Register Tiling for Unstructured Sparsity in Neural Network Inference

LUCAS WILKINSON*, University of Toronto, Canada
KAZEM CHESHMI*, McMaster University, Canada
MARYAM MEHRI DEHNAVI, University of Toronto, Canada

Unstructured sparse neural networks are an important class of machine learning (ML) models, as they compact model size and reduce floating point operations. The execution time of these models is frequently dominated by the sparse matrix multiplication (SpMM) kernel, $C = A \times B$, where A is a sparse matrix, and B and C are dense matrices. The unstructured sparsity pattern of matrices in pruned machine learning models along with their sparsity ratio has rendered useless the large class of libraries and systems that optimize sparse matrix multiplications. Reusing registers is particularly difficult because accesses to memory locations should be known statically. This paper proposes Sparse Register Tiling, a new technique composed of an unroll-and-sparse-jam transformation followed by data compression that is specifically tailored to sparsity patterns in ML matrices. Unroll-and-sparse-jam uses sparsity information to jam the code while improving register reuse. Sparse register tiling is evaluated across 2396 weight matrices from transformer and convolutional models with a sparsity range of 60-95% and provides an average speedup of 1.72× and 2.65× over MKL SpMM and dense matrix multiplication, respectively, on a multicore CPU processor. It also provides an end-to-end speedup of 2.12× for MobileNetV1 with 70% sparsity on an ARM processor commonly used in edge devices.

CCS Concepts: • **Software and its engineering → Source code generation**; • **Computing methodologies** → *Neural networks*.

Additional Key Words and Phrases: Register Tiling, Sparse Matrix, Pruned Neural Networks

1 INTRODUCTION

Unstructured sparse matrices are an important class of matrices in deep neural networks. They primarily arise from element-wise pruning (or fine-grained pruning) of weight matrices, a technique that reduces both the size and computation cost of neural networks with minimal accuracy loss [Hoefler et al. 2021]. Pruning is especially useful for resource-constrained edge devices. However, element-wise pruning focuses exclusively on the impact of individual weights on the accuracy of the network [Ding et al. 2019; Han et al. 2015; Liu et al. 2021; Miao et al. 2022; Sanh et al. 2020; Yu et al. 2022] with little regard to performance and struggles to provide meaningful speedup over their dense counterparts. For neural networks, pruning sparsity values in the range of 60% − 95% are of particular interest due to a concept known as Occam's Hill [Rasmussen and Ghahramani 2000]. This refers to the empirically-observed [Hoefler et al. 2021] fact that some pruning can improve accuracy (through improved generality) but too much can lead to steep losses in accuracy.

The computationally intensive layers in deep neural networks are frequently lowered to matrix multiplications, $C = A \times B$, hence accelerating this kernel is essential to performance. As a result of pruning, the weight matrix (A) becomes sparse and the sparse weight matrix (A) needs to get multiplied with a dense activation matrix (B), also known as sparse matrix multiplication (SpMM). The unstructured nature of element-wise pruning leads to matrices with nearly random sparsity patterns. As a result, developing an efficient implementation of SpMM that outperforms dense matrix multiplication becomes extremely challenging. Consequently, most ML frameworks will use

*Both authors contributed equally to this research.

Authors' addresses: Lucas Wilkinson, lwilkinson@cs.toronto.edu, University of Toronto, Canada; Kazem Cheshmi, cheshmi@mcmaster.ca, McMaster University, Canada; Maryam Mehri Dehnavi, mmehride@cs.toronto.edu, University of Toronto, Canada.

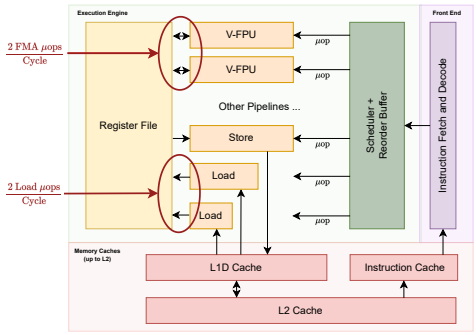

Fig. 1. Simplified Common CPU Microarchitecture with 2 load units and 2 vector floating point units (V-FPUs), capable of performing 2 vector-loads-per-cycle and 2 vector-FMAs-per-cycle, respectively.

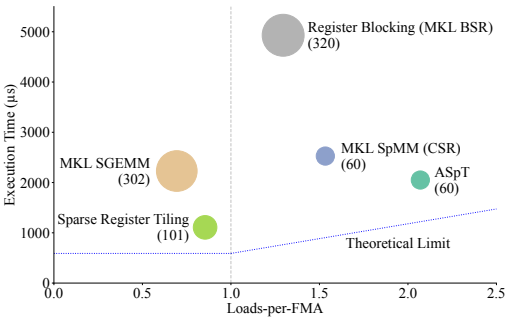

Fig. 2. Loads-per-FMA in state-of-the-art SpMM/S-GEMM implementations, (A(784x784) $\times B$(784x256), 80% uniform random sparsity pattern), bubble size and the numbers in the parentheses represent total mega floating point operations (MFLOPs).

implementations that operate on dense matrix routines instead and forgo the potential benefits that pruning delivers.

Data movement is the major challenge in implementing an efficient SpMM because it is memory bound, i.e., its execution time is dominated by loading and storing data. Individual values from matrices A, B, and C should be first loaded into the first level cache, i.e. L1, from RAM through a cache hierarchy, e.g. L2 and L3. Once a partition of each of the three matrices is in L1, they are loaded to registers, where a fused-multiply-add (FMA) operation is applied. Considering the irregular random memory accesses in SpMM, prior work such as [Hong et al. 2019; Kurt et al. 2020] have investigated techniques to reuse data in cache levels and reduce the accesses to DRAM. These methods are known as cache tiling techniques as they tile loops to reuse data already in the cache. Cache tiling methods run operations with common accesses (i.e. temporal locality) in the vicinity of each other (i.e. spatial locality) by restoring matrix A or reordering FMA operations. These methods aim to effectively reuse the data in the cache. However, the efficient use of registers, which can potentially lead to noticeable reductions in execution time, is not well explored.

When data movement in SpMM is optimized for the cache hierarchy, the bandwidth between L1 and the registers becomes the new bottleneck. Loading data from the L1 cache into registers is limited by the number of load units, which is often less than or equal to the number of floating point units (FPUs) capable of performing FMAs, i.e., an element of A times an element of B accumulated into an element of C. For example, Figure 1 shows a simplified microarchitecture similar to what exists in most modern CPUs. In this example, the execution engine (backend) supports a throughput of up to two FMA μops per cycle and two loads μops per cycle. To fully saturate the floating point units, the code should execute at most one load-per-FMA, above this threshold loads become the bottleneck. An FMA operation requires more than one operand to be loaded. Therefore, to get an average of at most one load-per-FMA, each loaded operand must be used by at least one other operation (FMA). For SpMM, this means that each element of A, B and C loaded into registers must be reused; failure to reuse a loaded element for any of the matrices results in an average loads-per-FMA greater than one. The reciprocal of load-per-FMA is similar to arithmetic intensity, however, we do not use this term to prevent confusion with the roofline model [Williams et al. 2009] that looks at DRAM-to-cache bandwidth and not L1-to-register bandwidth.

Register tiling increases register reuse and as a result, decreases the number of loads-per-FMA. This is typically done using a combination of loop tiling and unroll-and-jam [Carr and Kennedy

1994], which creates an unrolled sequence of FMA operations with common data accesses. Since the operations are unrolled, the common access pattern is statically known. A statically known access pattern is required because registers are only addressable using absolute addresses at compile-time; the compiler is responsible for both the allocation and addressing of registers. Unlike cache tiling where the hardware detects common accesses (i.e. cache hits), for register reuse to occur, the compiler should have acknowledge of common accesses to allocate and address registers efficiently. The irregular memory accesses in SpMM make statically resolving the access pattern of a sequence of operations with sufficient common accesses difficult. Thus register tiling techniques used in SpMM are frequently limited in applicability or the amount of reuse they expose.

A common register tiling strategy for SpMM is to unroll multiple operations associated with a single nonzero of A [Elsen et al. 2020; Gale et al. 2020; Hong et al. 2019; Yang et al. 2018]; as a result, elements of A can be loaded into a register and reused multiple times. Then by traversing the nonzeros row-by-row, elements of C are loaded into registers and also reused during the computation of an entire row (or an entire column in column-by-column order, and by reusing B instead). Since this strategy involves traversing a single row (or column) at a time, the reused elements of C (or B) form a vector, and for this reason, it is sometimes referred to as 1D register tiling. This strategy has benefits because only operations associated with a nonzero (required operations) are performed, reducing the number of floating point operations (FLOPs) compared to dense matrix multiplication. Also, since unrolled operations are oblivious to the nonzero pattern of A, the unrolled code can be jammed. However, with 1D register tiling, one operand still needs to be loaded per FMA, failing to find register reuse on all three matrices, and as a result, the average loads-per-FMA will be greater than one. For a row-by-row traversal, the elements of A and C can be loaded once and reused but this fails to reuse the loaded element of B.

To enable reuse in all three matrices, register blocking [Vuduc et al. 2005, 2002] unrolls operations associated with a dense 2D block of nonzeros in A. The block shape is predetermined and thus all of the nonzero locations inside the block are known statically along with their associated accesses to B and C. Matrix multiplication is then computed block by block. Because a 2D tile of one of the matrices can be loaded into registers and operated on, reuse is enabled on all three matrices, i.e., 2D register tiling. However, the nonzero locations in unstructured sparse matrices often do not form blocks. Thus, the register blocking technique fills in part of the blocks with explicit zeros, these zeros are operated on as if they are nonzeros. The explicit zeros increase the number of FLOPs required to compute the SpMM. The extra FLOPs are referred to as *redundant FLOPs*. The register blocking technique will then select blocks that minimize redundant FLOPs while covering all nonzeros in A. The fill ratio [Vuduc and Moon 2005] measures the efficiency of blocking and is the sum of explicit zeros and nonzeros divided by the number of nonzeros. Register blocking is profitable for matrices with a low fill ratio; conceptually the fill ratio can be seen as the number of additional FLOPs the register blocking approach will compute compared to a 1D tiling approach. The unstructured sparse matrices found in machine learning typically have a very high fill ratio. We tested the common block shapes 2×2, 4×4 and 8×8 on 2396 unstructured ML matrices in the sparsity range of 60% - 95%, and found an average fill ratio of 2.91, 6.59 and 11.75, respectively. This is because these matrices typically have a highly uniform random distribution of nonzeros.

We propose *sparse register tiling* which achieves state-of-the-art performance on unstructured sparse matrices by enabling register reuse for all three matrices (A, B, and C) while avoiding excessive redundant FLOPs. Sparse register tiling has a compile-time *unroll-and-sparse-jam* transformation and a runtime *scheduler/compressor*. The unroll-and-sparse-jam transformation enables register reuse in all three matrices by unrolling operations associated with nonzeros of A. It does not require that the nonzeros of A conform to a block structure and thus reduces the fill ratio (i.e. reduces redundant FLOPs). To efficiently jam the unrolled code, unroll-and-sparse-jam enumerates possible

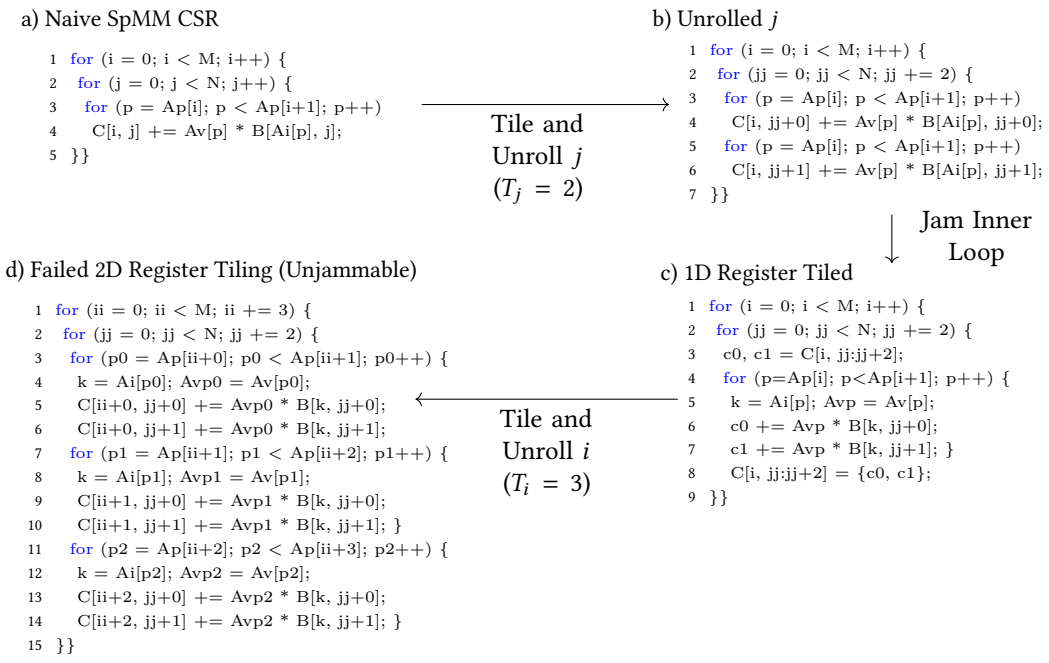

Fig. 3. Register Tiling for SpMM Compressed Sparse Row (CSR) with register tile $T_i \times T_j = 3 \times 2$

execution patterns and generates specialized code for each. This can lead to excessive code bloat negatively impacting performance and is resolved using a mathematical program that trades off code size with redundant FLOPs. The runtime scheduler/compressor schedules operations ahead of time to further reduce the overhead from conditionals in the code and compresses matrix A to reduce memory usage. For ML inference, the scheduler/compressor needs to run once during model deployment, and the executor code is used for every inference run. On a 20-core Intel processor, sparse register tiling provides an average speedup of 2.65× and 1.72× over MKL's matrix multiplication (SGEMM) and MKL SpMM using a CSR storage format, respectively for 2396 matrices in the deep learning matrix collection (DLMC) [Gale et al. 2020], using 32-bit floating point math. Sparse register tiling also shows 2.12× speedup for the MobileNetV1 [Howard et al. 2017] model over XNNPACK SpMM on an ARM processor used in edge devices.

2 RUNNING EXAMPLE

This section provides an example of performing 1D register tiling for SpMM (operating on a Compressed Sparse Row (CSR) storage format) to illustrate common challenges and limitations. It is then followed by a discussion of the approach used in sparse register tiling.

Figure 3a shows an example code for a naive SpMM implementation using CSR. We assume a tile of C remains in registers for the full execution of the innermost loop. This is a common strategy as, unlike elements of A and B, elements of C are accessed twice for each FMA (one load and one store). Figures 3a–c show the steps for performing 1D register tiling. By unrolling the tiled j loop, multiple operations (FMAs) associated with the same nonzero are unrolled Av[p] (p is constant when only updating j). As mentioned earlier, 1D register tiling struggles because at least one operand gets loaded from the L1 cache to a register per FMA. In Figure 3c, an element of B is loaded per FMA.

So while this code enables register reuse for both A and C, failure to reuse the loaded element of B leads to this code averaging more than one load-per-FMA.

To enable reuse in B, multiple operations should be unroll-and-jammed while keeping the loop iterators k and j constant. This is because accesses to B are determined by k and j. Thus in SpMM, the i loop is the only candidate for unroll-and-jam. If successful, reuse is enabled for all three matrices. The result is a 2D register tile of C since its accesses are determined by unrolled loop iterators i and j. However, attempting this for the SpMM CSR code in Figure 3d results in unjammable loops after unrolling. From Figure 3 (d), the i loop is unrolled but has inner loops (lines 3, 7, 11) that cannot be jammed because their loop bounds are not known at compile-time. Here common accesses to B, which occur when $\mathrm{Ai[p0]} == \mathrm{Ai[p1]}$ or $\mathrm{Ai[p1]} == \mathrm{Ai[p2]}$, are in separate loops and hidden from the compiler preventing reuse. Thus without knowledge of the location (or relative location) of other nonzeros, the best that can be achieved is 1D register tiling (i.e. reuse for only two of the matrices). The register blocking approach works around this by grouping nonzeros into dense blocks and adding explicit zeros; within a dense block, all of the relative nonzero (including explicit zeros) locations are known enabling the reuse of B at the cost of redundant FLOPs.

Figure 2 shows the impact of both the limited reuse in 1D register tiling and redundant FLOPs in register blocking empirically using state-of-the-art SpMM implementations. Adaptive sparse tiling (ASpT) [Hong et al. 2019] is a state-of-the-art CSR-based implementation that focuses on adaptive cache tiling and uses 1D register tiling. In comparison, we show an implementation of register blocking from Intel's highly optimized math kernel library (MKL) [Intel 2022]. For reference, we also included a CSR-based implementation from MKL, labeled MKL SpMM CSR, as well as a dense implementation also from MKL, labeled MKL SGEMM. As shown both CSR-based implementations average more than one load-per-FMA as a result of only operating on one nonzero at a time; placing them in the load-bound region (load-per-FMA > 1). The register blocking implementation, which is the MKL block compressed sparse row (BSR), does on average fewer loads-per-FMA compared to the CSR-based implementations due to its use of 2D blocks, however, the excessive amount of redundant FLOPs makes it a poor performer overall. The dense implementation (MKL SGEMM) is an extreme version of register blocking where the entire matrix forms a single block and performs better as a result of the computation pattern being known with no storage format overhead, however, its performance is also hindered by excessive redundant FLOPs.

In our method, sparse register tiling, instead of viewing SpMM as looping over nonzeros (1D register tiling) or blocks of nonzeros (register blocking) we treat SpMM as a dense matrix multiplication with guards to skip redundant iterations (Listing 1). This approach reflects the weight pruning process in machine learning, weight matrices are most commonly trained dense and then pruned for inference. As a result, in sparse register tiling, loop transformations are performed in the same was as dense code; all the loop bounds and accesses are statically known. Thus the code is successfully unrolled and jammed for the i and j loops as shown in Listing 2. However, this approach introduces new conditionals in the loop body. While all accesses are statically known, potential common accesses to B, e.g. lines 7, 10, and 13, are wrapped with different conditionals preventing register reuse and limiting instruction-level parallelism inside the loop body.

To remove the conditionals over accesses to B, we enumerate all the possible execution paths in the loop body at compile-time. This leads to the code in Listing 3 that has all, i.e. $2^3 - 1 = 7$, enumerated execution paths as condition-less code blocks guarded by individual guard clauses. As a result, inside each guarded code block, the common accesses to B lead to register reuse inside the code block. However, Listing 3 has three drawbacks: *(i)* Code size and conditionals increase exponentially with the tile size of T_i, leading to inefficient use of the instruction cache; *(ii)* A large number of conditionals is detrimental to performance as, in the worst-case scenario, all conditionals of the inner loop body must be checked. This problem becomes more prominent since

```
for (i = 0; i < M; i++) {
 for (j = 0; j < N; j++) {
  for (k = 0; k < K; k++)
   if (A[i, k])
    C[i, j] += A[i, k] * B[k, j];
}}
```

Listing 1. Using Conditions

```
for (ii = 0; ii < M; ii += 3) {
 for (jj = 0; jj < N; jj += 2) {
  c00, c01 = C[ii+0, jj:jj+2];
  c10, c11 = C[ii+1, jj:jj+2];
  c20, c21 = C[ii+2, jj:jj+2];
  for (k = 0; k < K; k++) {
   if (A[ii+0, k]) {
    c00 += A[ii+0, k] * B[k, jj+0];
    c01 += A[ii+0, k] * B[k, jj+1]; }
   if (A[ii+1, k]) {
    c10 += A[ii+1, k] * B[k, jj+0];
    c11 += A[ii+1, k] * B[k, jj+1]; }
   if (A[ii+2, k]) {
    c20 += A[ii+2, k] * B[k, jj+0];
    c21 += A[ii+2, k] * B[k, jj+1]; }
  }
  C[ii+0, jj:jj+2] = {c00, c01};
  C[ii+1, jj:jj+2] = {c10, c11};
  C[ii+2, jj:jj+2] = {c20, c21};
}}
```

Listing 2. Unrolled *i*, *j* (and Jammed)

```
1  for (ii = 0; ii < M; ii += 3) {
2   for (jj = 0; jj < N; jj += 2) {
3    c00, c01 = C[ii+0, jj:jj+2];
4    c10, c11 = C[ii+1, jj:jj+2];
5    c20, c21 = C[ii+2, jj:jj+2];
6    for (k = 0; k < K; k++) {
7  e1: if (A[ii+0, k] && A[ii+1, k]
           && A[ii+2, k]){//avg_freq=0
8      c00 += A[ii+0, k] * B[k, jj+0];
9      c01 += A[ii+0, k] * B[k, jj+1];
10     c10 += A[ii+1, k] * B[k, jj+0];
11     c11 += A[ii+1, k] * B[k, jj+1];
12     c20 += A[ii+2, k] * B[k, jj+0];
13     c21 += A[ii+2, k] * B[k, jj+1]; }
14 e2: if (A[ii+0, k] && !A[ii+1, k]
           && A[ii+2, k]) {//avg_freq=2
15     c00 += A[ii+0, k] * B[k, jj+0];
16     c01 += A[ii+0, k] * B[k, jj+1];
17     c20 += A[ii+2, k] * B[k, jj+0];
18     c21 += A[ii+2, k] * B[k, jj+1]; }

    ... other cases (e3, e4, e5, e6)

34 e7: if (!A[ii+0, k] && !A[ii+1, k]
           && A[ii+2, k]) {//avg_freq=1
35     c20 += A[ii+2, k] * B[k, jj+0];
36     c21 += A[ii+2, k] * B[k, jj+1]; }
37   }
38   C[ii+0, jj:jj+2] = {c00, c01};
39   C[ii+1, jj:jj+2] = {c10, c11};
40   C[ii+2, jj:jj+2] = {c20, c21};
41 }}
```

Listing 3. Enumerated

```
1  for (ii = 0; ii < M; ii += 3) {
2   for (jj = 0; jj < N; jj += 2) {
3    c00, c01 = C[ii+0, jj:jj+2];
4    c10, c11 = C[ii+1, jj:jj+2];
5    c20, c21 = C[ii+2, jj:jj+2];
6  r1: for (p=Atp[ii][0]; p<Atp[ii][1]; p++) {
7      k = Atk[p];
8      c00 += Atv[0] * B[k, jj+0];
9      c01 += Atv[0] * B[k, jj+1];
10     c20 += Atv[1] * B[k, jj+0];
11     c21 += Atv[1] * B[k, jj+1];
12     Atv+=2; }
13 r2: for (p=Atp[ii][1]; p<Atp[ii][2]; p++) {
14     k = Atk[p];
15     c10 += Atv[0] * B[k, jj+0];
16     c11 += Atv[0] * B[k, jj+1];
17     c20 += Atv[1] * B[k, jj+0];
18     c21 += Atv[1] * B[k, jj+1];
19     Atv+=2; }
20   C[ii+0, jj:jj+2] = {c00, c01};
21   C[ii+1, jj:jj+2] = {c10, c11};
22   C[ii+2, jj:jj+2] = {c20, c21};
23 }}
```

Listing 4. Sparse register tiling's generated executor code

the unstructured nature of the pruning increases branch mispredictions. *(iii)* Memory usage is inefficient since A is stored as dense; many zero locations of A are stored but never used beyond the guard clauses.

An example of the resulting executor code from sparse register tiling is shown in Listing 4. As shown the code is not only more compact compared to Listing 3 but also the guard statements (lines 7,14,34) are replaced with more branch predictor friendly loops (lines 6,13). The code compression is made possible by our sparse-jam solver that trades off redundant FLOPs for code size based on the sparsity pattern of the target matrices; for example, the lines 8-11 (r1) in Listing 4 can be used in lieu of executing lines 15-18 (e2) and 35-36 (e7) in Listing 3, with lines 8-9 (Listing 4) becoming redundant FLOPs in the latter case. The code compression in unroll-and-sparse-jam uses frequency information, the average frequency at which the conditionals in Listing 3 appear in ML matrices; avg_freq in line 15 in Listing 3 shows the frequency of the condition. The use of loops instead of guard statements is made possible with a scheduler in the pre-processing phase that populates the Atp and Atk arrays. With these arrays, the iterations of the inner k loop are scheduled such that the iterations using the same code are executed sequentially. As a result, code branches are easier to predict. Sparse register tiling additionally compresses the matrix using the provided schedule as a part of the pre-processing phase. This compression is beneficial on small devices with limited memory or for processing large models containing millions of parameters requiring large expensive servers otherwise. As shown in Listing 4, nonzero elements are loaded from Atv, which is the compressed matrix A and is populated by our data compressor.

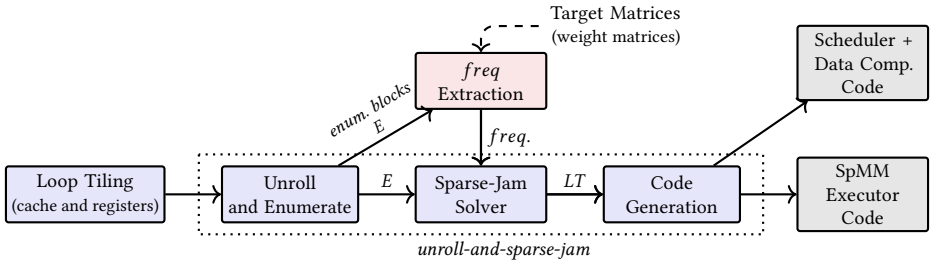

Fig. 4. Overall view of code generation components in sparse register tiling, LT stands for lookup table

3 SPARSE REGISTER TILING

This section covers the sparse register tiling technique. We start with an overview of sparse register tiling and then explain each of its components and finally discuss its safety.

3.1 Overview

Figure 4 shows an overall view of how sparse register tiling is applied to an SpMM code. The first step is loop tiling, it partitions the matrix multiplication such that each partition fits in caches, i.e. cache tiles, and then recursively partitions cache tiles to fit into registers, i.e. register tiles.

Sparse register tiling then unrolls the register tile as shown in Listing 2. All the code paths through the inner loop body (lines 7–15 in Listing 2) are then enumerated as shown in Listing 3. We refer to each enumerated code path as an *enumerated block*. The access patterns of each enumerated block is statically known, allowing the compiler to efficiently allocate and reuse registers. The sparse-jam solver takes in the frequency at which the enumerated blocks appear in target matrices. It uses the frequency information along with an execution cost model for the enumerated blocks, to find a mapping from all of the enumerated blocks (set E) to a subset of enumerated blocks that will be generated as the final code (set R; $R \subseteq E$), i.e. the sparse-jam solver finds a mapping from E to R. The computed mapping is then stored as a lookup table (LT).

The final step is code generation, it takes in LT from the sparse-jam solver and generates pre-processing code, which are the scheduler and data compressor, as well as executor code that performs the matrix multiplication. For the executor, the code generator will generate specialized code snippets for each enumerated block in R (output of the lookup table). The loop tiling, unroll and enumerate, sparse-jam solver and code generation modules are compile-time (purple modules in Figure 4. The frequency extraction is based on profiling (in pink), and the generated scheduler, data compression, and executor code are runtime modules. In neural network inference, the sparse weight matrices are static. All modules except for the SpMM executor run once for a given static sparse weight matrix.

3.2 Loop Tiling

Loop tiling is required to optimize SpMM for all levels of the CPU memory hierarchy, i.e. caches and registers. With loop tiling, large computations (loops) are partitioned into smaller sub-computations to reduce cache misses (with cache tiling) and register spills (with register tiling). This is achieved by ensuring that the working set size, bytes of data needed to perform the sub-computation, is small enough to fit into a target level of the memory hierarchy. For example lets consider a dense matrix multiplication where A is size of $M \times K$, B is size of $K \times N$ and C is size of $M \times N$ with loops i, j and k that iterate over dimensions M, N and K respectively. If the three loops are tiled with tile sizes of T_i, T_j, and T_k, the matrix multiplication will be partitioned into a sequence of smaller

matrix multiplications. As a result, each of the small matrix multiplications has a small working set size of $T_iT_j + T_iT_k + T_kT_j$ multiplied by the size of a single scalar; T_iT_j, T_iT_k, and T_kT_j represent contributions from C, A and, B respectively.

This work focuses on register tiling, however, adopting an efficient cache tiling strategy is essential. Failure to tile efficiently for caches results in excessive data movement between DRAM and the caches, and becomes the dominant bottleneck. To choose cache tile sizes, we should first find the working set size for a set of tile sizes (T_i^c, T_j^c and T_k^c, superscript c indicates tiling for caches). However, determining the working set size in SpMM is challenging, e.g. in Listing 1 the guard clause in line 4 causes some accesses to C, A, and B to be skipped decreasing the working set size.

To find a loop tiling strategy for caches, we measure the coefficient of variation (CoV) of the working set size for different tile sizes; a larger CoV translates to a larger change in the working set size between tiles, making tile size selection challenging. Figure 5 depicts variation for 200 randomly selected matrices for the matrices with 70%+ sparsity in the deep learning matrix collection (DLMC) [Gale et al. 2020], orange dots. For contrast, we also analyzed 200 randomly selected matrices with the same sparsity range from the SuiteSparse collection [Davis and Hu 2011] (blue dots), which contains sparse matrices from scientific computing. For a dense matrix, the CoV is zero for all tile sizes because tile size and working set size are well correlated. Compared to matrices in SuiteSparse, the DLMC matrices have a low CoV because the nonzeros in these matrices are more uniformly distributed. Given the low CoV across tile sizes in DLMC, the working set size can be inferred from the tile size and a similar tiling strategy to that of a dense matrix can be applied. We adopt the dense tiling strategy from [Kung et al. 2021] to find cache tile sizes (T_i^c, T_j^c and T_k^c). For a compressed A, we take the size of metadata for compression into account when calculating the working set size.

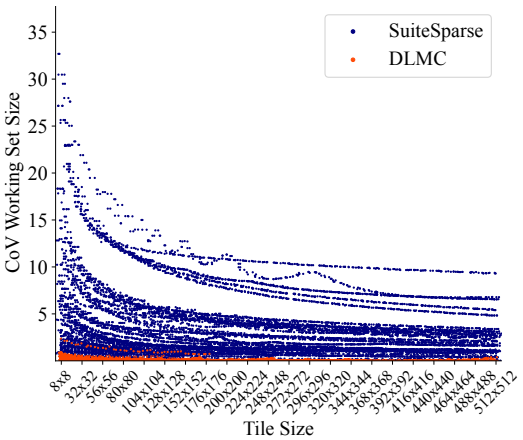

Fig. 5. Coefficient of Variation (CoV) in working set size vs Tile Size ($T_i^c \times T_j^c$), where $T_k^c = 256$.

After cache tiling, loops are tiled for registers (prior to unroll-and-jam) to prevent register spill. In matrix multiplication (dense and sparse) typically a tile of C is placed in registers because accesses to C (unlike A and B) require a load and a store. Since accesses to C are determined by loop iterators i and j, loops i and j are tiled so that a $T_i^r \times T_j^r$ tile of C fits in registers (superscript r indicates tiling for registers). To ensure sufficient scratch registers exist for the values of A and B, we also satisfy the constraint: $T_i^r * T_j^r + min(T_i^r, T_j^r) + 1 \leq num\_reg$; here $T_i^r * T_j^r$ indicates the number of registers required for the 2D tile of C, while $min(T_i^r, T_j^r) + 1$ is the number of scratch registers required for A and B. This limits the number of live scalars at any one time such that they all fit in registers preventing spill. Hereon, in the text, the tile size superscripts are removed for readability as we will discuss register tiling exclusively (i.e. T_i refers to T_i^r).

3.3 Unroll and Enumerate

Loop tiling is not sufficient for register reuse, the tiled loop should also be unroll-and-jammed to expose reuse to the compiler and to determine register lifetimes. In sparse register tiling, we replace unroll-and-jam with unroll-and-sparse-jam. The first step of unroll-and-sparse-jam is to unroll

the inner loops that are tiled for registers (T_i and T_j). When unrolling the SpMM code (Listing 1), conditionals break up the innermost loop body preventing the compiler from assigning common accesses of B to the same registers. To solve this, sparse register tiling enumerates all possible code paths in the innermost loop body; by enumerating the code paths, common accesses to B become statically known enabling register reuse for B.

An *Enumerated block* is a group of FMA instructions (a basic block, enclosed by a guard clause, e.g. Line 7 in Listing 3) representing a path in the innermost loop body that is computed with no register spill. Register spill occurs when the number of overlapping live ranges at any one time during the execution of the block exceeds the number of registers on a given architecture. A live range is the span of instructions from the first use of a value to its last use. For SpMM the instructions in an enumerated block run independently without violating correctness.

We show an enumerated block with its iteration space $\mathcal{I}_e$, access functions f, g, and h, and its guard clause. The iteration space $\mathcal{I}_e$ contains the iterators of the unrolled loops, i.e., i and j, and a single iteration of k. Iterators i, j, and k have a lower bound of zero and upper bounds of T_i, T_j, and K, respectively. The data in A, B, and C are accessed with $A[g(\mathcal{I}_e)]$, $B[h(\mathcal{I}_e)]$, and $C[f(\mathcal{I}_e)]$, respectively. Access functions in Listing 1 are $f(\mathcal{I}_e) = i*N+j$, $g(\mathcal{I}_e) = i*K+k$, and $h(\mathcal{I}_e) = k*N+j$, where N and K are columns of C and A respectively. The guard clause shows when an enumerated block executes and depends on nonzero locations in A.

Table 1 shows the compact form of all the enumerated blocks for Listing 3. A row in the table shows an enumerated block and its associated guard clause conditions, iteration space, and access functions. The number of rows in Table 1 corresponds to all combinations of truth states for the $T_i = 3$ conditionals after unrolling i. Lines 7, 10, and 13 in Listing 2 show the three conditionals for $T_i = 3$. After enumeration, the three conditions result in $2^{T_i} - 1$ possibilities, ignoring the empty case, each corresponding to an enumerated block as shown in Table 1. For example, for $e2$ in the table, the conditional columns correspond to truth states of values A in line 14 in Listing 3. The unrolled statements in lines 14–18 are the result of unrolling the iteration space of $e2$ as shown in the second column of Table 1. The access function columns show unrolled accesses to the three arrays, for example, A[ii+0,j] and A[ii+2,j] in lines 14–18 in Listing 3 correspond to $g(0, j) \cup g(2, j)$ in the g column in Table 1.

The iteration space and access functions in an enumerated block also show operations on nonzero elements, i.e. *required operations* and potential reuse opportunities. For example for $e2$ in Table 1, the iteration space is: $i \in \{0, 2\} \wedge 0 \leq j < T_j = 2$. For $e2$, based on the iteration space ($\mathcal{I}_e$) there are four required FMA operations. However, the number of unique accesses determined by functions g and h is less than four, meaning values from matrices A and B are being reused. All the enumerated blocks operate on the same tile of C, i.e. $f(i, j)$ where $0 \leq i < T_i = 3 \wedge 0 \leq j < T_j = 2$. Therefore the tile of C can be pre-loaded into registers and used for multiple executions of a enumerated block, creating reuse in C. For $e5 - e7$, since there exists no reuse in B (the h access function), they become a form of 1D tiling, creating degenerate cases.

3.4 Sparse-Jam Solver

The number of enumerated blocks grows exponentially with T_i (is 2^{T_i}) motivating the need for the sparse-jam solver to reduce the number of enumerated blocks generated. It is often impractical to generate unrolled code for all enumerated blocks as it negatively impacts performance due to inefficient use of the instruction cache and excessive branching. The sparse-jam solver replaces the set of all enumerated blocks (E) with a smaller set R. The code generated using set R uniquely covers all required operations, guaranteed with a mathematical model, and leads to a generated code size that fits in the instruction cache. To find R, we use a cost model to estimate the execution time of enumerated blocks, and their *merged* variations, at compile-time. Our cost model also accounts

Table 1. The list of enumerated blocks in Listing 3 ($T_i = 3, T_j = 2$) after unrolling loop iterators i and j. The conditional column refers to the if condition for each enumerated block. Loop iterators i, j, and k represent the iteration space of a enumerated block ($\mathcal{I}_e$). Access functions f, g and h indicate accesses to a tile of matrices C, A, and B, respectively. A' points to the beginning of a tile in A. K is the upper bound for k.

	Conditional						
	A'[0,k] != 0	A'[1,k] != 0	A'[2,k] != 0	Interation Space ($\mathcal{I}_e$) $0 \leq k < K$	$f(i,j)$	$g(i,k)$	$h(k,j)$
e1	1	1	1	$i \in \{0,1,2\} \wedge 0 \leq j < T_j$	$f(0,j) \cup f(1,j) \cup f(2,j)$	$g(0,k) \cup g(1,k) \cup g(2,k)$	$h(k,j)$
e2	1	0	1	$i \in \{0,2\} \wedge 0 \leq j < T_j$	$f(0,j) \cup f(2,j)$	$g(0,k) \cup g(2,k)$	$h(k,j)$
e3	1	1	0	$i \in \{0,1\} \wedge 0 \leq j < T_j$	$f(0,j) \cup f(1,j)$	$g(0,k) \cup g(1,k)$	$h(k,j)$
e4	0	1	1	$i \in \{1,2\} \wedge 0 \leq j < T_j$	$f(1,j) \cup f(2,j)$	$g(1,k) \cup g(2,k)$	$h(k,j)$
e5	1	0	0	$i \in \{0\} \wedge 0 \leq j < T_j$	$f(0,j)$	$g(0,k)$	$h(k,j)$
e6	0	1	0	$i \in \{1\} \wedge 0 \leq j < T_j$	$f(1,j)$	$g(1,k)$	$h(k,j)$
e7	0	0	1	$i \in \{2\} \wedge 0 \leq j < T_j$	$f(2,j)$	$g(2,k)$	$h(k,j)$

a) Identifying enumerated blocks in two sample tiles ($T_i = 3, K = 7$) of A from target matrices. (Dots represent nonzeros in A)

b) Enumerated blocks (E) with their frequency. (Dots represent accesses to C, hence the tile dimensions are $T_i = 3, T_j = 2$)

c) The set of merged enumerated blocks (R) produced by the *Sparse-Jam* solver.

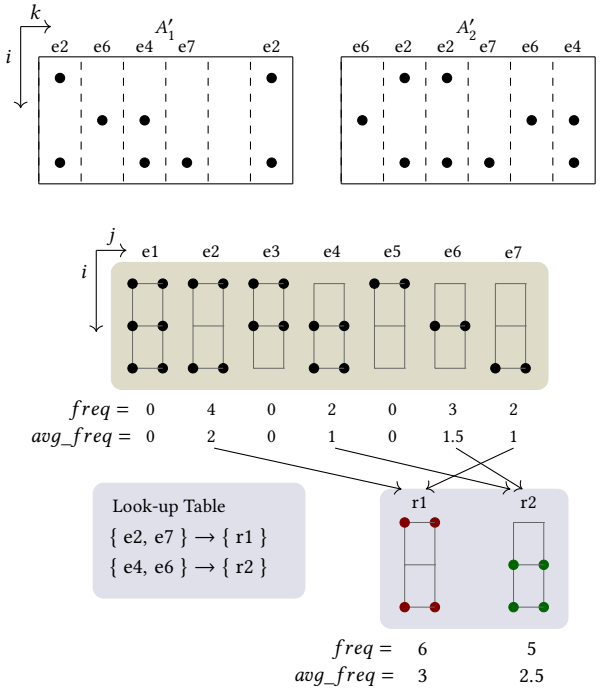

Fig. 6. Inputs and outputs of the sparse-jam solver for the unroll-and-sparse-jam example: a) shows two sample tiles A'_1 and A'_2 extracted from the target sparse matrices with $T_i = 3$. Each column is labeled with its associated enumerated blocks. b) shows all enumerated blocks for $T_i = 3, T_j = 2$, forming set E. The frequency information of each enumerated block in A'_1 and A'_2 is also shown. Finally, c) shows that the *sparse-jam* solver takes E and its associated frequencies and produces a set of merged enumerated blocks (R) and its corresponding lookup table to be used by the scheduler/data compressor.

for the frequency at which the enumerated blocks appear in sparse ML weight matrices. Figure 6 shows how sparse-jam generates the compact code in Listing 4 from the enumerated code in Listing 3 and is used in the following to first explain the frequency and cost model of enumerated blocks and then demonstrate the sparse-jam mathematical model.

Frequency. The frequency of enumerated block e, or $freq(e)$ is an extracted statistic from profiling an extensive set of tiles from target matrices and represents how often the enumerated block e would execute if generated. The target matrices in this work are composed of a large benchmark of pruned ML weight matrices. To compute the frequency information, the frequency extractor in sparse register tiling iterates over tiles of weight matrix A. An enumerated block is uniquely identified using accesses to A, i.e. g, thus iterating over matrix A accurately counts the frequency of each enumerated block, shown with $freq(e)$. Also, avg_freq is $frequency$ divided by the number of tiles sampled.

Figure 6a, shows two example tiles A'_1 and A'_2 with $T_i = 3$. The columns in each tile are labeled with their associated enumerated block. For $T_i = 3$ a maximum of 7 unique enumerated blocks exists (neglecting the empty block) and are stored in E. Each enumerated block corresponds to one row of Table 1. Figure 6a shows the access pattern of A, i.e. access function g, and Figure 6b shows accesses to C via access function f. While $T_i = 3$ is chosen for brevity, sparse-jam supports larger values of T_i. The occurrence of each $e \in E$ in the two tiles in Figure 6a is shown with $freq$ and avg_freq below tiles in Figure 6a. In practice, sparse-jam computes $freq$ for all tiles in the target weight matrices.

Cost Model. The cost model in sparse-jam provides a compile-time estimation of the execution time of individual enumerated blocks and their merged variants. Frequency is also accounted for in the overall cost. All possible variations of merging enumerated blocks, including individual enumerated blocks, are stored in G which has m members. The overall cost of $G_i \in G$ is as follows:

$$overall\_cost(G_i) = exec\_cost(G_i) * \sum_{e \in G_i} avg\_freq(e) + overhead \quad \forall i = 1, ..., m \tag{1}$$

where $exec\_cost(G_i)$ is the cost of a single execution of the variant results from merging all enumerated blocks in G_i. When merged, the enumerated blocks in G_i create one enumerated block with its iteration space and its access functions being the union of the iteration space and access functions of all enumerated blocks in G_i. The merging leads to padding in A and hence redundant computation. To compute $exec\_cost(G_i)$, we use two architecture-specific parameters to obtain the cost per operation for loading A, performing the FMA, and the cost of loading B. The $overall\_cost$ in Equation 1 includes avg_freq to account for the frequency at which the enumerated block in G_i appears in the input tiles. The $overhead$ term represents the incremental branching cost of generating an additional enumerated block (irrespective of how many times it runs or the number of required operations within enumerated block). The two architectural parameters for $exec\_cost(G_i)$ along with $overhead$ are obtained empirically.

Using Figure 6 as an example, if we assume $G_1 = \{e2, e7\}$, $exec\_cost(\{e2, e7\}) = 0.3$, $avg\_freq(e2) = 4$, $avg\_freq(e7) = 2$ and $overhead = 0.7$ then $overall\_cost(G_1)$ would be: $0.3 * (2 + 1) + 0.7 = 1.6$. Figure 6b to figure 6c shows that $e2$ and $e7$ ($e2, e7 \in E$) get merged to become $r1$ ($r1 \in R$). The merging causes redundant operations as a result of the fact that $e7$ only requires operations for: $i = 2 \wedge j \in \{0, 1\}$, however, $r1$ computes: $i \in \{0, 2\} \wedge j \in \{0, 1\}$. In other words, instead of executing $e7$, we execute $r1$ and perform redundant operations for: $i = 0 \wedge j \in \{0, 1\}$. If {e2} and {e7} were not merged and executed independently then because $exec\_cost(\{e2\}) = 0.3$ and $exec\_cost(\{e7\}) = 0.2$, the overall cost for $e2$ and $e7$ would be: $(0.3 * 2 + 0.7) + (0.2 * 1 + 0.7) = 2.2$. Hence in this example, it is profitable to merge $e2$ and $e7$ despite the redundant computation.

Mathematical Model. We use a mathematical model, an integer linear optimization problem, to search the space of cost-efficient merged enumerated block combinations while satisfying instruction cache limits and correctness. The model finds the set of merged groups with minimal cost, R, subject to two constraints ensuring correctness and compact code size. The input to the model is G,

i.e. the list of m merging variations of enumerated blocks. Only enumerated blocks with nonzero frequencies are considered. The sparse-jam mathematical model finds a set R with a minimum cost using:

$$Min \sum_{i=1}^{m} x_i * overall\_cost(G_i) \tag{2}$$

where x is the boolean unknown and $x_i = 1$ shows that the merged enumerated block resulting from G_i is selected as a member of R. The *covering constraint* ensures that the enumerated blocks in the target matrix are included exactly once in the final set R as shown in Equation 3:

$$\sum_{i=1}^{m} x_i * M_{ij} = 1, \forall j = 1, ..., z \tag{3}$$

where z is the number of nonzero enumerated blocks, which can be up to 2^{T_i}. Matrix M is $m \times z$ and M_{ij} is set to one when enumerated block j is part of G_i. The *code size* constraint ensures the size of the generated code for set R is smaller than the instruction cache size, i.e., s_{icache}. The constraint is:

$$\sum_{i=1}^{m} x_i * l_i \leq s_{icache} \tag{4}$$

where the code size for group i is shown with l_i.

To handle larger values of T_i and the exponential growth in the number of possible merged enumerated blocks (m), we instead use a pruned set of merged enumerated blocks as input. Given the relatively low number of vector registers on a common CPU, usually ≤ 32, we only test T_i values up to 8. If T_i is above 8, to have a 2D register tile of C ($T_i \times T_j$), T_j has to reduce, leading to limited register reuse. Even for $4 < T_i \leq 8$ pruning is required. We partition the enumerated blocks to a maximum of 16 partitions, each partition contains enumerated blocks with similar accesses to A. For each partition i, all possible merging of enumerated blocks are stored in the partition set GP_i. The union of all sets in GP_i is passed to the sparse-jam solver as G. Each member of GP_i is labeled with its frequency. Members with the highest frequency across different partition sets are also considered for merging and are added to G to be considered by the solver.

The output of the sparse-jam solver is then converted to a lookup table, which shows how the enumerated block should be merged, i.e. the surjective mapping: $E \rightarrow R$. The scheduler needs this information to map enumerated blocks to a set of generated merged enumerated blocks supported by the executor. In our example, since x_1 and x_4 are selected, the enumerated blocks in each group get mapped to the union of the group, i.e. $\{e2, e7\} \rightarrow \{r1 = e2 \cup e7\}$, the lookup table is in Figure 6c.

3.5 Code generation

In the final step of sparse register tiling, as shown in Figure 4, codes for a scheduler, a data compressor, and an SpMM executor are generated. The scheduler and the data compressor are part of a pre-processing phase and run before the SpMM executor code. The scheduler in the pre-processing step finds the corresponding mapping to R for an input matrix A (compressed into Atk and Atp to be used by the executor). The compressor takes the output of the scheduler to also compress the values of A (into Atv). The executor code performs the SpMM on the compressed matrix, using the computation order from the pre-processing phase.

3.5.1 Scheduler and Data Compressor Code Generation. The scheduler takes as input the sparse matrix A and the look-up table from the sparse-jam solver and provides as output the computation order of merged enumerated blocks. The scheduler code is shown in Listing 5. It loops over row

blocks of height T_i in A and for each row block it visits its columns (iterator k). For each of these columns, accesses to A are used to find the associated enumerated block, $e \in E$. Using the lookup_table created by sparse-jam, each enumerated block is mapped to a merged enumerated block $r \in R$ (line 4 in Listing 5). The column indices (or iterations k) for each r are stored in 3D tensor, location, and then flattened to Atk and Atp. Atp[ii] is an array of pointers to Atk for row block i. Atp[ii][id - 1] points to the start of column indices related to a merged enumerated block id in row block i, and Atp[ii][id] marks the end of it.

```
1 scheduler(A, {Ti,M,K}, lookup_table){
2  for (ii = 0; ii < M; ii += Ti) {
3   for (k = 0; k < K; k++) {
4    r = lookup_table(A[ii:ii+Ti, k]);
5    location[ii][r.id].append(k); }}
6  Atk, Atp = flatten(location);
7  return Atk, Atp; }
```

Listing 5. Scheduler code

```
1 data_compressor(A, {Ti,M,K}, {Atk,Atp}, R){
2  for (ii = 0; ii < M; ii += Ti) {
3   for (r: sorted_by_id(R)){
4    for (p=Atp[ii][r.id-1]; p<Atp[ii][r.id]; p++) {
5     for (offset: r.g) { // r.g shows accesses to A
6      Atv.append(A[ii * K + offset, Atk[p]]); }}}}
7  return Atv; }
```

Listing 6. Data Compressor code

The data compressor re-stores nonzero values in A contiguously based on the execution order of merged enumerated blocks provided by the scheduler and produces Atv. The compression improves spatial locality while simultaneously reducing the amount of memory required for the computation. The template code for the compressor is shown in Listing 6, where R is the set of generated merged enumerated blocks with their associated access functions. As shown in Listing 6, the order in which the data is accessed is determined using the output of the scheduler (Atp, Atk) as well as accesses to A (access function g), for each of the merged enumerated blocks.

```
1  for (int ii = 0; ii < M; ii += 3) {
2   for (int jj = 0; jj < N; jj += 2) {
3    for (int p = Atp[ii][0]; p < Atp[ii][1]; p++) {
4     int k = Atk[p];
5     // r1, i = {0, 2}, j = {0, 1}
6     C[ii+0, jj+0] += A[ii+0, k] * B[k, jj+0];
7     C[ii+0, jj+1] += A[ii+0, k] * B[k, jj+1];
8     C[ii+2, jj+0] += A[ii+2, k] * B[k, jj+0];
9     C[ii+2, jj+1] += A[ii+2, k] * B[k, jj+1]; }
10   for (int p = Atp[ii][1]; p < Atp[ii][2]; x++) {
11    int k = Atk[p];
12    // r2, i = {1, 2}, j = {0, 1}
13    C[ii+1, jj+0] += A[ii+1, k] * B[k, jj+0];
14    C[ii+1, jj+1] += A[ii+1, k] * B[k, jj+1];
15    C[ii+2, jj+0] += A[ii+2, k] * B[k, jj+0];
16    C[ii+2, jj+1] += A[ii+2, k] * B[k, jj+1]; }
17 }}
```

Listing 7. Executor code for the lookup table in Figure 6c before scalar promotion and data compression (C[i,j]=C[i*N+j], A[i,k]=A[i*K+k] , and B[k,j]=B[k*N+j])

```
1  for (int ii = 0; ii < M; ii += 3) {
2   for (int jj = 0; jj < N; jj += 2) {
3    float c00 = C[ii+0, jj+0], c01 = C[ii+0, jj+1];
4    float c10 = C[ii+1, jj+0], c11 = C[ii+1, jj+1];
5    float c20 = C[ii+2, jj+0], c21 = C[ii+2, jj+1];
6    for (int p = Atp[ii][0]; p < Atp[ii][1]; p++) {
7     int k = Atk[p];
8     float a0 = Atv[0], b0 = B[k, jj+0];
9     float a2 = Atv[1], b1 = B[k, jj+1]; Atv+=2;
10    c00 += a0 * b0; c01 += a0 * b1;
11    c20 += a2 * b0; c21 += a2 * b1; }
12   for (int p = Atp[ii][1]; p < Atp[ii][2]; p++) {
13    int k = Atk[p];
14    float a1 = Atv[0], b0 = B[k, jj+0];
15    float a2 = Atv[1], b1 = B[k, jj+1]; Atv+=2;
16    c10 += a1 * b0; c11 += a1 * b1;
17    c20 += a2 * b0; c21 += a2 * b1; }
18   C[ii+0, jj+0] = c00; C[ii+0, jj+1] = c01;
19   C[ii+1, jj+0] = c10; C[ii+1, jj+1] = c11;
20   C[ii+2, jj+0] = c20; C[ii+2, jj+1] = c21; }}
```

Listing 8. Executor code for Figure 6c, after scalar promotion and data compression

3.5.2 Executor Code Generation. For the executor, the code generator will generate an unrolled code for each merged enumerated block $r \in R$. Listing 7 shows the generated executor code for the set R shown in Figure 6c. The code of each merged enumerated block r is inside a for loop that

iterates over all occurrences of r obtained from the scheduler, i.e. Atk and Atp. The code of a merged enumerated block is also transformed to operate on the compressed matrix A followed by scalar promotion and vector intrinsic generation to enable vectorization. Listing 8 shows the executor code after scalar promotion, operating on compressed nonzero values, Atv, which is populated by the data compressor. As shown in Listing 8, accesses to C, A, and B are unrolled, enabling scalar promotion. Lines 3–5 in Listing 8 show scalar promotion on accesses to C and lines 8–9 in Listing 8 show scalar promotions on accesses to A and B for $r1$. Finally, FMA operations are performed on scalar values, enabling the use of vector units, e.g. both FMA operations in line 10 in Listing 8 are vectorized and performed using a single vector FMA.

3.5.3 Multi-Threaded ML Inference. To use thread-level parallelism, the scheduler assigns a row block with height T_i to a thread. We use the number of nonzeros in a panel to balance workloads between threads. The row blocks are scheduled to threads dynamically similar to [Gale et al. 2020]. The row block scheduling is also done as part of the pre-processing in sparse register tiling. For ML inference using a pre-processing step to pre-pack weight matrices is a common technique to boost performance, frequently done during model compilation or during inference server setup.

3.6 Safety

The associativity property of FMAs in SpMM allows for an arbitrary reordering of operations in single-threaded executions. Thus the tentative change in computation order from unroll-and-sparse-jam, scheduler, and data compression is safe. The covering constraint ensures all required statements are covered in the jammed code, hence any change in the number of operations from unroll-sparse-jam is safe. For multi-threading, the executor runs row blocks in parallel. Since row blocks write to disjoint locations of C, write-after-write dependence is not violated.

4 EXPERIMENTAL RESULTS

This section evaluates the performance of sparse register. We first show the efficiency of sparse register tiling for Deep Learning Matrix Collection [Gale et al. 2020]. Then MobileNetV1 is used as a case study to show the efficiency of our method on an edge device.

4.1 Experimental Setup

Target architecture: For testbeds we use an Intel(R) 20-core server CPU with AVX512, Xeon(R) Gold 6248 CPU (2.50GHz, 1MB L2 Cache, 28160K L3 Cache), as well as a common edge processor Armv8-A 4 core CPU, ARM Cortex-A72 (1.5 GHz, 1MB shared L2 Cache, Raspberry Pi 4B). The Intel CPU is capable of 2 vector FMAs per cycle and 2 vector loads per cycle [Abel and Reineke 2019], while the ARM CPU is capable of 1 vector FMA per cycle and 1 vector load per cycle [ARM 2015]. AVX512 has 512-bit vectors or 16 single precision values, while Armv8-A has 128-bit vectors or 4 single precision values. Both have 32 vector registers. Experiments are performed using the Google Benchmark[1], we take the median of 7 repetitions. Google Benchmark initially runs warm iterations to determine a statistically stable number of iterations per repetition, as a result, our tests are done using a warm cache. For each experiment, the *required FLOPs*, i.e. operations required on nonzero elements, is calculated per run and is twice the number of nonzeros times the number of B columns and is common for all implementations.

Software setup on Intel: We compare our method against ASpT [Hong et al. 2019], MKL GEMM [Intel 2022], MKL SpMM using a CSR storage format, MKL SpMM using a BSR storage format, TACO [Kjolstad et al. 2017; Senanayake et al. 2020] using the schedules from [Senanayake et al. 2020], and the latest version of SpMM from the locality-based codelet mining inspector/executor (LCM

[1] https://github.com/google/benchmark

I/E) [Cheshmi et al. 2022]. For ASpT, the latest version is compiled using their default build options. ASpT assigns a row block to each thread for thread parallelism. The default row block size in ASpT is set to a large value, suitable for SuiteSparse matrices that often have a large number of rows. However, the number of rows in the DLMC matrices tested is small (mean: 899, median: 512) leading to limited parallelism with the default ASpT setting. Thus, we execute ASpT with both the default and adaptive row block size, i.e. rows divided by the number of threads, and report the best. For MKL we use version 2021.4.0. LCM I/E only supports double precision so we compare it with double precision code generated using sparse register tiling. All methods on Intel are compiled using GCC v.11.2 with the -O3 flag, however, numbers related to performance counters are reported with GCC v.8.3 for compatibility with PAPI [Terpstra et al. 2010] on our server.

Software setup on ARM: We compare our method against the SpMM kernel found in XNNPACK [Elsen et al. 2020; Google 2022], a well-known ML inference library for CPUs, as well as to the SGEMM in the ARM Compute Library [ARM 2022a] (version 22.08). We also tested the SGEMM in ARM Performance Libraries [ARM 2022b] (version 22.1) but found that the Compute Library was consistently faster for the tested matrices and therefore omitted the ARM Performance Library results. All methods on ARM are compiled using GCC v.10.2 with the -O3 flag.

Sparse register tiling setup: Our code generator is implemented in Python and emits C++ code with vector intrinsics for ARM and Intel. The MOSEK package [ApS 2022] is used to solve the mathematical program in the sparse-jam solver. Since all of the pruning methods in the DLMC are unstructured and result in highly random sparsity patterns we find that a common set of scheduler-executor pairs generated by sparse register tiling work well across the board. We generate four different mappings from E to R (lookup tables) where the resulting mapping sizes are, $size(E) \rightarrow size(R)$: $15 \rightarrow 15$ ($T_i = 4$), $255 \rightarrow 19$ ($T_i = 8$), $15 \rightarrow 9$ ($T_i = 4$) and $255 \rightarrow 19$ ($T_i = 8$). The first two are targeted at lower sparsity ratios (60% - 80%) while the latter two are targeted at higher sparsity ratios (80% - 95%). Then 2 executors are generated per mapping by changing T_j, when $T_i = 4$ we use T_j values of 4 and 6, and when $T_i = 8$ we use T_j values of 2 and 3. Because some matrices don't have multiple of 6 row counts, $T_i = 6$ is not used to simplify the implementation. We use an empirically generated heuristic to select among the eight generated executor-scheduler pairs, with $T_i = 8$ being favored for smaller B column counts and $T_i = 4$ being favored for larger B column counts.

4.2 Deep Learning Matrix Collection Evaluation

We test our method on all 2396 matrices (for sparsity ranges of 60% to 95%) from the Deep Learning Matrix Collection [Gale et al. 2020], which contains matrices from pruned Transformer and ResNet50 Models. The ResNet50 matrices come from unfolding the convolution filters, i.e. assuming the input has undergone the im2col operation to convert the convolution to matrix multiplication. The weight matrices were pruned using a variety of unstructured pruning methods: variational dropout [Kingma et al. 2015], l_0 regularization [Louizos et al. 2018], random [Zhu and Gupta 2017] and an extended version of magnitude pruning [Zhu and Gupta 2017]. This results in a total of 59 models in the dataset. Matrices with above 95% sparsity are not tested because models with this level of sparsity regularly lead to substantial drops in accuracy [Gale et al. 2019] also unstructured matrices with 95%+ sparsity typically only have one nonzero per enumerated block making our method equivalent to 1D tiling.

Overall sparse register tiling achieves a geometric mean single-threaded speedup of 2.24×, 1.96× and 1.99× and multi-threaded (20 Threads) speedup of 2.65×, 1.72× and 3.23× over MKL SGEMM, MKL SpMM (CSR) and ASpT respectively. For ARM, Sparse Register Tiling achieves a geometric mean single-threaded speedup of 1.89× and 1.24×, and multi-threaded speedup of 1.86× and 1.97× respectively over ARM Compute Library SGEMM and XNNPACK SpMM. Figure 7 and Figure 8 show the overall performance of sparse register tiling for all 2396 DLMC matrices tested.

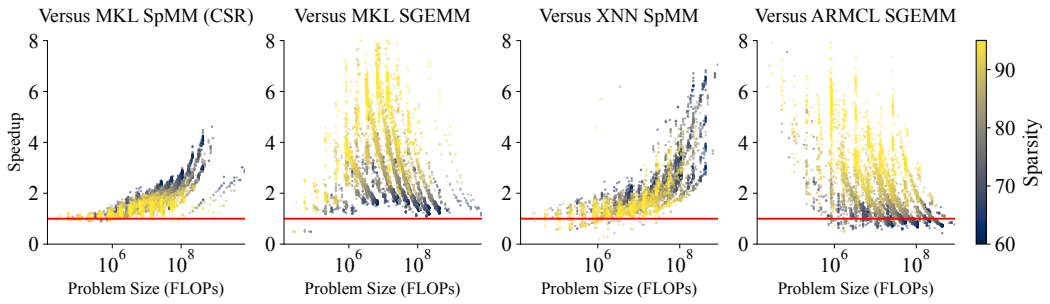

Fig. 7. Speedup of sparse register tiling (B Columns = 32, 128, 256, 512) over dense and the sparse baseline for the respective architectures using 20 threads for Intel and 4 threads for ARM.

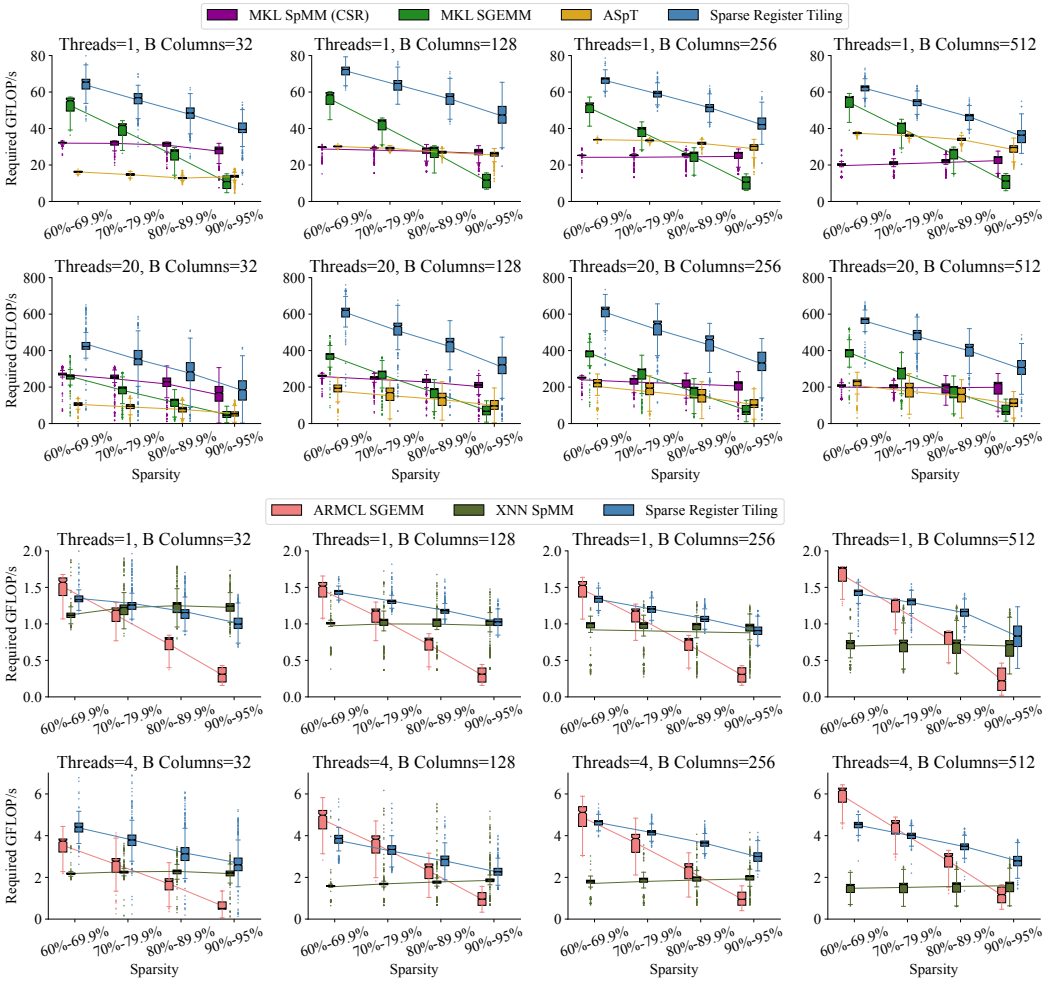

Fig. 8. Breakdown by B column sizes and thread count for Intel (rows 1–2) and ARM (rows 3–4). Center of boxes represent median values, while the lines represent mean GFLOP/s.

4.2.1 Overall Performance and Storage Cost. Figure 7 shows the speedups versus the fastest sparse and dense baselines for both Intel and ARM, MKL SpMM (CSR), and XNNPACK respectively, for multi-threaded runs. Figure 8 shows the throughput broken down by B column counts and the number of threads. In each experiment in Figure 8, the memory footprint, i.e. the collective size of matrices A, B, and C, is between 26KB–130MB. For the experiments in Figure 8, the memory footprint is larger than Intel's L3, Intel's single-core L2, and ARM L2 caches in 0.8%, 90%, and 90% cases respectively. The performance of sparse register tiling is better than sparse and dense baselines due to lower loads-per-FMA and redundant operations, respectively. Compared to MKL SpMM (CSR) and XNNPACK SpMM, respectively, in 66% and 82% of matrices, for which sparse register tiling provides over 1.5 times speedup, fall within sparsity range is within 60–80%. This is because for lower sparsity ranges, the execution time of existing sparse implementations is dominated by indexing loads and (re)loading values of B (thus higher loads-per-FMA). ASpT on Intel and XNNPACK SpMM on ARM use an efficient 1D register tiling approach. As shown, sparse register tiling outperforms ASpT and XNNPACK SpMM in 100% and 78% of matrices, respectively. Compared to the dense baselines, in 88% and 99% of matrices where sparse register tiling is over 1.5 times faster, the sparsity range is within 70–95%, for Intel and ARM respectively. This is because dense baselines do more redundant FLOPs, i.e. operations on zeros, compared to sparse register tiling. We also compared sparse register tiling with MKL's implementation of register blocking (BSR) with block sizes 2×2, 4×4, and 8×8 and TACO. For MKL BSR, 4×4 provides the best overall performance for DLMC matrices and is only faster than MKL SGEMM and SpMM (CSR) in 3.16% and 4.19%, respectively. Our experiments show the BSR (4x4) implementation performs on average 5.69× more computation than sparse register tiling. TACO is faster than MKL SpMM in less than 1% of DLMC matrices and is consistently slower than sparse register tiling because TACO uses 1D register tiling and does not exploit cache-level locality. For these reasons, both MKL BSR and TACO are not included in the plots.

Figure 9 compares the required storage size in sparse register tiling vs the CSR storage format (all normalized over a dense baseline). Sparse register tiling has a lower storage cost than CSR in 60% of matrices because it only needs one indirection per merged enumerated block; CSR has an indirection per nonzero. The effect is more pronounced at lower sparsity values where there is an increased number of nonzeros per enumerated block. As the sparsity increases, CSR requires less storage for some cases. This is because of the overhead of cache-level tiling in sparse register tiling.

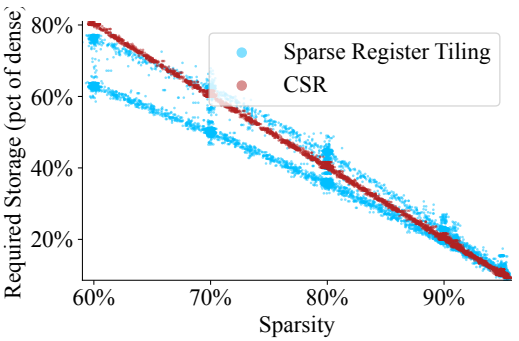

Fig. 9. Bytes required to store A normalized over the size of the matrix if treated as dense (lower is better).

4.2.2 Comparing with Sparsity-specific Methods. In this section, we compare the performance of sparse register tiling with LCM I/E, a sparsity-specific technique, on DLMC and SuiteSparse matrices. Since LCM I/E is implemented in double precision, we use the double precision version of sparse register tiling for comparison and all GFLOP/s are reported as double precision FLOPs. The performance of double precision MKL DGEMM and SpMM are also provided as a reference. Figure 10a compares the performance of the two implementations for DLMC matrices when B Columns=128. As shown, sparse register tiling is faster than LCM I/E, MKL SpMM, and MKL DGEMM for DLMC with a geometric mean speedup of 2.06×, 1.90×, and 2.87×, respectively.

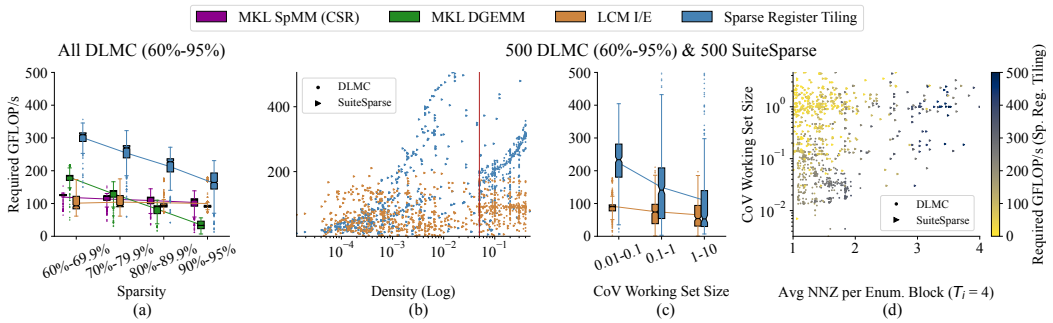

Fig. 10. (a) - (c) Double precision performance of sparse register tiling vs LCM I/E, (d) Double precision performance of sparse register tiling vs CoV of working set size and average nonzero per enumerated block, All plots use B Columns=128 and Threads=20.

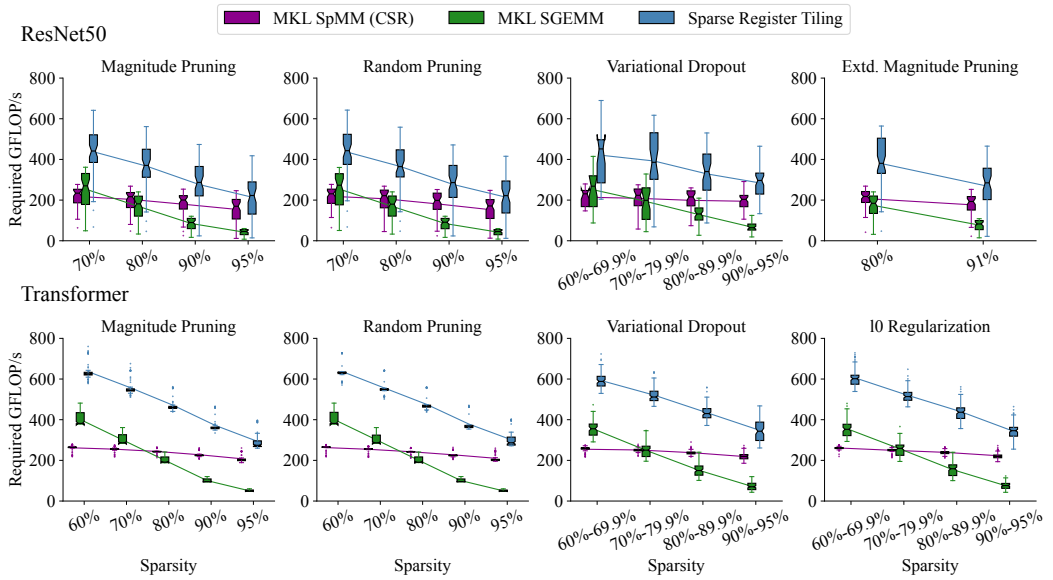

Fig. 11. Performance breakdown by model and pruning method (B Columns=128 and Threads=20)

Figure 10b shows the performance of LCM I/E and sparse register tiling for DLMC and SuiteSparse matrices. The evaluation is performed for 500 randomly-selected matrices in SuiteSparse and 500 randomly-selected matrices from DLMC. The matrices are sorted based on density which acts as a soft separator, some SuiteSparse matrices have a density greater than 5% (the red line) while none of the tested DLMC matrices have a density less than 5%. LCM I/E performs better than sparse register tiling in 37.8% of SuiteSparse matrices while it is faster only in 7.9% of DLMC matrices.

To explain the trends in Figure 10b, we describe how LCM I/E and sparse register tiling use sparsity information to reuse data in caches and registers. Based on the sparsity pattern of A, LCM I/E generates three different types of codelets, BLAS, partially strided codelet (PSC) type I, and type II (PSC I and PSC II). These codelets differ in type based on the number of unstrided (random) access functions in their code. LCM I/E enables 2D register reuse for BLAS and PSC I codelets because they can have at least two (zero) strided access functions. However, for PSC II, where accesses to both A and B are unstrided, LCM I/E only enables 1D register reuse because only accesses to C are known at compile-time. We analyzed the codelets generated from LCM I/E. Over 65% of its codelets

for SuiteSparse matrices is of type PSC I, while in matrices from DLMC 96% of codelets are PSC II. Hence, LCM I/E does not enable efficient register tiling, i.e. 2D register reuse, in the DLMC. LCM I/E does enable cache reuse for codelet types. Sparse register tiling enables 2D register reuse for unstrided accesses, e.g. PSC II. This is done by unrolling the unstrided accesses, i.e. accesses to C, revealing accesses at compile-time, As a result, per Figure 10b, compared to LCM IE, sparse register tiling provides better performance for DLMC matrices.

The cache tiling strategy in sparse register tiling is not well-suited for SuiteSparse matrices. Per Section 3.2, sparse register tiling implements the tiling strategy used for dense matrices, i.e. fixed tile sizes, based on the fact that the coefficient of variation (CoV) of the working set is low in both DLMC and dense matrices. Thus our approach works better when CoV is low as shown in Figure 10c. LCM I/E is provided as a reference. From Figure 10b, sparse register tiling provides higher performance in some SuiteSparse matrices compared to the DLMC matrices. This is because these SuiteSparse matrices have more dense structures compared to unstructured DLMC matrices resulting in a higher average number of nonzeros per enumerated block per Figure 10d.

4.2.3 The Effect of the Pruning Method on Performance. Figure 11 shows the performance of sparse register tiling, MKL SpMM (CSR), and MKL SGEMM for all pruning methods in the DLMC, broken down by models. As shown, the performance of SpMM for matrices from ResNet50 is on average lower than those from the transformer model because the average row count in ResNet50 is lower than transformers, reducing thread-level parallelism. Figure 12 shows that the row counts of weight matrices from ResNet50 are often less than 250, which is smaller than the transformer model row counts. The highest variation in timing belongs to variational dropout, extended magnitude pruning, and $l0$ regularization pruning methods because the sparsity of each layer does not match the total model sparsity resulting in more variation in sparsity (nonzero count) between the weight matrices.

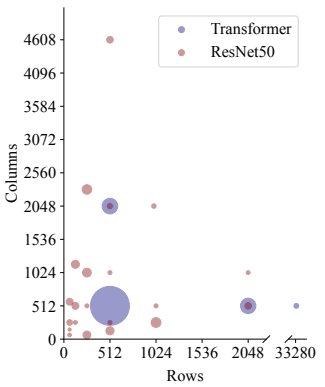

Fig. 12. Matrix shapes for DLMC.

4.2.4 Profiling and Analysis. Figure 13 shows the effect of unroll-and-sparse-jam and data compression on the obtained performance of sparse register tiling. Unroll-and-sparse-jam reduces the code size and branch miss-prediction, while data compression improves spatial locality in accesses to A. Figure 13 also shows that the effect of spatial locality improvement in A is diminished as the number of columns in B increases, leading to the data movement cost of B and C becoming dominant. Figure 15 shows that as sparse register tiling reduces loads-per-FMA, its performance increases relative to the baseline. We measure the number of loads and operations using PAPI [Terpstra et al. 2010] counters MEM_UOPS_RETIRED:ALL_LOADS and MEM_UOPS_RETIRED:ALL_STORES, respectively. As sparsity increases, the loads-per-FMA increase since the number of operations (nonzeros) per register tile decreases. Thus our method typically degenerates to the more standard 1D tiling around 95% sparsity, when the matrices are unstructured. The figure shows a loads-per-FMA threshold of 1, with the highest speedups being below this point.

Figure 14 shows execution time, redundant computation, and loads-per-FMA for matrices with uniform randomly generated sparsity patterns ranging from 30% to 99%. The MKL SGEMM execution time is always a constant line and is used as a baseline. Redundant FLOPs in sparse register tiling reduce as the sparsity ratio increases. This is because fewer nonzeros exist per enumerated block leading to high-frequency enumerated blocks dominating the degenerate cases, reducing the need for aggressive merging. Also for above 95%, sparse register tiling degenerates to 1D tiling in the case of uniform random sparsity closely matching the loads-per-FMA of ASpT.

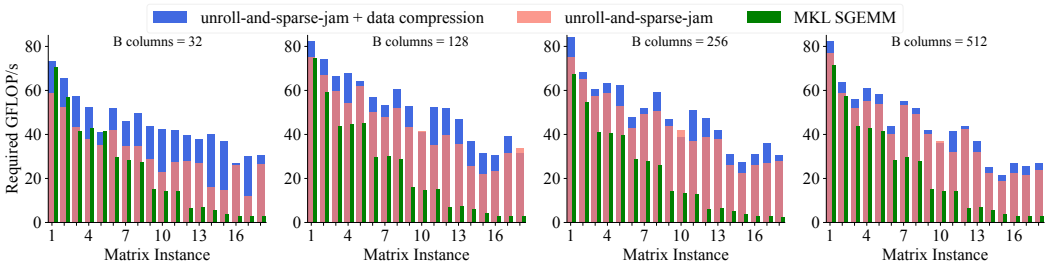

Fig. 13. The effect of unroll-and-sparse-jam and data compression on overall performance (Intel)

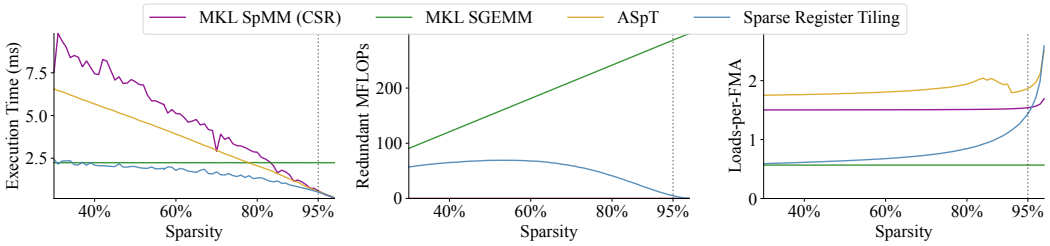

Fig. 14. A sweep of randomly generated sparsity patterns (784×784, B Columns = 256). (Left) Shows the execution time vs Sparsity. (Middle) Shows the number of redundant FLOPs computed by each method. (Right) Shows the number of loads-per-FMA per method.

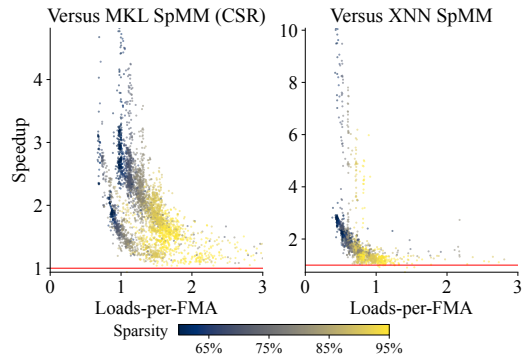

Fig. 15. Loads-per-FMA for the randomly selected subset of matrices from DLMC

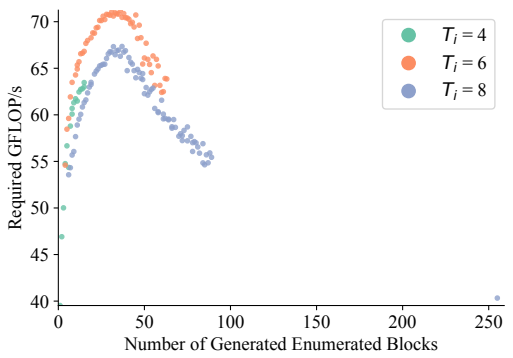

Fig. 16. Correlation between the number of generated enumerated blocks and performance (single-threaded) (random sparsity pattern 70%).

Figure 16 shows how the sparse-jam solver explores the space of code size and performance for a small SpMM where all matrices fit in the L1 cache. A is selected to be 48×128 and B has 32 columns. Three different T_i sizes are selected 4, 6, and 8 (all with a T_j of 2). Each is passed to the sparse-jam solver with different constraints on the number of generated enumerated blocks. Per Figure 16, the performance improves up until a point. From this point on, instruction-cache misses and branching overhead are no longer offset by the incremental reduction in redundant computations. Sparse-jam takes the code size threshold to ensure efficient use of instruction cache size.

4.3 Case Study: MobileNetV1 on the Edge

Table 2. MobileNetV1 Latencies

	1 Thread		4 Threads	
	XNNPACK	Sparse Reg.	XNNPACK	Sparse Reg.
Dense	142ms	–	75ms	–
70%	132ms	**103ms**	77ms	**36ms**
80%	95ms	**82ms**	54ms	**31ms**
90%	61ms	**55ms**	34ms	**26ms**

MobileNetV1 [Howard et al. 2017] is a well-known ML model for image classification on mobile or edge devices. Reducing latency and improving throughput is critical for these models as it can directly affect frame rate when processing video, the user experience, and/or battery life. We integrate our kernels into XNNPACK and use the bundled MobileNetV1 benchmark, and then execute it on the ARM target processor. Similar to [Elsen et al. 2020], we replace the 1×1 convolutions with our SpMM because they dominate computation time and can be directly implemented as matrix multiplication. The benchmark uses random weights and a random sparsity pattern for all layers (only sparsifying the 1×1 convolution layers). We also fuse the bias-add and Relu activation into the 1×1 convolutions, similar to XNNPACK's SpMM and SGEMM implementations. The results for four threads are in Figure 17. The vertical blue lines represent the sparsified 1×1 convolution layers. The overall execution times are shown in Table 2.

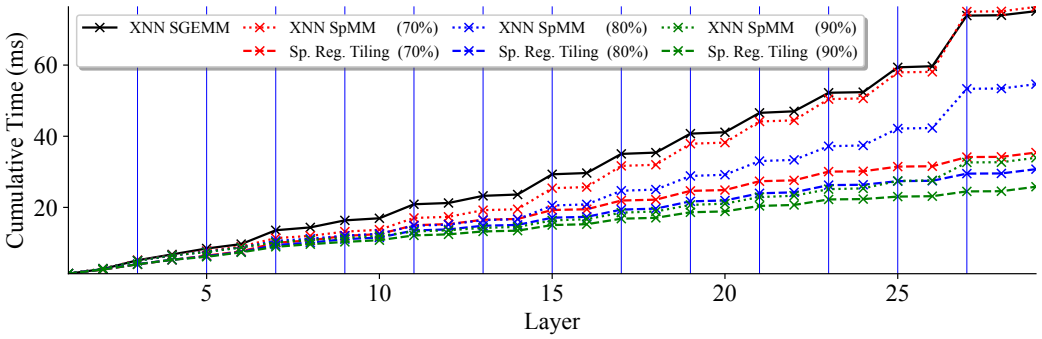

Fig. 17. MobileNET V1 on Cortex-A72 (4 threads)

5 RELATED WORK

Pruning in Neural Networks. Pruning methods in Neural Networks have gained great traction in recent years as it reduces model size and floating point operations. To turn FLOP reduction into performance, semi-structured or pattern-based pruning methods are proposed [Guo et al. 2020; Huang et al. 2022; Lagunas et al. 2021; Mao et al. 2017; Mishra et al. 2021; Zhu et al. 2019], these patterns are better fit to current computing hardware and software systems. Examples of structured pruning methods are, [Lagunas et al. 2021] that prunes to create blocks, [Mishra et al. 2021] which prunes to an N:M sparsity structure, and [Zhu et al. 2019] that prunes to create vectors. However, these pruning approaches often lead to accuracy loss compared to unstructured (fine-grained) pruning methods, with this tradeoff varying per architecture and application. PatDNN [Niu et al. 2020] is an example of a fine-tuning system for unstructured weight matrices. They select random pre-defined patterns and fine-tune a model to enforce the pre-defined patterns. This enables performance but it reduces accuracy and also requires fine-tuning the model. Sparse register tiling accelerates the performance of inference without compromising accuracy and without the need to fine-tune or re-prune an already pruned model.

Register tiling in libraries and compilers. 1D or 2D register tiling techniques are commonly used by compilers and libraries for matrix multiplication. Dense matrix multiplication implementations [Goto and Geijn 2008; Kung et al. 2021; Low et al. 2016] exploit all levels (register and cache) of the memory hierarchy using tiling methods. Due to the existing regularity in dense storage formats, micro-kernels [Low et al. 2016] are used for efficient 2D register tiling. Micro-kernels are hand-optimized code for fixed-size matrix multiplications that fit into registers of the target processor. Micro-kernels do not exploit sparsity.

A large class of sparse libraries and compilers exists that use efficient cache tiling and apply 1D register tiling to sparse codes. [Hong et al. 2019; Kurt et al. 2020] use analysis of working set sizes to perform cache tiling and then apply 1D register tiling. TACO [Kjolstad et al. 2017; Senanayake et al. 2020] is a tensor algebra compiler that creates tiles based on a given schedule, then leverages compiler directives to use vector processors and registers. For SpMM, this frequently leads to 1D register tiling. DCSC [Hong et al. 2019], DSCR [Hong et al. 2018] and CT-CSR [Rajbhandari et al. 2017] re-store each cache tile based on the sparse matrix row access order and then apply a 1D register tiling. COO and related storage formats like CSB [Aktulga et al. 2014; Buluç et al. 2009] have indirection on both rows and columns making even 1D register tiling challenging leading implementations that only exploit reuse on A. While these methods are efficient at alleviating the DRAM-to-cache bottleneck (or bottlenecks between cache levels), because of only applying 1D register tiling their performance is bottlenecked by data movements between L1 to registers.

Register blocking [Im and Yelick 2001] is the commonly used approach to enable 2D register tiling in SpMM. Libraries such as OSKI [Vuduc et al. 2005] SparTA [Zheng et al. 2022] and MKL [Intel 2022] use register blocking for a class of methods, including SpMM, to accelerate sparse computations. Register blocking is also successfully integrated into the existing compiler frameworks. Sparse polyhedral Framework [Mohammadi et al. 2019a,b; Zhao et al. 2022] and Sympiler [Cheshmi et al. 2017, 2019] apply register blocking [Venkat et al. 2015] to sparse matrix code through a sequence of data and code transformations. SpMM in ML frameworks is typically performed on sparse matrices stored in a dense format. However, these methods for unstructured matrices lead to a significant fill-in ratio, on average 2.69 for a small 2x2 block, removing the effect of FLOP reduction. Sparse register tiling uses 2D register tiling but does not introduce excessive fill-in.

Code compaction. Code compaction is necessary for register tiling to ensure efficient use of the instruction cache. For sparse codes, code compaction is used to compress the code extracted from the memory trace of the sparse kernel. Augustine et.al. [Augustine et al. 2019; Horro et al. 2023] unroll all operations of sparse matrix-vector multiplication and then generate a compact code of pre-defined multi-dimensional blocks with constant stride. Their approach is effective on structured or scientific matrices. For unstructured matrices, blocks with constant strided are not frequent even after data transformation. LCM I/E [Cheshmi et al. 2022] unrolls all operations of SpMM and then jams random accesses by grouping unstrided accesses into partially strided codelets. However, their approach does not allow for efficient 2D register reuse. Sparse register tiling uses unroll-and-sparse-jam to generate specialized code for random access, enabling register reuse.

6 CONCLUSION

This work proposes a novel sparse register tiling approach tailored to pruned neural networks. It introduces a novel unroll-and-sparse-jam code transformation that uses a mathematical model to generate compact code while enabling 2D register reuse. Sparse register tiling provides state-of-the-art performance on unstructured matrices arising from well-studied pruning techniques as well as wall-clock speedup in an end-to-end model. We plan on extending this work to support GPUs as well as other matrix multiplication-based kernels.

SOFTWARE AVAILABILITY

The artifact can be found here: [Wilkinson et al. 2023], the artifact source can also be found here: https://github.com/SpRegTiling/sparse-register-tiling/tree/pldi-artifact. The lastest version of the code can be found here: https://github.com/SpRegTiling/sparse-register-tiling/.

ACKNOWLEDGEMENTS

We would like to thank Faraz Shahsavan for his help in improving the paper. This work is supported by NSERC Discovery Grants (RGPIN-06516, DGECR00303), the Canada Research Chairs program, the Ontario Early Researcher Award, and the Digital Research Alliance of Canada (www.alliance-can.ca).

REFERENCES

Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1213–1222. https://doi.org/10.1109/IPDPS.2014.125

MOSEK ApS. 2022. MOSEK Optimization Suite. https://docs.mosek.com/10.0/pythonapi.pdf.

ARM. 2015. Cortex-A72 Software Optimization Guide Application Note UAN 0016A. https://developer.arm.com/documentation/uan0016/a/

ARM. 2022a. ARM Compute Library. https://github.com/ARM-software/ComputeLibrary

ARM. 2022b. ARM Performance Libraries. https://developer.arm.com/downloads/-/arm-performance-libraries

Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating Piecewise-Regular Code from Irregular Structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 625–639. https://doi.org/10.1145/3314221.3314615

Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

Steve Carr and Ken Kennedy. 1994. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. Program. Lang. Syst.* 16, 6 (nov 1994), 1768–1810. https://doi.org/10.1145/197320.197366

Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing Sparse Matrix Computations with Partially-Strided Codelets. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 32, 15 pages. https://doi.org/10.1109/SC41404.2022.00037

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3126908.3126936

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2019. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 62, 15 pages. https://doi.org/10.1109/SC.2018.00065

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

Xiaohan Ding, Guiguang Ding, Xiangxin Zhou, Yuchen Guo, Jungong Han, and Ji Liu. 2019. *Global Sparse Momentum SGD for Pruning Very Deep Neural Networks*. Curran Associates Inc., Red Hook, NY, USA.

E. Elsen, M. Dukhan, T. Gale, and K. Simonyan. 2020. Fast Sparse ConvNets. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 14617–14626. https://doi.org/10.1109/CVPR42600.2020.01464

Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. *CoRR* abs/1902.09574 (2019). https://doi.org/10.48550/arXiv.1902.09574 arXiv:1902.09574

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 17, 14 pages. https://doi.org/10.1109/SC41405.2020.00021

Google. 2022. XNNPACK. https://github.com/google/XNNPACK

Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating Sparse DNN Models without Hardware-Support via Tile-Wise Sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 16, 15 pages.

Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *J. Mach. Learn. Res.* 22, 1, Article 241 (jan 2021), 124 pages.

Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 66–79. https://doi.org/10.1145/3208040.3208062

Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. 2023. Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 160–171. https://doi.org/10.1145/3559009.3569668

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. https://doi.org/10.48550/arXiv.1704.04861 arXiv:1704.04861 [cs.CV]

Guyue Huang, Haoran Li, Minghai Qin, Fei Sun, Yufei Ding, and Yuan Xie. 2022. Shfl-BW: Accelerating Deep Neural Network Inference with Tensor-Core Aware Weight Pruning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1153–1158. https://doi.org/10.1145/3489517.3530588

Eun-Jin Im and Katherine Yelick. 2001. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Computational Science — ICCS 2001*, Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C. J. Kenneth Tan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–136. https://doi.org/10.1007/3-540-45545-0_22

Intel. 2022. Intel Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Accessed: 2022-08-17.

Durk P Kingma, Tim Salimans, and Max Welling. 2015. Variational Dropout and the Local Reparameterization Trick. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2015/file/bc7316929fe1545bf0b98d114ee3ecb8-Paper.pdf

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

H. T. Kung, Vikas Natesh, and Andrew Sabot. 2021. CAKE: Matrix Multiplication Using Constant-Bandwidth Blocks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 85, 14 pages. https://doi.org/10.1145/3458817.3476166

Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayyapan. 2020. Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 87, 14 pages. https://doi.org/10.1109/SC41405.2020.00091

François Lagunas, Ella Charlaix, Victor Sanh, and Alexander Rush. 2021. Block Pruning For Faster Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 10619–10629. https://doi.org/10.18653/v1/2021.emnlp-main.829

Shiwei Liu, Tianlong Chen, Xiaohan Chen, Zahra Atashgahi, Lu Yin, Huanyu Kou, Li Shen, Mykola Pechenizkiy, Zhangyang Wang, and Decebal Constantin Mocanu. 2021. Sparse Training via Boosting Pruning Plasticity with Neuroregeneration. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 9908–9922. https://proceedings.neurips.cc/paper/2021/file/5227b6aaf294f5f027273aebf16015f2-Paper.pdf

Christos Louizos, Max Welling, and Diederik P. Kingma. 2018. Learning Sparse Neural Networks through L_0 Regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1Y8hhg0b

Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (aug 2016), 18 pages. https://doi.org/10.1145/2925987

Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the Regularity of Sparse Structure in Convolutional Neural Networks. https://doi.org/10.48550/arXiv.1705.08922 arXiv:1705.08922 [cs.LG]

Lu Miao, Xiaolong Luo, Tianlong Chen, Wuyang Chen, Dong Liu, and Zhangyang Wang. 2022. Learning Pruning-Friendly Networks via Frank-Wolfe: One-Shot, Any-Sparsity, And No Retraining. In *International Conference on Learning Representations*. https://openreview.net/forum?id=O1DEtITim__

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. https://doi.org/10.48550/arXiv.2104.08378 arXiv:2104.08378 [cs.LG]

Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2019a. Extending index-array properties for data dependence analysis. In *Languages and Compilers for Parallel Computing: 31st International Workshop, LCPC 2018, Salt Lake City, UT, USA, October 9–11, 2018, Revised Selected Papers 31*. Springer, 78–93. https://doi.org/10.1007/978-3-030-34627-0_7

Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019b. Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 594–609. https://doi.org/10.1145/3314221.3314646

Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 907–922. https://doi.org/10.1145/3373376.3378534

Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. 2017. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/3037697.3037745

Carl Edward Rasmussen and Zoubin Ghahramani. 2000. Occam's Razor. In *Proceedings of the 13th International Conference on Neural Information Processing Systems* (Denver, CO) *(NIPS'00)*. MIT Press, Cambridge, MA, USA, 276–282.

Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1711, 12 pages.

Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (nov 2020), 30 pages. https://doi.org/10.1145/3428226

Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 521–532. https://doi.org/10.1145/2737924.2738003

Richard Vuduc, James Demmel, and Katherine Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series* 16 (01 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 26–26. https://doi.org/10.1109/SC.2002.10025

Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *Proceedings of the First International Conference on High Performance Computing and Communications* (Sorrento, Italy) *(HPCC'05)*. Springer-Verlag, Berlin, Heidelberg, 807–816. https://doi.org/10.1007/11557654_91

Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. *Register Tiling for Unstructured Sparsity in Neural Network Inference: Artifact.* https://doi.org/10.5281/zenodo.7774964

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings* (Turin, Italy). Springer-Verlag, Berlin, Heidelberg, 672–687. https://doi.org/10.1007/978-3-319-96983-1_48

Xin Yu, Thiago Serra, Srikumar Ramalingam, and Shandian Zhe. 2022. The Combinatorial Brain Surgeon: Pruning Weights That Cancel One Another in Neural Networks. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 25668–25683. https://proceedings.mlr.press/v162/yu22f.html

Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054

Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 213–232.

Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. https://doi.org/10.48550/ARXIV.1710.01878

Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 359–371. https://doi.org/10.1145/3352460.3358269