SAND2017-13017C

Performance-Portable Sparse Matrix-Matrix Multiplication for Many-Core Architectures

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 sparse matrix-sparse matrix multiplication (SPGEMM) kernel for many-core architectures. We approach the SPGEMM kernel from the perspectives of algorithm design and implementation, and its practical usage. First, we design a hierarchical, memory-efficient SPGEMM algorithm. We then design and implement thread scalable data structures that enable us to develop a portable SPGEMM implementation. We show that the method achieves performance portability on massively threaded architectures, namely Intel's Knights Landing processors (KNLs) and NVIDIA's Graphic Processing Units (GPUs), by comparing its performance to specialized implementations. Second, we study an important aspect of SPGEMM's usage in practice by reusing the structure of input matrices, and show speedups up to $3\times$ compared to the best specialized implementation on KNLs. We demonstrate that the portable method outperforms 4 native methods on 2 different GPU architectures (up to $17\times$ speedup), and it is highly thread scalable on KNLs, in which it obtains $101\times$ speedup on 256 threads.

I. INTRODUCTION

Modern supercomputer architectures are following two different paths using either the Intel's Knights Landing processors or NVIDIA's Graphic Processing Units. As a result, it is important to design algorithms that can perform well on both platforms. The focus on this problem has been around programming models to implement an algorithm on multiple architectures [1], [2]. We consider this problem from an algorithmic perspective to design a "performance-portable algorithm", an algorithm that can perform well on multiple architectures with similar accuracy and robustness. Another approach would be to consider different, architecture specific ("native") algorithms for every key kernel. This leads to the question "How much performance will be sacrificed for portability?". We address this question by comparing the implementation of our performance-portable algorithm to several native implementations.

We choose the sparse matrix-matrix multiply (SPGEMM) kernel as our benchmark for this study due to its importance in several applications. SPGEMM is a fundamental kernel that is used in various applications such as graph analytics and scientific computing, especially in the setup phase of multigrid solvers. The kernel has been studied extensively in the contexts of sequential [3], shared memory parallel [4], [5] and GPU [6], [7], [8], [9] implementations. There are native kernels available in different architectures [5], [7], [9], [10], [11] providing us with good comparison points.

This paper focuses on SPGEMM from the perspectives of algorithm design and implementation for performance portability and its practical usage. Specifically, we try to address the following questions:

- What are the performance critical design choices and data structures for a SPGEMM algorithm to map well to different architectures (thousands vs hundreds of threads, streaming multiprocessors vs lightweight cores, shared memory vs MCDRAM) ?
- How will the kernel serve the needs of real applications, when there is a reuse of the symbolic structure?

In addressing these questions we make the following contributions in this paper.

- We design two thread scalable data structures (a multilevel hashmap and memory pool) to achieve performance portability, and a graph compression technique to speedup symbolic factorization phase of SPGEMM.
- We design a hierarchical, thread-scalable SPGEMM algorithm and implement it using the Kokkos programming model [1]. Our implementation is available at `https://github.com/trilinos/Trilinos`.
- We evaluate the performance portability of our method on various platforms, including traditional CPUs, KNLs, and two different GPU architectures. We show that our method outperforms 4 native methods on 2 different GPU architectures, (up to $17\times$ speedup). We also show that it has better thread-scalablilty than native OpenMP methods on KNLs, where it obtains $101\times$ speedup on 256 threads w.r.t. its sequential run.
- We also present results for the practical case of matrix structure reuse, where the method is up to $3\times$ faster than best native OpenMP method.

The rest of the paper is organized as follows: Section II covers the background for SPGEMM. Our SPGEMM algorithm and related data structures are described in Section III. Finally, the performance comparisons that demonstrate the efficacy of our approach is given in Section IV.

II. BACKGROUND

Given matrices A of size $m \times n$ and B of size $n \times k$ SPGEMM finds the $m \times k$ matrix C s. t. $C = A \times B$. Multigrid solvers use triple products in their 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. SPGEMM is also widely used in the literature for various graph analytic problems.

In the literature, most parallel SPGEMM methods follow Gustavson's algorithm [3] (Algorithm 1). This algorithm iterates over rows of A in a 1D fashion (line 1) to compute all entries in the corresponding row of C. Each iteration of the second loop (line 2) *accumulates the intermediate values* of multiple columns within the row using an accumulator. The number of the necessary multiplications to perform this matrix multiplication is referred as f_m (there are f_m additions too) for the rest of the paper.

Algorithm 1 1D row-wise SPGEMM for $C = A \times B$. We use matlab notation, i.e., $C(i,:)$ and $C(:,i)$ refer to i^{th} row and column of C, respectively.

```
Require: Matrices $A$, $B$
1: for $i \leftarrow 0$ to $m-1$ do
2:     for $j \in A(i,:)$ do
3:         //accumulate partial row results
4:         $C(i,:) \leftarrow C(i,:) + A(i,j) \times B(j,:)$
```

Design Choices: There are three design choices that can be observed in Algorithm 1: (a) the partitioning needed for the iteration, (b) how to determine the size of C as it is not known ahead of time, and (c) the different choices for the accumulators. The key differences in past work are related to these three choices in addition to the target parallel programming model (distributed memory vs shared memory). This section gives a brief background of the different choices and what is explored in the literature with a summary of different methods listed in Table I.

Different partitioning schemes have been used in the past for SPGEMM. A 1D method [12] partitions C in single dimension, and each row (or column) is computed by a single execution unit. On the other hand, 2D [4], [13] and 3D [9], [14] methods assign each nonzero of C or each multiplication to a single execution unit, respectively. Hypergraph model [15], [16] based partitioning schemes have also been used in the past. 1D row-wise is the most popular choice for the scientific computing applications. Using other partitioning schemes just within SPGEMM will require reordering and maintaining a copy of one or both of the input matrices when used within a scientific computing application. There are also hierarchical algorithms where rows are assigned to first level of parallelism (blocks or warps), and the calculations within the rows are done using the second level parallelism [6], [10], [7]. In this work, *we use a hierarchical partitioning of the computation where the first level will do 1D partitioning and the second level will exploit further vector parallelism.*

The next design choice is to determine the size of C. Finding the structure of C is usually as expensive as finding C. There exists some work in the literature to estimate its structure [17]. However, it does not provide a robust upper bound. It also requires a generation of a random dense matrix, and a sparse matrix-dense matrix multiplication which is not significantly cheaper than calculating the exact size in practice. As a result, one-phase or two-phase methods are commonly used. One-phase methods rely either on finding a loose upper bound for the size of C or doing dynamic reallocations when needed. The former could result in over-allocation and the later is not feasible in GPUs. Two-phase methods use the structure of A and B to determine/estimate the structure of C (*symbolic* phase), before computing C in the second phase (*numeric* phase). They either find the size of the rows of C or the exact non-zero pattern of C [11], [6], [18], and allow reusing of the structure for different multiplies with the same structure. This is an important use case in scientific computing, where matrix structure stays the same with changing values. For example, Algorithm 2 presents an example of a nonlinear system solve. Given a mesh, a nonlinear solve consists of solving series of linear systems with the same non-zero pattern but different values. The structure of A^k remains the same, but the values are assembled in each iteration of k (line 6) based on the solution of the previous nonlinear system. Multigrid solvers usually exploit this in their reuse case [19]. Given the matrix A^k, the `solve` method in Algorithm 2 constructs a multigrid hierarchy of some number of levels ($maxl$). At each level l ($l = 1 \dots maxl$) the corresponding matrix is coarsened using a triple matrix product $A^k_{l+1} = R^k_l \times A^k_l \times P^k_l$. The structure of the prolongation (P^*_l) and restriction (R^*_l) operators as well as the linear system A^*_l stays same over all the nonlinear systems. Therefore, the symbolic phase of a two-phase SPGEMM method can be executed only once per level of the first linear system solve, and can be reused for the rest of the linear systems. In this work, *we use a two-phase approach, and we aim to speedup the symbolic phase using matrix compression.*

Algorithm 2 Nonlinear system solve

```
Require: $A$ representing the input mesh, $b$ right handside vector
1: //time step
2: for $timestep \in [0,n]$ do
3:     $X_0 \leftarrow$ initial guess
4:     //nonlinear solve
5:     for $k \in [0,...]$ until $X_0$ converges do
6:         $A^k \leftarrow$ assemble_matrix $(A, X_k)$ //linear matrix
7:         //calculate residual
8:         $r_k \leftarrow b - A^k \times X_k$
9:         //solve problem - using multigrid
10:        $\Delta_{X_k} \leftarrow solve(A^k, r_k)$
11:        //update the solution
12:        $X_{k+1} \leftarrow X_k + \Delta_{X_k}$
```

The third design choice is the data structure to use for the accumulators. Some algorithms use a dense data structure of size k. However, such thread private arrays are not scalable on massively threaded architectures. 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.*

Related Work: There are a number of distributed-memory algorithms for SPGEMM [13], [15], [16], [14], [12].

Table I: Summary of the SPGEMM literature. HM, ESC, and MS denotes Hashmap, Expand-Sort-Compress and Merge Sort, respectively.

	Partition	Phases	Parallelism	Accumulator
Gustavson	1D	1	Sequential	Dense
Trilinos [12]	1D	1	Dist.	Dense
CombBLAS [13]	2D	1	Dist.	Heap
Azad et. al. [14]	3D	1	Dist./Multicore	Heap
MKL1 [5] mkl_sparse_spmm	1D	1	Multicore	
MKL2 [5] mkl_dcsrmultcsr	1D	2	Multicore	
Patwary et. al. [4]	1D/2D	1	Multicore	Dense
ViennaCL - OMP [10]	1D	2	Multicore	MS
CUSP [9]	3D	1	GPU	ESC
cuSPARSE [11]		2	GPU	
AmgX [6]	Hier.	2	GPU	2-level HM
ViennaCL-Cuda [10] Gremse et. al. [8]	Hier.	2	GPU	MS
bhSPARSE [7]	Hier.	hybrid	GPU	Heap/MS/ESC
KokkosKernels	Hier.	2	Multicore/GPU	HM

Table II: Kokkos Hierarchy mapping to GPUs and KNLs/CPUs

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

Most of the multithreaded SPGEMM studies [4], [14], [10], [8], [5] are based on Gustavson's algorithm. They usually differ in the data structure used for row accumulation. Some use dense accumulators [4], others a heap with an assumption of sorted columns in B rows [14], or a sorted list with row merges [10], [8] or other sparse accumulators [5].

Most of the SPGEMM algorithms for GPUs are hierarchical. CUSP [9] uses a $3D$ algorithm where each multiplication is computed by a single thread and later accumulated with a sort operation (ESC). Such global expansion results in high memory requirements. AmgX [6] follows a hierarchical Gustavson algorithm. Each row is calculated by a single warp, and multiplications within a row are done by different threads of the warp. It uses 2-level hash map accumulators, and does not make any assumption on the order of the column indices. On the other hand, row merge algorithm [8] and its implementation in ViennaCL [10] use merge sorts for accumulations of the sorted rows. bhSPARSE [7] 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 chooses different accumulators based on the size of the row.

Table I lists the summary of the literature. The last line of the table lists our method, KKMEM in KokkosKernels package of Trilinos, and the choices we follow in this work.

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 optimizations for backends such as OpenMP and CUDA. We use Kokkos' features such as parallel_for, parallel_scan, atomics, and views (arrays), and the Kokkos thread hierarchy for hierarchical parallelism. Using Kokkos allows us to run the same code on the CPUs, KNLs and GPUs just compiled differently.

The kokkos-parallel hierarchy consists of teams, threads and vector lanes. Table II shows how these terms map to execution units on GPUs, KNLs and CPUs. The mapping is the same for KNLs and CPUs. A *team* in Kokkos handles a workset assigned to a group of threads sharing resources. On GPUs, a team is mapped to a thread block, which has access to a software managed L1 cache. A team on the KNLs can be the threads sharing either the DDR memory, or the L2/L1 cache. We use the hyperthreads that share an L1 cache as the team. There is no one-to-one mapping from kokkos-teams to number of execution units used. That is the number of teams even on the CPUs is much higher than the number of execution units. An execution unit is assigned various number of teams. A *kokkos-thread* within a team maps to a warp (half, quarter warp, etc.) on the GPUs and work on a single thread on the KNLs. A kokkos-thread uses multiple vector lanes. The *vector lanes* map to threads within a warp in the GPUs and the vectorization units on the KNLs. We will use the terms, kokkos-teams/teams, kokkos-threads/threads and vector lanes in the rest of the paper.

The portability provided by Kokkos comes with some overhead. For example, template meta-programming used heavily in the Kokkos model results in some of the current compilers failing to perform some optimizations. Typically, portable data structures in Kokkos have some small overheads as well. Portable methods also avoid exploiting any architecture-specific assumption or instructions. However, all these overheads can be overcome by careful algorithm design unless the problem sizes are very small.

III. Algorithm

Algorithm 3 Overall structure of KKMEM.

```
Require: Input matrices $A$, $B$ s.t. $C = A \times B$
1: //allocate row pointers of $C$
2: allocate $C_{\texttt{row\_pointers}}$
3: //symbolic phase
4: $B_c \leftarrow$ compress_matrix($B$)
5: $C_{\texttt{row\_pointers}} \leftarrow$ CORE_SPGEMM ($`symbolic', A, B_c$)
6: //allocate columns and values of $C$
7: allocate $C_{\texttt{columns}}$, $C_{\texttt{values}}$
8: //numeric phase
9: $C \leftarrow$ CORE_SPGEMM ($`numeric', A, B, C_{\texttt{row\_pointers}}$)
```

The overall structure of our SPGEMM method, KKMEM, is given in Algorithm 3. It follows a two-phase approach, in which the first (symbolic) phase computes the number of nonzeros in each row (lines $4-5$) of C, and second (numeric) phase (line 9) computes C. Both phases use the CORE_SPGEMM kernel with small changes to the input. The amount of work is typically doubled in such two-phase approaches. The main difference is that, the symbolic phase does not use matrix values, it does not perform floating point operations. We aim to improve this phase and reduce memory usage by performing compression of B (Line 4).

A. Core SPGEMM *Kernel*

The core SPGEMM kernel used by both symbolic and numeric phases follow a hierarchical, row-wise, algorithm. This core SPGEMM kernel is given in Algorithm 4. A simplified example of the kernel is shown in Figure 1. The

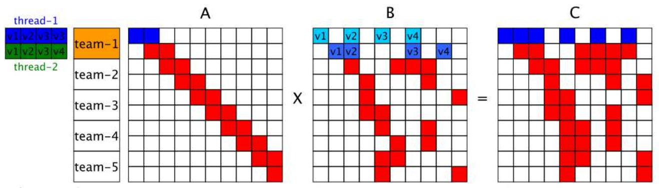


Figure 1: SPGEMM kokkos thread hierarchy. Team-1 consists of 2 kokkos-threads, and each kokkos-thread has 4 vector lanes (v[1-4]).

Algorithm 4 CORE_SPGEMM Kernel for $C = A \times B$. Colors show how the hierarchical algorithm maps to Algorithm 1. Red shows the first level, and green shows the second level of parallelism. Blue part is sequential within the first level parallelism.

```
Require: Phase, Matrices $A$, $B$, $C$
 1: // $B$ is a compressed/standard matrix if Phase
 2: // is symbolic/numeric. $C_{\texttt{row\_pointers}}/C$
 3: // allocated if Phase is symbolic/numeric.
 4: for each thread $\in$ thread_team assigned to subset of rows $C$ do
 5:     $i \leftarrow$ GETMYROW(thread_team, thread)
 6:     allocate the first level accumulator $\mathcal{L}_1$
 7:     for j $\in A(i,:)$ do
 8:         $cols, vals \leftarrow$ VECTORREAD$(B(j,:))$
 9:         $vals \leftarrow$ VECTORMULT$(vals, A(i,j))$
10:         if FULL $=\mathcal{L}_1$.VECTORINSERT$(cols, vals)$ then
11:             if $L2\_not\_allocated$ then
12:                 allocate the second level accumulator $\mathcal{L}_2$
13:                 $L2\_allocated \leftarrow True$
14:             $\mathcal{L}_2$.VECTORINSERT$(cols, vals)$
15:     if PHASE IS SYMBOLIC then
16:         $C_{\texttt{row\_pointers}}(i) \leftarrow$ total $\mathcal{L}_1/\mathcal{L}_2$ Acc sizes
17:     else
18:         $C(i,:) \leftarrow$ values from $\mathcal{L}_1/\mathcal{L}_2$ Acc
19:     if $L2\_allocated$ then
20:         release $\mathcal{L}_2$
```

performance of KKMEM derives from the hierarchical parallelism and two thread-scalable data structures, a memory pool and an accumulator. First, we focus on partitioning the work for hierarchical parallelism. The kernel achieves the first level of parallelism by assigning a set of rows of C to kokkos-teams (loop in Line 4). Each kokkos-thread within the team is assigned a subset of these rows. For example, the first two rows of C is assigned to team-1 (shown in orange in Figure 1). Each team has two threads in this example (highlighted in blue and green). Thread-1 of team-1 is assigned to the first row (highlighted in blue). These two levels of team and thread parallelism use a 1D (row-wise) distribution. At the third level, each thread has four vector lanes in this example. Vector lanes are assigned to the non-zeros in a row of B. Thread-1 will use four vector lanes for each row of B it accesses (highlighted in different shades of blue) in succession. B is naturally laid out for this level of vector parallelism as the non-zeros in a row are consecutive. Vector parallelism is implemented using *vectorized* reads and multiplies on the corresponding rows of B. For example, thread-1 first does a read of $A(0,0)$, a VECTORREAD$(B(0,:))$ and the corresponding VECTORMULT followed by the read of $A(0,1)$, VECTORREAD$(B(1,:))$ and the corresponding VECTORMULT (loop in Line 7).

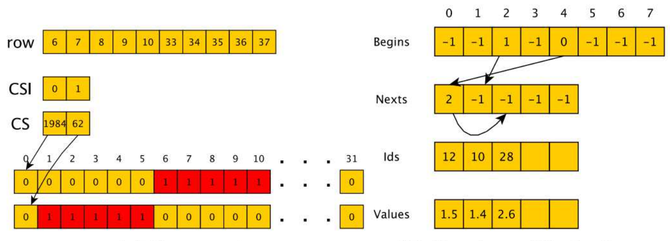
Given the partitioning above, we can describe the accumulator to accumulate the entries of C. We use a two-level, sparse, hashmap-based accumulator. Accumulators are used to compute the row size of C in the symbolic phase, and the column indices and their values of C in the numeric phase. Once kokkos-threads determine the row they work on, they allocate some scratch memory (Line 6) for their private level-1 ($\mathcal{L}_1$) accumulator (not to be confused with the L1 cache). The scratch memory maps to the GPU shared memory in GPUs and the default memory (i.e., DDR4 or high bandwidth memory) on KNLs. If the $\mathcal{L}_1$ accumulator runs out of space, global memory is allocated (Line 12) in a scalable way using memory pools (explained below) for a *thread private* $\mathcal{L}_2$ accumulator. This $\mathcal{L}_2$ accumulator is used for failed insertions from $\mathcal{L}_1$, and its size is chosen to guarantee to hold all insertions. Upon the completion of a row computation, any allocated $\mathcal{L}_2$ accumulator is explicitly released. Scratch spaces used by $\mathcal{L}_1$ accumulators are automatically released by Kokkos when the threads retire.

1) Implementation: This subsection focuses on two implementation aspects - length of the vector lanes and size of the accumulators. The length of vector lanes in each kokkos-thread is a runtime parameter, and it is fixed for all threads in a parallel kernel. We set it by rounding the average number of nonzeroes in a row of B (δ_B) to the closest power of 2 on GPUs (upper bound by the warp size). Kokkos sets the length depending on the compiler and underlying architecture specifications on the KNLs.

The size of $\mathcal{L}_1$ accumulators depends on the available *shared memory* on GPUs. In the numeric phase, the size of the $\mathcal{L}_2$ accumulator (in the *global memory*) is chosen to equal the "maximum row size in C" (MAXRS) to guarantee enough space for the work in any row of C. MAXRS is not known before symbolic so this phase uses an upper bound. The upper bound is the maximum number of multiplies (MAXRF) required by any row. This is found by summing up the size of rows of B that contributes to a row. In contrast to GPUs, both $\mathcal{L}_1$ and $\mathcal{L}_2$ accumulators are in the same memory space on KNLs/CPUs (DDR or MCDRAM). Since there are more resources per thread on the KNLs/CPUs, we choose the size of $\mathcal{L}_1$ accumulator big enough to hold MAXRF or MAXRS depending on the phase. Even this is usually small enough to fit into L1 or L2 caches of KNLs/CPUs.

B. Compression

This section addresses the problem of compressing the graph of B in the symbolic phase. This method, based on packing columns of B as bits, can reduce the size of B up to 32×. The graph structure of B encodes binary relations - existence of a nonzero in (i,j) or not. This can be represented using single bits. We compress the rows of B such that 32 (64) columns of B are represented using a single integer (long integer) similar to the color compression idea [20]. In this scheme, the traditional column index array in a compressed-row matrix is represented with 2 arrays of smaller size: "column set" (CS) and "column set index" (CSI). Set bits in the CS denote existing columns. That is, if



(a) Compression **(b)** Structure of the hashmap

Figure 2: Compression and Hashmap examples

the i^{th} bit in CS is 1, the row has a nonzero entry at the i^{th} column. The CSI is used to represent more than 32 columns. Figure 2a shows an example of the compression of a row with 10 columns. The compression is more successful if the column indices in each row are packed close to each other.

Each thread in Algorithm 4 performs $C(i,:) = A(i,:)\times B$, in which threads scan the rows A and C only once. However, a nonzero value in B is read multiple times, and total accesses to B are as many as f_m, i.e., $B(i,:)$ is read as many times as the size of $A(:,i)$. If we assume that the structure of A is uniform, that is, there are δ_A nonzeroes in each column and row of A, each row of B is accessed δ_A times. Thus, f_m becomes $O(\delta_A \times nnz_B)$, where nnz_B denotes the number of nonzeroes in B. When a compression method with linear time complexity ($O(nnz_B)$) like the one above reduces the size of B by some ratio $Comp$, the amount of work in the symbolic phase can be reduced by $O(Comp \times \delta_A \times nnz_B)$.

Compression reduces the problem size, allows faster row-union operations using BITWISEOR, and makes the symbolic phase more efficient. The reduction in row lengths of B also reduces the calculated MAXRF, the upper bound prediction for the required memory of accumulators in the symbolic phase. There exist more complicated graph compression methods in the literature. Some uses delta encoding to compress the columns of a row [21], while others seek for similarity of the rows (block structures) [22], [23], [24]. These expensive methods can be amortized on repetitive or computationally intensive problems. We avoid using such expensive methods, and show the effectiveness of our light-weight compression method in Section IV-C.

C. Memory Pool

Algorithm 4 requires a portable, thread-scalable memory pool to allocate memory for $\mathcal{L}_2$ accumulators when a row of C cannot be fit in a $\mathcal{L}_1$ accumulator. The memory pool is allocated and initialized before the kernel call and services requests to allocate and release memory from thousands of kokkos-threads. As a result, allocate and release has to be thread scalable. Allocate function returns a memory chunk to a thread and locks it. This lock is released when the thread releases the chunk back to the pool. The memory pool reserves NUMCHUNKS many memory chunks, where each has fixed size (CHUNKSIZE). CHUNKSIZE is chosen based on the MAXRF, and MAXRS in the symbolic and numeric phases, respectively. The memory pool has two operational modes: unique (non-unique) mapping of chunks to threads (ONE2ONE and MANY2MANY).

The parameters of the memory pool are architecture specific. NUMCHUNKS is chosen based on the available concurrency in an architecture. It is an exact match to the number of threads on the KNLs/CPUs. On GPUs, we over estimate the concurrency to efficiently acquire memory. We use an upper bound for the maximum allocated memory for the pool (such as $O(nnz_C)$), and reduce the NUMCHUNKS if the memory allocation becomes too expensive on GPUs. CPUs/KNLs use ONE2ONE and GPUs use MANY2MANY. The allocate function of the memory pool uses thread indices. These indices assist the look up for a free chunk. Pool returns the chunk with the given thread index on CPUs/KNLs (ONE2ONE mode). This allows CPU/KNL threads to reuse local NUMA memory regions. It starts a scan from the given thread-index until an available chunk is found on GPUs. It also helps the scan of GPU threads as they start their scan from different indices.

D. HashMap Accumulator

This section describes the hashmap based accumulator that supports parallel insertions/accumulations from multiple vector lanes. It is thread-private within kokkos-threads, so it needs to be highly scalable in terms of memory to fit within a small scratch space. The hashmap accumulator here extends the hashmap we used in [24] for parallel insertions. It consists of 4 parallel arrays as shown in Figure 2b, which shows an example of a hashmap that has a capacity of 8 hash entries and 5 (key, value) pairs. Hashmap is stored in a linked list structure. Ids and $Values$ stores the (key, value) pairs, which corresponds to column indices and their numerical values in numeric phase, to CSI and CS in symbolic phase. $Begins$ 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 ($Ids[0]$, $Values[0]$). The linked 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 are done via a vectorInsert operation. Each vector lane calculates the hash values, and traverses the corresponding linked list. 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, vector lanes read a single row of B, and the columns (keys) in a given row are unique. Thus, atomic operations are not needed for accumulation. If the key does not exist in the hashmap, vector lanes reserve the next available index in Ids

Table III: The specifications of the architectures used in the experiments.

Cluster	Shepard	Bowman	Hansen	White
CPU/GPU	Haswell	KNL	K80	P100
Compiler	icc 17.0.098	icc 17.0.098	gcc 4.8.4 cuda 7.5	gcc 5.4.0 cuda 8.0
Core specs	32×2.30 GHz cores 2 hyperthreads	68×1.40GHz cores 4 hyperthreads	Com. Cap. 3.7	Com. Cap. 6.0
Memory	128 Gb 2 NUMA	- 16 GB MCDRAM 460 GB/s - 96 GB DDR4 102 GB/s	12 GB	16 GB

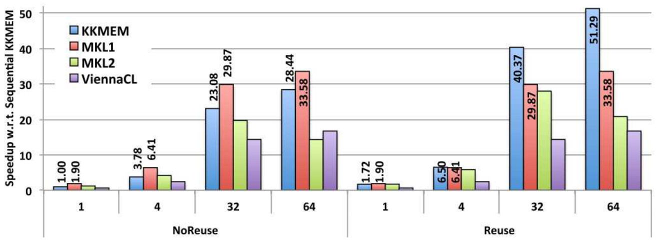


Figure 3: Geometric mean of the speedups of algorithm w.r.t. sequential KKMEM on Haswell CPUs. The numbers above the bars correspond the speedups of KKMEM and MKL1. The numbers below the bars are the number of threads. We exclude intermediate thread numbers since the trend of the speedups are similar.

and $Values$ with an atomic counter. They set the $Begins$ of the corresponding hash value to their insertion index with atomic_compare_and_swap, and the $Next$ of the inserted index is set to the old beginning index. If hashmap runs out of memory, it returns "FULL" and the failed vector lanes insert their values to the second level hashmap.

IV. Experiments

Setup: The configurations of the single nodes of the four clusters we use are listed in Table III. KKMEM method is implemented using the Kokkos library (Trilinos 12.8 release), and publicly available in Trilinos. Each run reported in this paper is the average of 5 executions with double precision arithmetic and 32 bit integers. We evaluate matrix multiplications in the forms of $P^T \times A \times P$ and $A \times A$ depending on the application domain of the corresponding matrices. The experiments use 29 (20 on Haswell and KNL) matrices from various multigrid problems and UF sparse matrices [25] used in [7], [14] (for $A \times A$). The problems are listed in Table IV. The experiments are organized in four parts. Section IV-A and Section IV-B evaluate the performance of KKMEM on CPUs/KNLs and GPUs respectively. Section IV-C evaluates the effect of the compression method. Overall results are summarized in Section IV-D.

A. Experiments on CPUs and KNLs

In this subsection, we compare KKMEM against ViennaCL (OpenMP) and the two SPGEMM methods provided in Intel Math Kernel Library (MKL) on multicore CPUs and KNLs. In the experiments, `mkl_sparse_spmm` routine in MKL's inspector-executor is referred as MKL1, and `mkl_dcsrmultcsr` routine is referred as MKL2. `mkl_dcsrmultcsr` is used with the option to sort the output rows, as it does not return sorted outputs. However, it requires sorted input rows. In the multigrid hierarchy the output matrices in one level is the input for the next level. This constrains us to set the outputs to be sorted. Both MKL routines are $2-3\times$ *slower* for the first call than the following calls in any run. Even though an application using MKL will observe this difference, we exclude the first run, for these routines. Thus, *the comparisons against* MKL *are conservative*. The test dataset includes 12 multigrid multiplications from Table IV, and 8 $A \times A$ multiplications for the largest matrices from our dataset.

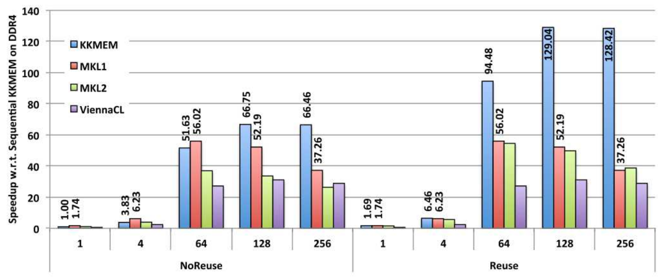
Figure 3 shows *the geometric mean of the speedup of the four different* SPGEMMs *with respect to single threaded* KKMEM *for all the* 20 *multiplications*. The figure also differentiates the NoReuse and Reuse cases. "Reuse" ("NoReuse") refers to the case where the symbolic phase is reused (recomputed) for every call to numeric phase. Two-phase algorithms (KKMEM and MKL2) benefit from the reuse. Although ViennaCL runs in two-phases, the public interface does not support the 2-phase usage. Hence we do not reuse the symbolic structure for ViennaCL.

In the NoReuse case MKL2 is up to 22% faster than KKMEM on smaller number of threads. However, KKMEM scales better and is the faster method after 16 threads (not shown). It is $2\times$ faster than MKL2 on 64 threads. Native implementation in ViennaCL is typically slower than the other three variants. MKL1 is the best method on traditional CPUs. KKMEM is designed to be thread-scalable for massively threaded architectures. The introduced overhead costs are usually not amortized on smaller number of threads. However, it scales better than MKL1. It is $1.90\times$ slower than MKL1 on sequential run, however the performance difference drops to 16% as the number of threads increase to 64 threads. The traditional CPU results are shown for demonstrating portability. It is not our target platform.

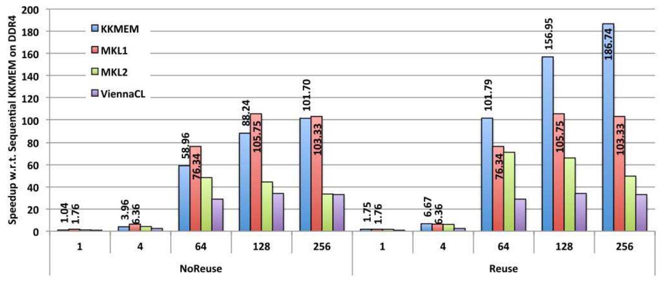
Two-phase methods take advantage of reusing the symbolic structure in Reuse experiments. MKL2 and KKMEM run up to 45% and 80% faster w.r.t. their NoReuse runs. KKMEM achieves a $51\times$ speed up on 64 threads ($30\times$ w.r.t. seq. Reuse). It is slower than MKL1 only when the resources are underutilized and become the fastest of the four methods beyond four threads, where it is $1.53\times$ faster than MKL1.

Figure 4 shows the speedups of the algorithms w.r.t. sequential KKMEM (on DDR4) on KNL. It differentiates runs using MCDRAM and DDR4 and the NoReuse/Reuse cases. In these experiments, we use 64 of 68 cores with 2 and 4 hyperthreads on 128 and 256 threads. MKL1 *does not complete in* 1000 *seconds or fails for* $7/20$ *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 $\frac{Speedup_{(20,256)}}{Speedup_{(20,128)}} = \frac{Speedup_{(13,256)}}{Speedup_{(13,128)}}$).

We compare the performance of the algorithms on both MCDRAM and DDR4. MCDRAM provides not only more bandwidth, but also more parallelism for the memory controllers compared to DDR4. When neither the bandwidth nor the memory controllers are saturated, the performance is expected to be similar on DDR and MCDRAM. As the



(a) DDR4



(b) MCDRAM

Figure 4: Strong scaling speedups on KNL's DDR4 and MCDRAM w.r.t. sequential KKMEM on DDR4. The results are geometric mean of 20 multiplications. cage15's result does not fit in MCDRAM due to the test driver's memory overhead, and it is excluded in MCDRAM results.

number of threads increase, the need for more bandwidth increases. MCDRAM can clearly help in this case. Even when the algorithm is not bandwidth bounded on DDR4, it may saturate memory controllers, and may suffer from latency due to the memory request queues. MCDRAM can help here with more parallelism in the memory request queues.

Figure 4a and 4b show speedups on DDR4 and MCDRAM, respectively. When using DDR4 (MCDRAM) KKMEM results in 66× and 128× (101× and 186×) speedup on 256 threads for the NoReuse and Reuse cases (w.r.t. sequential KKMEM on DDR4), respectively.

The DDR4 experiments (Figure 4a) demonstrate the strength of a thread-scalable KKMEM algorithm. MKL1 is initially $1.74\times$ faster than KKMEM on single-thread runs, however, the difference reduces to 8% on 64 threads. KKMEM becomes $1.28\times$ and $1.78\times$ faster than MKL1 on 128 and 256 threads, respectively. MKL1 performance drops on 128 and 256 threads suggesting that it is memory latency bound on 128 threads. On the other hand, KKMEM scales up to 128 threads, and its performance stays almost constant on 256 threads, which suggests that it is memory bandwidth bound on 256 threads. As in the multicore experiment, the performance of ViennaCL and MKL2 is slightly lower than MKL1 and KKMEM for NoReuse.

When the symbolic structure is reused, the performance of MKL2 and KKMEM improves up to $1.48\times$ and $1.93\times$ w.r.t. their "NoReuse" performance. KKMEM has similar performance to MKL1 on smaller number of threads, however it scales better beyond four threads. It is $2.47\times$ ($3.45\times$) faster than MKL1 on 128 (256) threads. MKL2 is usually slower than MKL1 even in Reuse case.

In general, storing matrices in MCDRAM improves the performances of all the algorithms (Figure 4b). KKMEM does not saturate the bandwidth or the memory controllers on DDR4 up to 32 threads (not shown), therefore the performance on MCDRAM is very close to its performance on DDR4 (at most 6% difference). After this point the speedups on MCDRAM are higher (up to $1.53\times$), and KKMEM scales well up to 256 threads. It results in a $101\times$ and $186\times$ speedup with 256 threads for NoReuse and Reuse case, respectively. The respective performance of ViennaCL and MKL2 improve up to $1.14\times$ and $1.32\times$, w.r.t. their performances on DDR4. However, they are still slower than KKMEM and MKL2 for both "NoReuse" and "Reuse" cases. More memory controllers provided in MCDRAM significantly helps the latency-bound problem of MKL1 on 128 and 256 threads. MKL1 scales up to 128 threads on MCDRAM, after which it hits the bandwidth bound. As a result, MKL1 improves its performance up to $2.77\times$ in MCDRAM. Although more bandwidth provided by the hardware helps to improve the scalability of MKL1, it still hits the memory bound earlier than KKMEM. In general, storing matrices in MCDRAM pays good dividends for larger thread counts as the larger bandwidth can be effectively utilized. This is reflected in the $101.70\times$ speedup for MCDRAM as opposed to the $66.46\times$ speedup for the DDR4 when using 256 threads.

B. Experiments on GPUs

We evaluate KKMEM against CUSP (`0.5.1`), bhSPARSE, cuSPARSE, ViennaCL (`1.7.1`) on two GPUs.

1) K80 Results: Table IV shows GFLOPS and execution time of KKMEM, and its speedup w.r.t. other methods on 28 different matrix multiplications on K80 GPUs. KKMEM and cuSPARSE are the most robust methods that can multiply all the matrices (except cage15, since the output matrix does not fit into memory). However, cuSPARSE is the slowest among the methods compared. ViennaCL, bhSPARSE and CUSP run out of memory for 21, 11 and 5 multiplications, respectively (excluding cage15). The bottom row shows the geometric mean of the speedup of KKMEM w.r.t. each algorithm for the set of matrices the method works. In general, the fastest algorithm is KKMEM, which is *on average* $3.80\times$, $1.59\times$, $1.47\times$ and $3.19\times$ faster than cuSPARSE, ViennaCL, BhSPARSE, and CUSP, respectively. KKMEM obtains the best performance on 22/30 multiplications. On Laplace_R_A and webbase multiplications, KKMEM is slower than BhSPARSE and CUSP. It is possible to change the vector length and improve the performance of KKMEM by $1.47\times$ and $2.80\times$, so that KKMEM has the best performance on these two matrices. However, we only report the performance with the default auto-parameter selection mechanism, and leave exploration of the better parameterization as future work. Overall, 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.

2) AmgX comparisons: In Table V, we compare the performance of KKMEM against the SPGEMM method in AmgX library [6]. This method is neither open-source, nor there exists a public interface for SPGEMM. Therefore, there is no way for us to compare with this method. However, we have been provided the performance numbers listed in the table for K80 GPUs. The performance of AmgX is better than other native methods, and more close to the performance of KKMEM. Execution time of both methods are usually in the orders of the milliseconds. We use Kokkos which is a wrapper around the CUDA library. It adds some overheads on the execution time, which are more likely to be visible in such tiny problems. KKMEM outperforms AmgX in the largest two matrices. Moreover, KKMEM is better than AmgX on webbase, since it is reported to be at least $3\times$ slower than CUSP [6]. In general, our portable method achieves similar performances to native AmgX method without using any architecture specific intrinsics functions as in AmgX.

3) P100 results: Table VI shows GFLOPS and execution time of KKMEM, and its speedup w.r.t. other methods on 29 different matrix multiplications on P100 GPUs. In general, performances of all methods improved w.r.t. K80 GPUs, and they run on larger set of matrices. KKMEM, CUSP, bhSPARSE, ViennaCL, and cuSPARSE improved their performances by $3.29\times$, $1.53\times$, $3.74\times$, $4.10\times$, and $5.26\times$, respectively (on the set of multiplications they run on both GPUs). With a faster GPU, the difference in execution time narrowed. The difference in execution time is negligible for the smaller matrices. Even in P100 KKMEM is the most robust and the fastest method. It obtains the best performance on 17 multiplications. In addition, the KKMEM times on ldoor, dielFilterV3real and delaunay_n24 and cage15 matrices are comparable to the times achieved in the distributed-memory implementation in [14] using 512 to 4096 cores. For example, the performance of KKMEM on delaunay_n24 on a single P100 GPU is comparable to using 2048 to 4096 Intel Ivy Bridge cores [14].

The effect of the memory pool: Kokkos-threads use memory pool to allocate memory for their $\mathcal{L}_2$ accumulators when their $\mathcal{L}_1$ accumulator is full. In the numeric phase, the size of each row of the result matrix is known beforehand. It is possible to allocate 4 arrays in the size of result matrix and allow threads to directly use them as accumulators to avoid using memory pools. We implemented this idea in the numeric phase to evaluate the performance of memory pool. The new method increases the memory requirement, where it runs out of memory on 3 multiplications on K80 GPUs (audi, dielFilterV3real, Brick_R_A), and on 1 multiplication on P100 (cage15). However, on average, the new method improves the results of numeric phase by 6% and 8% on K80, and 2% and 6% on P100 over using the memory pool (over all matrices). This demonstrates that memory pool increases the robustness of the algorithm by reducing the memory requirement, with a small increase on the runtime.

C. The effect of the compression

The compression technique is applied in the `symbolic phase` of KKMEM. This is a critical process to reduce the time and the memory requirements of the symbolic phase. At the beginning of this phase, the sizes of the rows of C are unknown. KKMEM estimates the maximum row size as MAXRF as used in the literature [10], [7]. The size of the accumulators are fixed to these sizes, which might cause memory problems on massively threaded architectures. By reducing the size of B using the compression technique, we reduce not only the number of insertions to hashmap accumulators (originally as many as f_m), but also the estimated maximum row size which in turn reduces the size of the accumulators used. We list the calculated maximum row sizes and the required size for accumulators with and without compression in Table VII together with the reductions on the size of B and the number of hashmap insertions. On average, the compression reduces the size of B, the number of insertions and the required memory by 57%, 59% and 53%. There are cases where compression increases the required size of an accumulator due to the additional array needed. However, in most of these cases, the size of accumulators are already very small. Therefore these cases are not likely to create memory issues. The memory reductions are more significant where it is more critical. Starting from Brick_R_AP multiplication, the required accumulator size per thread becomes more than $16KB$, and gets as high as $2.18MB$ on Empire_R_A matrices. In these sets of matrices, compression reduces the memory requirement on average by 74% and up to 96% on certain matrices. Compression usually reduces the runtime of the symbolic phase. When the reduction on insertions is low, it might not amortize compression cost. It is more successful when columns of the right hand side matrices are packed. Compression achieves lower reductions when the columns are spread out, which is the case for multigrid matrices (A and AP).

D. Overall Results

From previous experiments it is clear that the best method is different for different problems in different architectures. However, we allow a meta-algorithm that can choose the best method (excluding KKMEM) for a given problem in a given architecture. We can then compare KKMEM to this meta-algorithm that always chooses the best method for very conservative speedup numbers. Table VIII gives the geometric mean of the execution times of the meta-algorithm ("best method") for each instance (excluding KKMEM) on 19 matrices (excluding small matrices and cage15) and KKMEM on all architectures. These results also allows us to compare different architectures. First, the fastest overall multiplication times are obtained in P100 GPUs. Overall, performance achieved by the portable kernel is better than the best method by 17%, 4% and 54% on KNL-DDR4,

Table IV: The matrices and multiplications used throughout this paper. The (#rows, #cols, #nnz) of the input matrices and #multiplications performed are given in the first four columns. The right side lists the execution time in seconds and GFLOPS of KKMEM on K80, and its speedup w.r.t other SPGEMM methods. Blank spaces indicate the method failed. The matrices are sorted based the success of the algorithms, then by the #multiplications. The matrices 2cubes_sphere, cage12, webbase, offshore, filter3D, cant, hood, pwtk, ldoor are used as they are repetitively used in the literature [6], [7]. We run these matrices only on GPUs, and omit them on KNL experiments as their run times are negligible.

					KKMEM		**KKMEM speedups w.r.t.**			
	# row A	# cols A	#nnz A	# multiplications	time	GFLOPS	CUSP	bhSPARSE	ViennaCL	cuSPARSE
2cubes_sphere	101, 492	101, 492	1, 647, 264	27, 450, 606	0.05	1.098	3.40	1.80	1.40	5.20
cage12	130, 228	130, 228	2, 032, 536	34, 610, 826	0.08	0.865	2.75	1.25	1.50	4.50
webbase	1, 000, 005	1, 000, 005	3, 105, 536	69, 524, 195	0.57	0.244	0.72	0.65	6.07	2.07
offshore	259, 789	259, 789	4, 242, 673	71, 342, 515	0.12	1.189	3.58	1.42	1.42	9.25
filter3D	106, 437	106, 437	2, 707, 179	85, 957, 185	0.13	1.322	3.77	1.08	1.31	5.46
hugebubbles20	21, 198, 119	21, 198, 119	63, 580, 358	190, 713, 076	0.33	1.156	3.85	2.33	1.64	16.30
cant	62, 451	62, 451	4, 007, 383	269, 486, 473	0.16	3.369	9.69	1.38	2.19	1.44
Empire_RA_P	8, 800	2, 160, 000	25, 410, 400	91, 604, 280	0.22	0.833		2.45	0.59	1.59
europe	50, 912, 018	50, 912, 018	108, 109, 320	241, 277, 568	0.63	0.766		2.40	1.67	3.29
Laplace_A_P	15, 625, 000	15, 625, 000	109, 000, 000	400, 324, 972	0.52	1.540		1.56	0.83	15.19
Laplace_R_A	1, 969, 824	15, 625, 000	57, 354, 176	400, 324, 972	1.18	0.679		0.70	1.05	4.29
Laplace_R_AP	1, 969, 824	15, 625, 000	57, 354, 176	517, 542, 942	0.69	1.500		2.19	1.91	6.48
Laplace_RA_P	1, 969, 824	15, 625, 000	142, 929, 956	517, 542, 942	1.47	0.704		1.76	0.88	4.47
hood	220, 542	220, 542	10, 768, 436	562, 028, 138	0.28	4.014		1.25	2.57	3.61
pwtk	217, 918	217, 918	11, 634, 424	626, 054, 402	0.24	5.217		1.63	3.08	3.17
Empire_R_A	8, 800	2, 160, 000	8, 572, 251	1, 286, 511, 829	1.39	1.851		1.42	1.80	1.15
ldoor	952, 203	952, 203	46, 522, 475	2, 408, 881, 377	1.08	4.461		1.40	2.98	3.86
Empire_R_AP	8, 800	2, 160, 000	8, 572, 251	91, 604, 280	0.11	1.666			0.91	2.27
delaunay_n24	16, 777, 216	16, 777, 216	100, 663, 202	633, 914, 372	1.29	0.983			2.09	1.66
Brick_R_AP	592, 704	15, 625, 000	71, 991, 296	763, 551, 944	0.85	1.797			1.73	4.04
Empire_A_P	2, 160, 000	2, 160, 000	303, 264, 000	1, 286, 511, 829	1.09	2.361			1.11	3.61
channel	4, 802, 000	4, 802, 000	85, 362, 744	1, 522, 677, 096	1.56	1.952			2.03	4.06
Brick_AP	15, 625, 000	15, 625, 000	418, 508, 992	1, 934, 434, 936	1.69	2.289			1.09	7.75
Brick_R_A	592, 704	15, 625, 000	197, 137, 368	763, 551, 944	2.04	0.749				5.95
Brick_RA_P	592, 704	15, 625, 000	71, 991, 296	1, 934, 434, 936	1.70	2.276				2.31
bump	2, 911, 419	2, 911, 419	127, 729, 899	5, 745, 156, 927	3.12	3.683				3.08
audi	943, 695	943, 695	77, 651, 847	8, 089, 734, 897	4.76	3.399				2.68
dielFilterV3real	1, 102, 824	1, 102, 824	89, 306, 020	8, 705, 461, 058	6.68	2.606				3.04
cage15	5, 154, 859	5, 154, 859	99, 199, 551	2, 078, 631, 615						
						Geo.mean	**3.19**	**1.47**	**1.59**	**3.80**

Table V: Comparison against the performance numbers of AmgX on K80 GPUs. Last column shows the difference in running time between the two approaches. We use default parameters for KKMEM and compare against the best numbers of AmgX provided to us.

	AmgX		KKMEM		
	GFLOPS	Time	GFLOPS	Time	Time Diff.
2cubes_sphere	2.388	0.023	1.013	0.054	0.031
cage12	0.773	0.090	0.834	0.083	−0.007
offshore	2.359	0.060	1.143	0.125	0.064
filter3D	1.109	0.155	1.301	0.132	−0.023
cant	4.532	0.119	3.436	0.157	0.038
hood	5.435	0.207	4.082	0.275	0.069
pwtk	6.475	0.193	5.268	0.238	0.044
ldoor	4.074	1.183	4.477	1.076	−0.106
audi	2.112	7.661	3.396	4.765	−2.896

P100, and K80. The performance is as close as 1% on KNL-MCDRAM, and while it is 20% close the best method on Haswell. Moreover, the performance of KKMEM on P100 is the best methods among all the methods in all architectures. KKMEM obtains best performance on 13, 7, 5, 15 and 14 multiplications on KNL-DDR, KNL-MCDRAM, Haswell, Pascal, K80, respectively.

V. Conclusion

We described performance-portable, thread-scalable SPGEMM kernel for highly threaded architectures. We conclude by answering the primary question we started with - "How much performance will be sacrificed for portability?". We conclude that we do not sacrifice much in terms of performance on highly-threaded architectures. This is demonstrated by the experiments comparing our portable method against 5 native methods on GPUs, and 2 native methods on KNLs. Our SPGEMM kernel is also the most robust among publicly available codes whereas we see several failures with other kernels with the exception of cuSPARSE. It is possible to adapt our algorithm in a native method and improve it with architecture specific techniques.

Table VI: Execution time in seconds and GFLOPS of KKMEM speedup numbers w.r.t other SPGEMM methods on P100 GPUs.

	KKMEM		Speedup w.r.t.			
	time	gflops	CUSP	bhSPARSE	ViennaCL	cuSPARSE
2cubes_sphere	0.02	3.631	4.54	1.20	1.06	3.62
cage12	0.03	2.396	3.13	0.75	1.22	2.74
webbase	0.27	0.521	0.66	0.54	5.18	2.30
offshore	0.03	4.304	5.25	1.33	1.21	7.08
filter3D	0.03	4.918	5.78	0.83	1.47	4.30
hugebubbles20_0	0.10	3.804	4.99	4.81	1.94	12.14
Europe	0.18	2.669	3.41	5.57	2.57	2.50
cant	0.04	12.001	12.83	1.05	1.42	0.77
hood	0.08	13.944	14.22	0.97	1.77	1.72
pwtk	0.07	17.717	17.88	1.13	2.06	1.53
Empire_R_AP	0.04	4.734		0.89	0.65	0.88
Empire_RA_P	0.08	2.316		1.03	0.41	0.68
Laplace_R_A	0.39	2.041		0.68	0.73	2.71
Laplace_A_P	0.15	5.398		2.57	1.00	11.65
Laplace_R_AP	0.19	5.466		2.36	1.24	5.24
Laplace_RA_P	0.47	2.203		1.67	0.65	3.32
Brick_R_A	0.64	2.381		1.16	1.82	4.91
Empire_R_A	0.43	5.934		1.09	1.06	1.11
Empire_A_P	0.30	8.463		3.60	1.05	1.48
Brick_RA_P	0.61	6.326		1.26	0.43	1.14
ldoor	0.32	14.910		1.09	1.88	1.76
delaunay_n24	0.41	3.086			1.74	1.12
Brick_R_AP	0.24	6.349			0.76	1.91
channel	0.43	7.054			1.51	3.10
Brick_AP	0.49	7.954			0.95	4.54
cage15	1.56	2.660				4.86
Bump	0.88	13.126				1.58
audi	1.31	12.345				1.54
dielFilterV3real	1.80	9.679				1.85
		Geomean:	5.25	1.36	1.22	2.43

Table VII: The effect of the compression on hash insertions and memory requirements. Hashmaps would not need $Values$ (Figure 2b) without compression. The accumulator would require 3 parallel arrays, 2 of which must be at least MAXRF. This number is rounded up to closest power of 2 for faster hash calculations for $Begins$.

	Original		Compressed		Reduction		
	Max row size	Acc. size	Max row size	Acc. size	nnz_B	#ins.	Mem.
hugebubbles20	9	34	9	43	0.83	0.83	1.26
Laplace_A_P	36	136	36	172	0.93	0.93	1.26
Europe	44	152	35	169	0.63	0.64	1.11
Brick_AP	125	378	93	407	0.61	0.61	1.08
Laplace_R_A	252	760	184	808	0.72	0.72	1.06
delaunay_n24	280	1,072	194	838	0.48	0.47	0.78
channel	324	1,160	240	976	0.52	0.52	0.84
Laplace_RA_P	349	1,210	344	1,544	0.93	0.94	1.28
Laplace_R_AP	354	1,220	349	1,559	0.97	0.98	1.28
2cubes_sphere	544	2,112	342	1,538	0.48	0.48	0.73
offshore	562	2,148	380	1,652	0.61	0.61	0.77
cage12	989	3,002	678	3,058	0.75	0.74	1.02
Empire_A_P	999	3,022	275	1,337	0.57	0.55	0.44
Brick_R_AP	1,331	4,710	1,028	5,132	0.72	0.74	1.09
Brick_RA_P	1,331	4,710	907	3,745	0.61	0.65	0.80
cage15	1,997	6,042	1,555	6,713	0.80	0.80	1.11
filter3D	3,340	10,776	2,103	10,405	0.56	0.54	0.97
Brick_R_A	3,375	10,846	1,199	5,645	0.35	0.35	0.52
hood	3,871	11,838	609	2,851	0.13	0.13	0.24
ldoor	4,165	16,522	686	3,082	0.12	0.12	0.19
cant	5,913	20,018	698	3,118	0.13	0.12	0.16
pwtk	8,474	33,332	838	3,538	0.09	0.09	0.11
Empire_RA_P	8,800	33,984	275	1,337	0.57	0.59	0.04
Empire_R_AP	8,800	33,984	275	1,337	0.63	0.63	0.04
Bump	9,370	35,124	1,626	6,926	0.17	0.17	0.20
dielFilterV3real	26,163	85,094	6,810	28,622	0.24	0.23	0.34
audi	32,985	131,506	4,764	22,484	0.15	0.14	0.17
webbase	116,179	363,430	31,251	126,521	0.77	0.25	0.35
Empire_R_A	155,460	573,064	26,158	111,242	0.15	0.15	0.19
			GEOMEAN:		**0.43**	**0.41**	**0.47**

Table VIII: Geometric mean of the execution times. Best method is the geometric mean obtained by choosing the execution time of the best algorithm (excluding KKMEM) for each instance on each architecture.

	KNL-DDR4	KNL-MCDRAM	Haswell	Pascal	K80
Best Method	0.790	0.477	0.362	0.342	1.673
KKMEM	0.676	0.480	0.455	0.328	1.084

However, in our experience, there is only small room left to improve the portable kernel in these two architectures. We addressed the questions raised in Section I throughout the paper. We summarize the conclusions here.

- The key to performance portability is to design an algorithm that relies on thread scalable data structures, uses memory efficiently and maps correctly to the hierarchical parallelism. The data structures and compression technique play a major role in performance and robustness.
- Designing for application use cases such as the reuse results in significantly better performance than past methods.

Acknowledgements: We thank Grey Ballard, Erik Boman, Karen Devine, Carter Edwards and Simon Hammond 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.

REFERENCES

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

[2] J. Beyer and J. Larkin, "Targeting GPUs with OpenMP 4.5 device directives," in *Proceedings of the GPU Technology Conference*, 2016.

[3] 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.

[4] 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.

[5] Intel, "Intel Math Kernel Library," 2007.

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

[7] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th IPDPS*. IEEE, 2014, pp. 370–381.

[8] 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.

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

[10] 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.

[11] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[12] "The Trilinos project." [Online]. Available: https://trilinos.org

[13] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, 2011.

[14] 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*.

[15] 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.

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

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

[18] M. Deveci, E. G. Boman, and S. Rajamanickam, "Sparse matrix-matrix multiplication for modern architectures." in *SIAM Workshop on Combinatorial Scientific Computing (CSC16)*. IEEE, 2016.

[19] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro, "Design considerations for a flexible multigrid preconditioning library," *Scientific Programming*, vol. 20, no. 3, pp. 223–239, 2012.

[20] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *2016 30th IEEE IPDPS*, May 2016, pp. 892–901.

[21] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," *20th international conference on Supercomputing*. ACM, 2006, pp. 307–316.

[22] A. C. Gilbert and K. Levchenko, "Compressing network graphs," *LinkKDD workshop at the 10th ACM Conf. on KDD*, vol. 124, 2004.

[23] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: an extended compression format for SpMV on shared memory systems," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 247–256.

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

[25] 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.