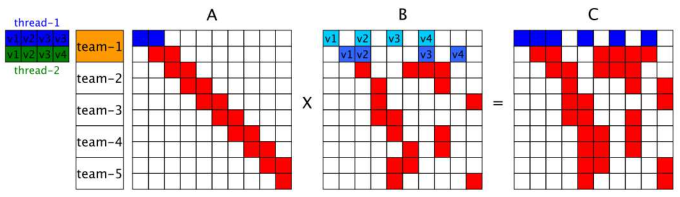


Fig. 1: SpGEMM kokkos thread hierarchy. Team-1 consists of 2 kokkos-threads, and each kokkos-thread has 4 vector lanes as shown with v[1-4]. Thread-1 is assigned to the first row. It does a vector read (and multiply) of the first row of B, then it continues with the second row of B (shown with different tones of blue). As a result of this accumulation, thread-1 simultaneously computes the result of the entries in the first row of C.

B. GPUs and KNL architectures

III. Algorithms

We consider a 2-phase approach for our matrix-matrix multiplication method. In the first (symbolic) phase, the number of nonzeros in each row is calculated, while in the second (numeric) phase, the actual flops are performed. We choose this approach over 1-phase approaches for the below reasons:

- 1-phase methods either over-allocate memory for the result matrix or dynamically increase its size throughout the computation. The former approach can require significantly larger memory than the actual result size, while the latter is not portable, i.e., they are not suitable for GPUs. Methods also exist to estimate the structure of the result matrix [10], however, they do not provide an upperbound. Moreover, since they perform a sparse matrix-dense matrix multiplication, they are not significantly cheaper than a symbolic phase.
- It is a common use case of SpGEMM in scientific computing to repeat multiplication of matrices that maintain the symbolic structure, as the numerical values evolve. For such cases, the first-phase only needs to be executed once, and can then be reused for the rest of the numerical multiplications.
- 1-phase aproaches allocate and return memory using their own memory management systems. This is not desirable in a practical use-case for applications using these service-kernels.

2-phase methods double the work done, since the first and second phases perform a similar amount of work, with the additional flops in the second phase. We aim to improve the performance of symbolic phase with some compression methods as explained below.

A. Core SpGEMM Kernel

Both symbolic and numeric phases follow a hierarchical row-wise algorithm. The structure of the core SpGEMM kernel is given in Algorithm 2, and a simple scenario is shown in Figure 1. Kokkos-teams are assigned to a set of rows, and kokkos-threads within the teams are assigned to a subset of these rows. The result of one row is computed by a single kokkos-thread. For example, the first row in Figure 1 is assigned to thread-1 (highlighted in blue). Kokkos-threads allocate some

Algorithm 2 SpGEMM Kernel for $C = A \times B$

```
Require: Matrices $A$, $B$
 1: for each thread $\in$ thread_team $\in C(*,:)$ do
 2:   $i \leftarrow$ GETMYROW(thread_team, thread)
 3:   allocate the first level accumulator $Acc\_L1$
 4:   for col $\in A(i,:)$ do
 5:     $cols, vals \leftarrow$ VECTORREAD($B(col,)$)
 6:     $vals \leftarrow$ VECTORMULT($vals, A(i,col)$)
 7:     if FULL = $Acc\_L1$.VECTORINSERT($cols, vals$) then
 8:       if $L2\_not\_allocated$ then
 9:         allocate the second level accumulator $Acc\_L2$
10:         $L2\_allocated \leftarrow True$
11:       $Acc\_L2$.VECTORINSERT($cols, vals$)
12:   if SYMBOLIC then
13:     $rowSize(i) \leftarrow$ total L1/L2 hashMap sizes
14:   else
15:     $C(i,:) \leftarrow$ values from L1/L2 hashmaps
16:   if $L2\_allocated$ then
17:     release $Acc\_L2$
```

scratch memory for their private level-1 (L1) accumulator. This scratch space is located at the shared memory of GPUs. It is at the default memory (i.e., DDR4 or high bandwitdh memory) on CPUs, but it is usually small enough to fit in L1 or L2 cache. Then, the kokkos-threads read the columns in $A(i,:)$ sequentially, perform a vectorized read and multiply on the corresponding row of B. Multiplication results are inserted into L1 accumulators. If the L1 accumulator runs out of space, some global memory is allocated (explained later) for a private L2 accumulator, and this L2 accumulator is used for failed insertions. Once all insertions to the accumulators are completed, the size of the accumulation is stored as the row size in the symbolic phase, while result column indices and their values are written to the corresponding rows of C in the numeric phase. L2 accumulators are released, if allocated.

Kokkos-threads consist of extra level of parallelism, i.e., vector lanes (Figure 1). The length of vector lanes is a runtime parameter, and it is fixed for all threads in a parallel kernel. It is calculated by rounding the average number of nonzeroes in a row of B (δ_B) to the closest power of 2 on GPUs. It is left to the compiler and underlying Kokkos framework based on the architecture specifications on the CPU side.

The size of L1 accumulators depends on the available shared memory of GPUs. If L1 runs out of memory, L2 accumulator is allocated. The size of L2 accumulator is chosen as the maximum row size so that it is guaranteed to have enough space for the rest of the accumulations. Before the symbolic phase, maximum size of a row is predicted as its upperbound, which is the max flops in a row. This is more accurately chosen as the maximum of the actual row sizes in the numeric phase, as the actual sizes are known. On the other hand, both L1 and L2 accumulators are located in the same memory space on CPUs. Moreover, since there are more resources per thread on

CPUs, we skip L1 accumulator, and only use L2 accumulators to hold all required entries. Yet, L2 accumulator is usually small enough to fit into L1 or L2 caches.

Note that we do not deal with the floating point values in the symbolic phase. Therefore, there is no read, multiply and accumulation of the floating points. However, in order to speed up this phase and reduce the memory requirements by accumulators, we compress B and introduce new values, a process which is explained below.

1) Compression: The row-wise algorithm in Algorithm 2 performs a single scan of A and multiple scans of B. That is, a nonzero value in B is read multiple times, and total accesses to B is as many as FLOPS. Specifically, row i of B, $B(i,:)$, is read as many times as the number of nonzeroes in $A(:,i)$. If we assume that the structure of A is uniform, that is, there are δ_A nonzeroes within each column and row, each row of B is accessed δ_A times. Thus, the FLOPs become $O(\delta_A \times nnz_B)$. If a compression method with linear time complexity reduces the size of B by some ratio $Comp$, the amount of work in the symbolic phase can be reduced by $O(Comp \times \delta_A \times nnz_B)$, with a compression price of $O(nnz_B)$.

Symbolic structure refers to the underlying graph structure with binary relations, which can be represented using single bits. We compress the rows of B such that 32 (64) columns are represented using a single integer (long integer) similar to the color compression we used in [17]. In this scheme, each column is represented with 2 integers (or possibly with long integer). The first integer refers to "column set" (CS) in which the set bit indices denote the indices of existing columns. That is, if the i^{th} bit in CS is 1, the row has a nonzero entry at the i^{th} column. The second integer refers to the column set index (CSI) to represent more than 32 columns. The reduction on the size of nonzeros of B can be up to 32×. The compression is more successful if the column indices in each row are packed close to each other. This reduces the problem size, allows faster row-union operations using BITWISEOR, and makes the symbolic phase more efficient. It also reduces the calculated row max flops, as a result of the reduction in the row lengths of B. This reduces the memory requirements of the symbolic phase. For example, the size of the dense accumulators are reduced from $numCols(B)$ to $\frac{numCols(B)}{32}$.

B. SpGEMM Variants

We implemented 4 SpGEMM methods that follow Algorithm 2, but differ in the used accumulator type. First variant is referred as KKMEM. It uses sparse hashmap accumulators and memory pools that allow it to be portable and thread scalable. The second variant, KKDENSE, adopts dense accumulators per thread. This method works only on CPUs since the memory requirement is not feasible on GPUs. The other 2 variants, KKMCR and KKMCW, base on distance-2 graph coloring so that threads share a dense accumulators. KKMEM is our main method in this paper, while the others are implemented for comparison and analysis reasons.

1) KKMEM: Uses sparse hash map accumulators to accumulate floating point values in numeric, and "column sets" in symbolic. It also makes use of a "portable memory pool" to allocate memory for L2 accumulators.

Memory Pool: A portable memory pool is used in Algorithm 2 to allocate memory for L2 accumulators. It is allocated and initialized before the kernel call, and it consists of num_chunks many memory chunks, where each has fixed size of $chunk\_size$. $chunk\_size$ is chosen based on the max (compressed) FLOPS in a row, and max length of a row of C in the symbolic phase and numeric phases, respectively. num_chunks is chosen based on the available concurrency. It is the number of threads on CPUs, while it is the maximum number of kokkos-threads that can run on GPUs ($\frac{131,062}{vectorlanesize}$ on K80). We use an upper bound for the maximum allocated memory for the pool (such as $O(nnz(C))$), we reduce the num_chunks if the memory allocation becomes too expensive on GPUs. The memory pool has 2 modes such as ONETHREAD2ONECHUNK and MANYTHREAD2MANYCHUNK. On CPUs, the mode of the memory pool is set to be one to one, while on GPUs we set it to many to one.

Allocate function of the memory pool requires thread indices. These indices assists the look up for a free chunk. Pool returns the chunk with the given thread index on CPUs, while, on GPUs, it starts a scan from the given thread-index until an available chunk is found. This allows CPU threads to reuse local numa memory regions. It also helps the scan of GPU threads as they start their scan from different indices. Allocate function locks the returned memory chunk, and this lock is released when the thread calls the release function of the memory pool.

HashMap Accumulator: The hashmap accumulator follows the parallel version of the hashmap we used in [18]. It consists of 4 parallel arrays as shown in Figure 2, which shows an example of a hashmap that has a capacity of 8 hash entries and 5 (key, value) pairs. Hashmap is stored in linked list structure. Ids and $Values$ stores the (key, value) pairs. $Begins$ holds the beginning indices of the linked lists corresponding to the hash values, and $Nexts$ holds the indices of the next elements within the linked list. For example, in the figure, the set of keys that have hash value of 4 are stored with a linked list. The beginning index of this linked list is stored at $Begins[4]$. We use this index to retrieve (key,value) pairs ($Ids[0]$, $Values[0]$). Then, link list is traversed using the $Nexts$ arrays. An index value -1 corresponds to the end of the linked list for the hash value. We choose the size of $Begins$ to be power of 2, therefore hash values can be calculated using BITWISEAND, instead of slow modular (%) operation. The insertions to the hashmap is done via a vectorInsert operation. Each vector lane calculates the hash values, and travels through their linked lists. If a key already exists in the hashmap, values are accumulated with "add" and BITWISEOR in numeric, and symbolic phases, respectively. We exploit the fact that vector insertions at a time are duplicate-free. That is, vectorlanes read a single row of B, and the rows of a matrix always have different columns (keys). Thus, atomic operations are not needed for accumulation. If the key does

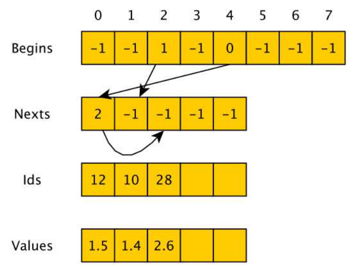


Fig. 2: Structure of the hashmap accumulator with 5 entries and 8 hash values.

not exist in the hashmap, vectorlanes reserve the next available index in Ids and $Values$ with an atomic counter. They set the $Begins$ of the correspond hash value to their insertion index with atomic_compare_and_swap, and the $Next$ of the inserted index is set to old beginning index. If hashmap runs out of memory, it returns "FULL" and the failed vector lanes insert their values to the next level hashmap.

The hashmap accumulator guarantees compact/packed insertions to Ids and $Values$ arrays. L2 accumulator uses the actual column and value arrays of C for Ids and $Values$ in numeric phase. Thus, we avoid copies of L2 hashmap values, and reduce the memory requirement. The size L2 hash map in the symbolic phase can be at most $mcf \times 5$ (1 for $Nexts$, Ids, $Values$, and at most 2 for $Begins$) for symbolic phase, where mcf denotes the maximum number of flops for a row. We need at most $2\times$ size for $Begins$ since we round the size of $Begins$ to powers of 2. Numeric phase L2 hashmap accumulator requires $Begins$ and $Next$ with a sizes at most $2 \times maxRow$, $maxRow$. It uses the result matrices column and value arrays for Ids and $Values$. $chunk\_size$ for the memory pool is chosen based on these computations.

2) KKDENSE: Algorithm follows the same thread-partitioning with KKMEM. Instead of hashmap accumulators, it uses single level dense accumulators per thread. It requires 3 parallel arrays, each has size of $O(cols(C))$. First one is used for floating point values, and they are initialized with 0's. Another parallel array is used to hold the indices that are inserted to dense accumulator. We also use another boolean array as a marker array whether a column index is inserted before or not. Because of the high memory requirements KKDENSE is not suitable for GPUs, thus it is not portable. This method is similar to [3] without $2D$ partitioning.

3) KKMCR and KKMCW: The major issue with KKMEM is the use of the sparse accumulators, which increases the comparison operations. Hashmaps reduce the number of comparisons, however they do not guarantee a constant access time. KKDENSE does not have this issue, however it demands very high amount of memory. Instead, in order to allow the shared usage of dense accumulators, we perform distance-2 graph coloring on C in the symbolic phase as shown in Figure 3. We find set of the rows of C that do not share any column. To do this, we calculate not only the size of the result matrix, but also the structure and nonzeroes of C (using the compressed B) in symbolic phase. Then, a dense accumulator can be shared among the threads by processing the rows with the same color at a time.

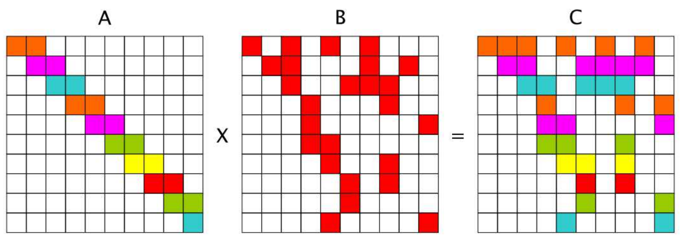


Fig. 3: Distance-2 graph coloring of C with 6 colors. The rows with the same colors do not share any columns.

TABLE III: Summary of the KK SpGEMM methods. All variants does hierarchical partitioning and compression. They use different type of accumulators.

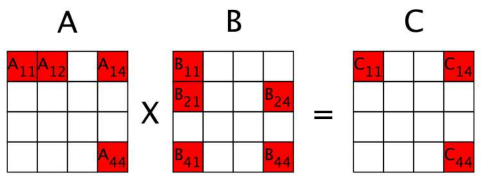
	Hier.	Comp.	Symbolic	Accumulator	# Comparison	# reads	#writes
KKMEM	✓	✓	#nnz	Sparse	≥ 1 per flop	ok	low
KKDENSE	✓	✓	#nnz	Dense per thread	1 per flop	ok	high
KKMCR	✓	✓	#nnz and structure	Dense per color	0	high	highest
KKMCW	✓	✓	#nnz and structure	Dense per color	0	higher	higher

Coloring on C is more restrictive coloring than A. That is, a row of C is found by merging the rows of B that correspond to the columns of A for that row. As a result, a distance-2 coloring on C is also a distance-2 coloring on A as shown in the figure. Similarly, the rows that have the same color do not share any columns on A, and they do not share any row of B during their accumulations. This reduces the data locality for B accesses, as each row of B is accessed at most once for the set of rows with the same color. We perform multiplications for multiple colors (as many as num_multi_color) at a time to improve this data locality. This requires num_multi_color many dense accumulator. We choose as num_multi_color as $\frac{nnz(C)}{cols(C)}$ so that we do not use more than the output size for the accumulators. Then, the first variant, MCR, permutes the rows that are in the same $multi\_color$ step, therefore, consecutive rows might have different colors. This will improve the reuse of B, however, consecutive rows with different colors will require the use different dense accumulator. Therefore, write locality is not preserved for this variant. The second variant, MCW, permutes the rows within the same color. Consecutive rows have the same color, and same accumulator is used for them. This case is similar to single color case with reduced synchronization cost..

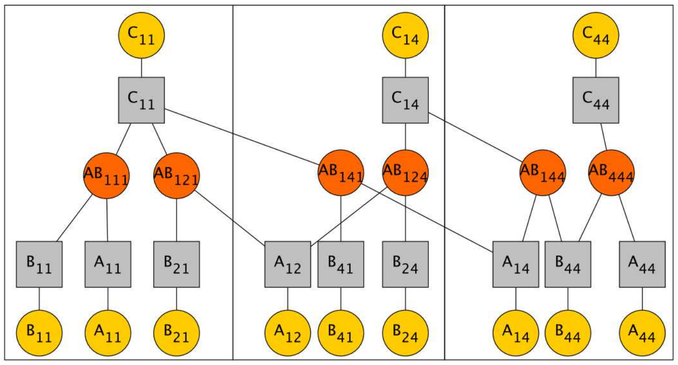
Coloring variants calculate the structure of C in the symbolic phase. As a result, they do not require parallel accumulator arrays and comparison operations in their numeric phases. We use a sequential distance-2 coloring, we only study the numeric part for these algorithms. Table III gives the overall summary of the KK SpGEMM variants.

IV. Hypergraph Model to Evaluate Memory Accesses

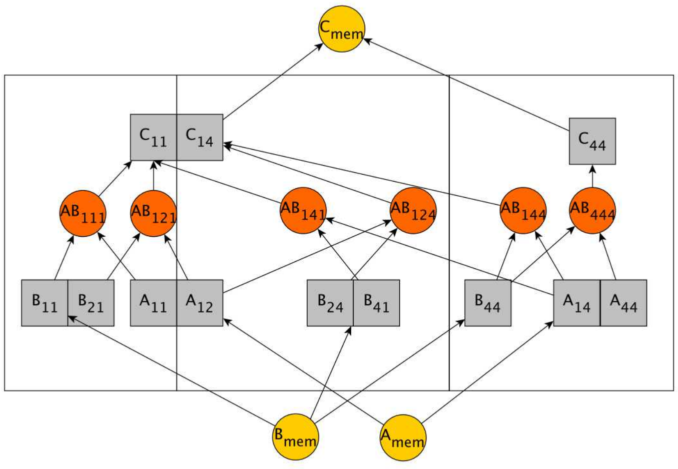
In this section we extend the hypergraph model for distributed matrix-matrix multiplication proposed in [15] with the


A
B
C
A_{11} A_{12} A_{14} A_{44}
X
B_{11} B_{21} B_{24} B_{41} B_{44}
=
C_{11} C_{14} C_{44}

(a) Simple Matrix Multiplication


C_{11} C_{14} C_{44}
AB_{111} AB_{121} AB_{141} AB_{124} AB_{144} AB_{444}
B_{11} A_{11} B_{21} A_{12} B_{41} B_{24} A_{14} B_{44} A_{44}

(b) Hypergraph Model


C_{mem}
C_{11} C_{14} C_{44}
AB_{111} AB_{121} AB_{141} AB_{124} AB_{144} AB_{444}
B_{11} B_{21} A_{11} A_{12} B_{24} B_{41} B_{44} A_{14} A_{44}
B_{mem} A_{mem}

(c) Directed Hypergraph Model for Shared Memory

Fig. 4: Hypergraph Models for the SpGEMM computation. Yellow vertices correspond to the nonzeroes of the input/output matrices where the row and

directed hypergraph model from [19] for shared memories. A hypergraph $H = (V, N)$ is defined as a set of vertices V and a set of nets (hyperedges) N among those vertices. A net $n \in N$ is a subset of vertices and represents multiway dependency between them. Vertices can be associated with computation weights and nets can be associated with communication costs (refer to section 2 of [19] for hypergraph details and communication metrics). Figure 4a shows a simple matrix multiplication example, and Figure 4b presents the hypergraph model of this matrix from [15]. The hypergraph model represents each multiplication of SpGEMM with a vertex (red circles). Nonzeroes of the input/output matrices are represented with a net (squares) and a vertex (yellow circles). Nets corresponding to the nonzeroes of A and B are connected to the multiplications that require them, while nets for the nonzeroes of C are connected to the multiplications that contributes to its value. Each red vertex is associated with a weight that represents the computation costs $w_1(v_{red}) = 1$, while each yellow vertex is associated with a weight to represent its memory requirement $w_2(v_{yellow}) = 1$. Then, a multi-criteria partitioning that reduces overall communication is performed using PaToH [20]. Multi-criteria partitioning balances both computation (red vertices) and memory requirements (yellow vertices). In this example, we have 3 parts, where each part is assigned 2 computation vertices and 4 data vertices. Overall communication volume is 4, because of the cut-nets $\{C_{11}, C_{14}, A_{12}, A_{14}\}$.

In this work, we slightly extend this hypergraph model with the directed hypergraph model. Directed hypergraphs also model the flow of the data within the nets, i.e., nets have senders and receivers. In the first step, we add directions to the nets. A and B vertices are the senders of their corresponding nets, and multiplications are the receivers (not shown). On the other hand, multiplications are the senders of the C nets, while C vertices are the receivers of them. With this hypergraph model and a directed hypergraph partitioner such as UMPa [19], one can model and reduce not only overall communication volume, but also other communication overheads such as the number of messages exchanged, or maximum communication volume. For example, with the addition of the directions as described, first part has a receive volume of 2 because of nets $\{C_{12}, A_{12}\}$, second part has send volume of 2 because of nets $\{C_{12}, A_{12}\}$ and receive volume of 2 because of nets $\{C_{14}, A_{14}\}$, and third part has send volume of 2. Moreover, first and second part receives a single message, while second and third part sends a single message.

We further modify the model for shared memory systems. On a shared memory partitioning, there is not a real "data" partitioning, we only partition the computations, since computation units do not own data. We do not have point to point communications between processors, and insetead of the communication metrics such as $Send/Receive$, we have metrics that correspond to $Write/Read$ to memory. Moreover, memory systems pack multiple data into single cache line. That is, when a data is accessed, the whole cache line is read (written) from the memory on CPUs so that the consecutive memory reads become cheaper. Similarly, the memory accesses on GPUs simultanously load multiple data that lie within the coalesced access range. Therefore, in Figure 4c, we pack the nets for the consecutive data elements that fall into a cacheline or a coalesced memory accesses (2 elements are packed in the example). Moreover, the data elements are not partitioned anymore, they are fixed in the memory (to a virtual part $k+1$). One can assign different cost for $Reads/Writes$ nets of A, B and C if they are in different memory spaces. UMPa can be used to reduce the $Read/Write$ costs of each thread with some modifications. However, this paper uses this model to evaluate the read/write costs of the SpGEMM methods rather than finding and optimized partitioning. Given an execution order, we stream the nets of this hypergraph into a cache, and

TABLE IV: Multigrid Matrices used in the experiments

		#row	#col	#nnz
Laplace	A	$15,625,000$	$15,625,000$	$109,000,000$
	P	$15,625,000$	$1,969,824$	$57,354,176$
Brick	A	$15,625,000$	$15,625,000$	$418,508,992$
	P	$15,625,000$	$592,704$	$71,991,296$
Empire	A	$2,160,000$	$2,160,000$	$303,264,000$
	P	$2,160,000$	$8,800$	$8,572,251$

simulate the read/write cache misses.

V. Experiments

a) Architecture:: We evaluate the performance of the proposed SpGEMM kernels on single nodes of the Bowman and Shannon clusters at Sandia. Each node in Bowman has a KNL processor with 68 1.40GHz cores with 4 hyper-threads per core. Each node has 16 GB MCDRAM with 460 GB/s peak bandwidth and 96 GB DDR4 memory with 102 GB/s peak bandwidth. The nodes can have various modes. In our experiments we use "Quadrant" mode, that has uniform memory accesses between the cores without any NUMA effects. Shannon is a GPUs cluster with two NVIDIA Tesla K80 GPUs in a node with compute capability 3.7 and 11.25 GB of global memory. Our SpGEMM variants are implemented using the Kokkos library, and compiled using the version within the Trilinos 12.2 release, with icc 17.0.042 on Bowman, gcc 4.7.2 with Cuda 7.5.7 on Shannon. Each run reported in this paper is the average of 5 executions. We used double precision for the floating points, and 32 bit integers for the index types.

We evaluate matrix multiplications in the forms of $P^T \times A \times P$ and $A \times A$ depending on the application domain. The experiments use matrices from various multigrid problems (Table IV) and UF sparse matrices [21] used in the literature for $A \times A$ from [6], [13] (Table V). The experiments are organized in three parts. First, we compare KKMEM with SpGEMM implementations in cuSPARSE [5], CUSP [8], bhSPARSE [6], and ViennaCL (CUDA Implementation) [9] on GPUs (Section V-A). Second, we compare KKMEM to Intel Math Kernel Library (MKL) [4] and ViennaCL on KNLs (Section V-B) and study the practical use case of reusing the symbolic structure. Finally, Section V-C analyzes the memory accesses of the four KK variants based on our model and evaluates the performance of the numeric phases in MCDRAM vs DDR memory.

A. GPU Experiments

Table V shows the achieved GFLOPs/sec for each method on 23 different matrix multiplications. cuSPARSE and KKMEM are the most robust methods than can multiply all the matrices. However, cuSPARSE is the slowest of all the methods compared here. ViennaCL, bhSPARSE and CUSP run out of memory for 4, 8 and 18 multiplications, respectively. The bottom 4 rows in the table show the speedup of the algorithms w.r.t cuSPARSE for the matrices the method could run on. KKMEM, bhSPARSE and ViennaCL obtain the best performance on 19, 3, and 1 matrices, respectively. In general

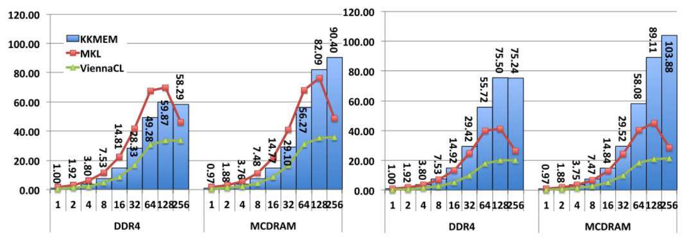
TABLE V: GFLOPs/second of SpGEMM variants in K80 GPUs. Best method for each multiply is in bold. The (#rows, #non-zeros) of the UFL matrices are given in parenthesis. Blank space indicate the method ran out of memory.

	cusp	bh	vienna	cusparse	kkmem
2cubes_sphere ($102K$,$1.6M$)	0.347	0.627	0.693	0.208	**1.059**
cage12 ($130K$, $2M$)	0.338	0.759	0.623	0.191	**0.881**
filter3Dp ($106K$,$2.7M$)	0.459	**1.175**	1.013	0.246	1.165
offshore ($259K$,$4.2M$)	0.343	0.868	0.849	0.130	**1.266**
webbase ($1M$, $3M$)	0.346	**0.385**	0.040	0.120	0.265
cant ($62K$,$4M$)		2.382	1.542	2.268	**3.071**
hood ($221K$,$10M$)		3.249	1.557	1.127	**3.509**
ldoor ($952K$,$42M$)		3.207	1.499	1.169	**3.662**
pwtk ($106K$,$11M$)		3.191	1.668	1.641	**4.080**
laplace_RA		**0.916**	0.634	0.159	0.877
laplace_RA_P		0.400	0.774	0.158	**0.683**
laplace_AP		0.789	1.377	0.101	**1.529**
laplace_R_AP		0.681	0.789	0.231	**1.490**
empire_RA		1.312	1.039	1.628	**2.086**
empire_RA_P		0.344	**1.413**	0.527	0.927
empire_AP			2.062	0.661	**2.408**
empire_R_AP			1.844	0.730	**1.957**
brick_AP			1.939	0.296	**2.310**
brick_R_AP			1.024	0.445	**1.810**
brick_RA				0.322	**1.904**
brick_RA_P				0.394	**0.888**
audi ($943K$,$77M$)				1.270	**3.318**
bump ($2.9M$,$127M$)				1.211	**2.988**
Speedup[1-5]:	2.112	4.149	2.507	1.000	**4.746**
Speedup[1-15]:		2.730	2.247	1.000	**3.753**
Speedup[1-19]:			2.436	1.000	**3.841**
Speedup[1-23]:				1.000	**3.689**

the fastest algorithm is KKMEM, which is on average 3.689, 1.577, 1.375 and 2.247 faster than cuSPARSE, ViennaCL, BhSPARSE, and CUSP, respectively on the sets each method could complete. The speedups of KKMEM w.r.t. cuSPARSE ranges from 1.28 to 15.103. KKMEM achieves the best performance without sacrificing robustness mainly due to the hierarchical algorithm that maps well to the GPUs and the thread scalable data structures that can effectively use the GPU hardware features.

B. KNL experiments

In this subsection, we compare KKMEM against ViennaCL (OpenMP implementation) and Intel MKL on KNLs. We use `mkl_sparse_spmm` routine in MKL's inspector-executor sparse BLAS. The runtime for this method takes up to $2-3x$ more time when it is called for the first time within an executable than the following calls. Even though an application using MKL will observe this poor performance, we exclude the first run, and report the following runs for this function. As a result, it should be noted the speedup numbers compared to MKL in this section are conservative. The test dataset includes 12 multigrid multiplications from Table IV, and 2 $A \times A$ multiplications for the biggest two matrices from our dataset - Audi and Bump. For these experiments, we only used 64 of 68 cores in the KNL node, with 2 and 4 hyperthreads on 128 and 256 threads. Figure 5a shows the *geometric mean of the normalized speedups* (w.r.t. sequential KKMEM on DDR4) of three methods for 14 multiplications



(a) Speedups for NoReuse case (b) Speedups for Reuse case

Fig. 5: Strong scaling speedups on DDR4 and MCDRAM for geometric mean of 14 multiplications w.r.t. sequential KKMEM on DDR4. The numbers above the bars correspond to KKMEM speedups.

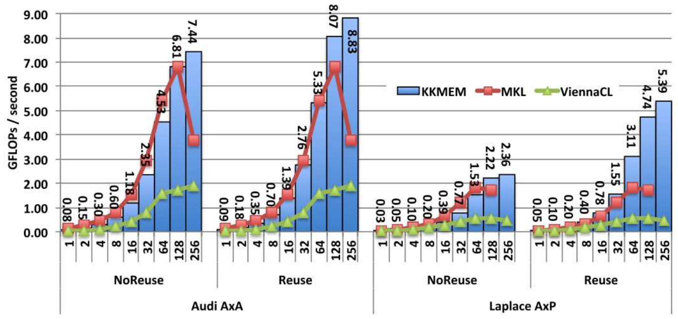


Fig. 6: Strong scaling for Audi and Laplace on MCDRAM

of NoReuse case. MKL fails (does not complete in 1000 seconds) for $4/14$ instances with 256 threads, for which we proportionally scale the speedup of the dataset that it runs so we can find a geometric mean (assuming that $\frac{Speedup_{(14,256)}}{Speedup_{(14,128)}} = \frac{Speedup_{(10,256)}}{Speedup_{(10,128)}}$). "NoReuse" refers to the case where the matrix multiplication requires both symbolic and numeric. We also run the experiments with all three matrices stored in either of the two different memory spaces DDR4 or MCDRAM. This is the first such experiment to compare the performance of on-chip vs off-chip memory for a complex kernel like SpGEMM in this latest architecture.

First, it is easy to see MKL does not scale on 256 threads and it fails for 4 instances, however, its performance is better than KKMEM upto 128 thread on DDR4 memory. On average, MKL is 69% faster than KKMEM on single thread, as the number of threads increase the difference reduces to 17% on 128 threads, and then KKMEM becomes faster on 256 threads for matrices MKL can complete. On MCDRAM, the performances curves are similar to DDR4 on smaller number of threads. However, KKMEM becomes 7% faster than MKL on 128 threads. The performance of ViennaCL is lower than both KKMEM and MKL. KKMEM is up to 78% and $2.51x$ faster than ViennaCL on DDR4 and MCDRAM, respectively. KKMEM scales close to linear up to 64 threads, for which it obtains 49.28 (56.27) speedup on DDR4 (MCDRAM). It also scales well with the addition of hyperthreads. The addition of 2 (4) hyperthreads speedups the KKMEM by 21% and 46% (18% and 61%) w.r.t. its performance with 64 threads on DDR4 and MCDRAM, respectively. Storing matrices in the MCDRAM paying good dividends for larger thread counts as the larger bandwidth can be effectively utilized. This is reflected in the $90x$ speedup for MCDRAM as opposed to the $58x$ speedup for the DDR4 when using 256 threads. However, when the bandwidth is not saturated on lower number of threads, DDR4 is slightly faster.

a) Symbolic Reuse and Compression: The other use case that is common in applications is the reuse of the symbolic structure of the matrices for several SpGEMM operations. This is not supported in MKL. KKMEM allows the symbolic phase to be reused and run just the numeric phase in the "Reuse" case. Figure 5b compares the performance of the three codes for this case. When there is a reuse of the symbolic structure, MKL and KKMEM have similar performances on single thread. As the number of threads increase to 128 (256), KKMEM gets upto $2x(3x)$ faster than MKL. Note that, ViennaCL also runs in two phases, similar to KKMEM. First it calculates the memory requirements than it performs the actual flops. However, the public interface does not support the 2-phase usage. Hence we do not reuse the symbolic structure when running ViennaCL. It is slower than both MKL and KKMEM. Note that the effect MCDRAM can be best observed here where we see an impressive $90x$ *geometric mean speedup on 128 threads* for a memory bound kernel such as SpGEMM.

Figure 5 shows significant performance differences between NoReuse and Reuse because in most of the multiplications, symbolic phase is as expensive as numeric phase. This is because the compression scheme was not successful at compressing the matrices to smaller sizes on multigrid matrices that dominate the dateset. It is more successful on the matrices from UF matrix collection for $A \times A$, since they the columns of those matrices are usually packed. To demonstrate this difference we compare two matrices that have different compression ratios. Figure 6 gives the strong scaling results with the achieved GFLOPs/second for Audi $A \times A$ and Laplace $A \times P$ multiplications. MKL fails to complete in 1000 seconds on Laplace with 256 threads, therefore it is excluded. The compression on Audi reduces the size of the matrix by 88%. Therefore symbolic phase becomes significantly cheaper than numeric phase, and the performance difference between Reuse and NoReuse case is *not* as significant. On the other hand, the compression is not successful on Laplace $A \times P$. It reduces the size of P on Laplace $A \times P$ multiplication only by 7%. As a result, the performance difference between the Reuse vs NoReuse case is high for Laplace AP. Different ordering schemes can be applied to improve packing. We leave the study of the ordering methods that would pack columns as future work.

Figure 7 gives the best GFLOPs/sec achieved by each method among all thread counts. Best GFLOPs/sec ratios are achieved with 128 threads for MKL, and 128 and 256 for KKMEM and ViennaCL. KKMEM-Reuse lists the perfor-

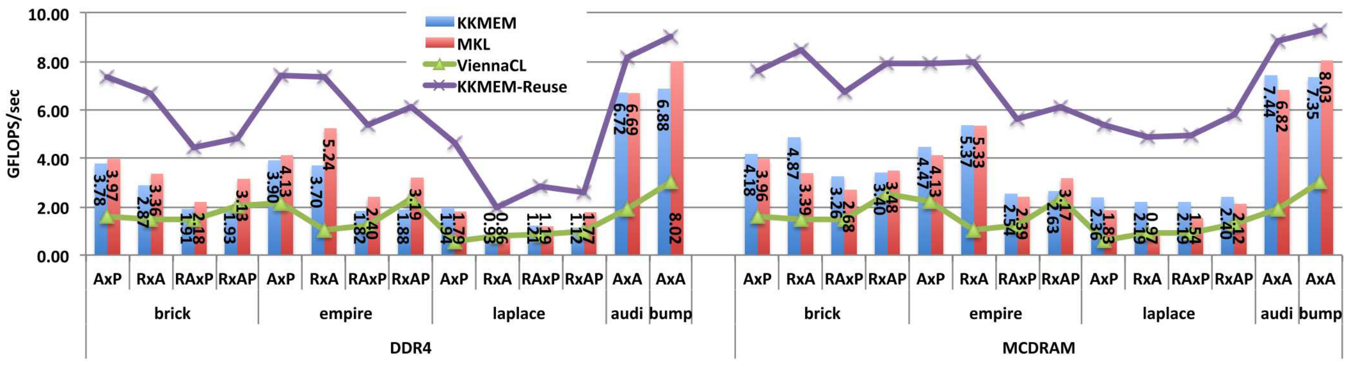


Fig. 7: Best GFLOPs achieved by each method for a given problem. The number of threads differ for different methods.

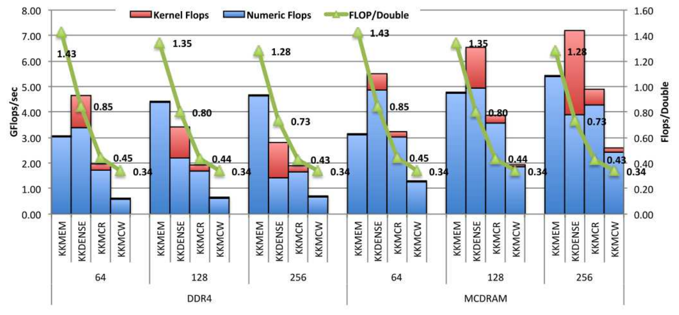


Fig. 8: The number of floating point operations performed per double access and GFLOPs/sec achieved by the methods. The numbers highlights the FLOP/Double ratios.

mance of KKMEM when the symbolic structure is reused. It obtains the best performance compared to any other method in all multiplications. ViennaCL performs worse than other methods. For the NoReuse case, MKL and KKMEM obtains best performances on 10 and 4 multiplications on DDR4. However, on MCDRAM, KKMEM is better than MKL on 11 matrices, while for 3 of them MKL achieves higher GFLOPs. The geometric mean of the best flops achieved by MKL 16% better than KKMEM on DDR4, while it is 16% worse on MCDRAM. Again the above discussed numbers are with "No Reuse" case. KKMEM clearly wins when "Reuse" is needed. On Average, KKMEM obtains $2.22x$ better performance than MKL with Reuse on MCDRAM.

Another mode of KNL is to use MCDRAM as cache. We have experimented the performance of SPGEMM methods on the Cache mode as well, however, since all experiments fit into MCDRAM, we obtain the same performance results as running them on MCDRAM in that case.

C. *Performance analysis using the model for memory accesses*

In this set of experiments, we use the hypergraph model to evaluate the memory accesses of each KK variant algorithm (KKMEM, KKDENSE, KKMCR, KKMCW), and compare their performances on the **numeric phase**. Exeuction time for symbolic phase or the time for preprocessing such as distance-2 coloring is excluded as we are interested in studying the numeric kernel. We simulate the cache behavior using the streamed execution order and the hypergraph model. We use cache size as $32KB$ which is the size of L1 cache in KNL, and cache line size as 8 doubles. For each variant, using the hypergraph model and given cache size, we calculated the number of memory reads and writes based on the L1 cache misses. The simulation primarily uses the execution order and the data structure used for "writes" which is either sparse or dense accumulators.

Figure 8 lists the calculated floating point operation per memory access and the performances for each algorithm on Laplace $A \times P$ multiplication. As before, the x axis is first divided based on the memory used, and then the number of threads ranging from 64 to 256. We only study the case where all cores are used, since memory bandwidth is not saturated when number of threads is less than 64. The green line refers to how many floating point operation is performed per double read, for which the y axis is given on the right side. The flops is the actual flops in the kernel and the memory access for the double is found using the model. This number is expected to be high for a method that exploits locality, while it would be lower for methods with more random memory accesses. This ratio drops for all methods as the number of hyperthreads increase within a core. Hyperthreads share L1 cache, therefore this reduces the amount of L1 cache per thread. KKMEM has better FLOP/Double ratios than KKDENSE. This is because KKMEM uses sparse accumulators which exploits the data locality since the the sparse accumulator is a compact data structure. Coloring variants have the worst FLOP/Double ratios, since they also have minimal reuse of B reads. Moreover, even though they use dense accumulators (as KKSPEED), they do more writes than KKDENSE since the consecutive rows are unlikely to share any columns (due to different colors).

Figure 8 also shows the GFLOPS for the four variants measured in two ways stacked on one another. Note that the numeric phase itself has two main parts. In the first part, it allocates and initializes the accumulators, and in the second part it performs the actual multiplication kernel. In the figure, we list GFLOPS in two separate bars. "Numeric flops" is the flops calculated based on both steps of numeric phase, while "Kernel flops" are calculated based only on the time of

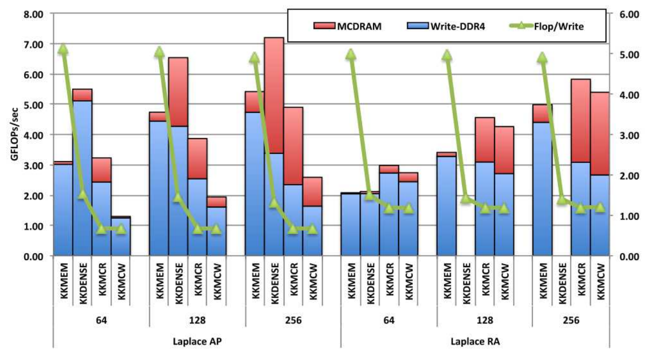


Fig. 9: The number of floating point operations performed per double write and GFLOPs/sec achieved by the methods on Laplace $A \times P$ and Laplace $R \times A$ matrices. The numbers highlights the FLOP/Double Write ratios.

the multiplication kernel, excluding the time of allocation and initialization of the accumulators. This is chosen since allocation/initialization becomes expensive for dense accumulators, and it interferes with the analysis, as our model's memory/read write analysis only includes the multiplication kernel. That is, KKMEM becomes usually faster than the other variants because of the low initialization cost, but for the rest of this section, we only consider the time for the multiplication kernels.

When the methods run on the DDR4, lower bandwidth memory, the performance of the multiplication kernels follows similar trends with the FLOP/double. KKMEM is usually the fastest among the kernels, even though it has extra comparison operations. On 64 threads, KKDENSE is the fastest, for which the bandwidth of DDR4 is enough to deliver the data needed by 64 threads. However, as the number of threads increase, the memory accesses become more significant and the methods with dense accumulator do not scale. This is despite the extra comparison operations required by KKMEM. On the other hand, when they are run on high bandwidth memory, memory is less likely to become bottleneck. KKDENSE has the fastest kernel in all thread counts in MCDRAM. KKMEM is still faster than coloring variants as they demand much higher memory access than KKDENSE. The performance difference of the algorithm between MCDRAM and DDR4 memory is much lower for KKMEM, since it is more robust to bandwidth related overheads.

Figure 9 shows the performances of the KK variants (multiplication kernel only) for 2 different multiplications. In this kernel, accumulated bars correspond to the performancess when all the matrices and accumulators lies in MCDRAM. The blue bars show the performance of the algorithms when the result matrix and accumulators (memory writes) lies in DDR4 while A and B are located in MCDRAM. Green line shows the number of flops performed per write access. As it uses sparse accumulators, KKMEM has the best FLOP/write ratio. KKDENSE has better writes then the coloring variants, and KKMCW has slightly better writes than KKMCR.

On Laplace $A \times P$, KKDENSE has the best performance when the writes are to MCDRAM despite having more writes and poor FLOPS/write ratio than KKMEM. However, its performance drops dramatically when the accumulators are allocated in DDR4. KKMCR and KKMCW are also affected significantly, but they do poorly than KKMEM even in MCDRAM. KKMEM is the most robust to where the writes are as predicted by its FLOP/write ratio. The fastest method KKDENSE is outperformed by KKMEM when only writes are placed to DDR4, since KKMEM exploits data reuse thanks to its sparse accumulators. Similarly, right side of Figure 9 shows the performance changes on Laplace $R \times A$ multiplications. KKDENSE does not run on higher number of threads, since it runs out of memory. KKMCR and KKMCW is faster than KKMEM in this multiplication, however, they are significantly affected by the memory that writes are performed, and they are outperformed by KKMEM when writes are placed to slow memory. These results demonstate the accuracy of our models in predicting the performance of different SpGEMM variants in different memories and the robustness of KKMEM both in terms of lower memory usage and as a consequence of that a lower impact on using either MCDRAM or DDR4.

VI. Conclusion and Future Work

In this work, we have studied portable implementation of SPGEMM kernel for massively threaded architectures. We have designed a hieararchical, memory-efficient algorithm, and portable thread-scalable data structures. We have demonstrated the superior performance of our method against 4 existing methods on GPUs, and 2 existing methods on KNLs. We have proposed a compression method that compresses the input matrices and speedups the structure prediction part of SPGEMM. We have also studied the practical usage of the SPGEMM kernel, where the our method exploits the reuse of the symbolic structure and obtains significant speedups w.r.t. existing methods. We have also developed a hypergraph model in order to measure the memory accesses of the algorithms, and shown the relationship of the measured performance with the expected performance on various memory spaces.

There are various branches we would like to extend the current work. Firstly, packing methods that would reorder the columns of a matrix is planned to be studied. This is different than bandwidth-minimization algorithms since the target is to reduce the pairwise distances of the columns within a row, so that the compression ratio can be increased. Moreover this packing can be usefull for common problems such as sparse matrix-vector multiplications, in which column indices are packed within a cache line on CPUs or in a range of coalesced memory access on GPUs. Secondly, we would like to study the ordering methods that will reorder the rows of A, so that the reuse for second matrix B is maximized. There exists ordering methods in the literature based on (hyper) graph partitioning, however they are usually expensive and the cost of them usually does not amortize the cost. We would like to study fast ordering methods that exploits the data reuse. Thirdly, we would like to extend the parallelism within the calculation of a row. That is, current hierarchical algorithm that assigns rows to kokkos-threads will be extended to assign rows to kokkos-teams. An analytical study to compare both approaches is planned to be studied.

Acknowledgements: We thank Erik Boman, Karen Devine, Grey Ballard, and Carter Edwards for helpful discussions, and test bed program at Sandia National Laboratories for supplying the hardware used in this paper. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Dept. of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000. This work is supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, and by the NNSA's Advanced Simulation and Computing (ASC) program.

REFERENCES

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J Parallel Distrib Comp*, vol. 74, no. 12, pp. 3202–3216, 2014.

[2] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[3] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *High Performance Computing*. Springer, 2015, pp. 48–57.

[4] Intel, "Intel math kernel library," 2007.

[5] J. Demouth, "Sparse matrix-matrix multiplication on the gpu," in *Proceedings of the GPU Technology Conference*, 2012.

[6] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 370–381.

[7] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015.

[8] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrixmatrix multiplication for the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, p. 25, 2015.

[9] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[10] E. Cohen, "Structure prediction and computation of sparse matrix products," *Journal of Combinatorial Optimization*, vol. 2, no. 4, pp. 307–332, 1998.

[11] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps *et al.*, "An overview of the trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[12] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, p. 1094342011403516, 2011.

[13] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *arXiv preprint arXiv:1510.00844*, 2015.

[14] K. Akbudak and C. Aykanat, "Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C568–C590, 2014.

[15] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *arXiv preprint arXiv:1603.05627*, 2016.

[16] M. MCCOURT, B. SMITH, and H. ZHANG, "Efficient sparse matrix-matrix products using colorings," *SIAM Journal on Matrix Analysis and Applications*, 2013.

[17] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 892–901.

[18] M. Deveci, K. Kaya, and U. V. Catalyurek, "Hypergraph sparsification and its application to partitioning," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 200–209.

[19] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek, "Hypergraph partitioning for multiple communication cost metrics: Model and methods," *Journal of Parallel and Distributed Computing*, vol. 77, pp. 69–83, 2015.

[20] Ü. Çatalyürek and C. Aykanat, "Patoh (partitioning tool for hypergraphs)," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1479–1487.

[21] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.