Sparse GPU Kernels for Deep Learning

Trevor Gale
Stanford University
United States of America
tgale@cs.stanford.edu

Matei Zaharia
Stanford University
United States of America
matei@cs.stanford.edu

Cliff Young
Google Brain
United States of America
cliffy@google.com

Erich Elsen
DeepMind
United Kingdom
eriche@google.com

Abstract—Scientific workloads have traditionally exploited high levels of sparsity to accelerate computation and reduce memory requirements. While deep neural networks can be made sparse, achieving practical speedups on GPUs is difficult because these applications have relatively moderate levels of sparsity that are not sufficient for existing sparse kernels to outperform their dense counterparts. In this work, we study sparse matrices from deep learning applications and identify favorable properties that can be exploited to accelerate computation. Based on these insights, we develop high-performance GPU kernels for two sparse matrix operations widely applicable in neural networks: sparse matrix–dense matrix multiplication and sampled dense–dense matrix multiplication. Our kernels reach 27% of single-precision peak on Nvidia V100 GPUs. Using our kernels, we demonstrate sparse Transformer and MobileNet models that achieve 1.2–2.1× speedups and up to 12.8× memory savings without sacrificing accuracy.

Index Terms—Neural networks, sparse matrices, graphics processing units

I. INTRODUCTION

Deep neural network architectures are composed of large, dense matrices used in matrix multiplication and convolutions [1], [2]. These matrices can be made sparse with little to no loss in model quality, leading to models that are more efficient in terms of both the floating-point operations (FLOPs) and parameters required to achieve a given accuracy [3]–[6].

The most common use of sparsity in deep neural networks is to accelerate inference. In addition to the standard training procedure, a sparsification algorithm is applied to produce a neural network where a high fraction of the weights are zero-valued [3], [7]–[9]. The weight matrices can then be stored in a compressed format, and sparse linear algebra kernels can be used to accelerate computation. In the context of generative models, sparsity has been applied to reduce the computational requirements of self-attention in Transformer architectures [6], [10], [11]. In addition to these applications, sparsity can be exploited to achieve higher predictive accuracy by training a larger, sparse model for a fixed computational cost [12]–[14]. To make training large sparse models feasible, all computation during training needs to operate directly on the compressed sparse representation of the model's weights.

The potential applications of sparsity in deep learning are numerous. However, it is difficult to realize the benefits of sparsity in real applications due to the lack of efficient kernels for core sparse matrix computations like sparse matrix–matrix multiplication (SpMM) and sampled dense–dense matrix multiplication (SDDMM) on accelerators like GPUs.
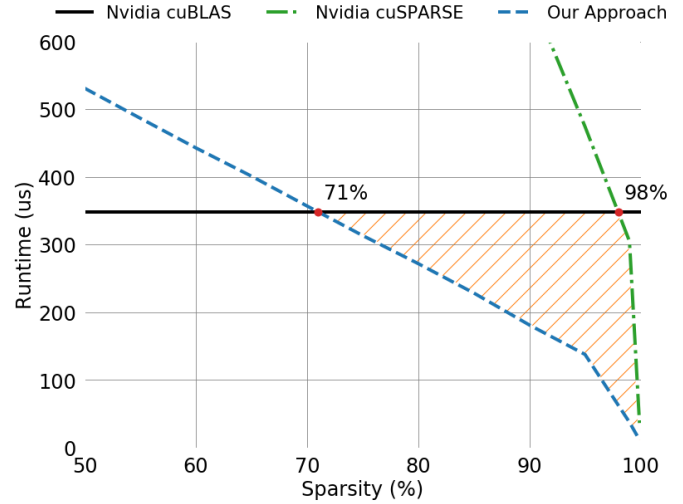

Fig. 1. **Sparse matrix–matrix multiplication runtime for a weight-sparse long short-term memory network problem**. Input size 8192, hidden size 2048, and batch size 128 in single-precision on an Nvidia V100 GPU with CUDA 10.1. Using our approach, sparse computation exceeds the performance of dense at as low as 71% sparsity. Existing vendor libraries require 14× fewer non-zeros to achieve the same performance. This work enables speedups for all problems in the highlighted region.

On parallel architectures, the performance of sparse linear algebra kernels can vary drastically with properties of the sparse matrix such as the topology of nonzero values and level of sparsity. Existing GPU kernels for sparse linear algebra are primarily optimized for scientific applications, where matrices are extremely (99%+) sparse. With the relatively moderate levels of sparsity found in deep neural networks, these kernels are not able to outperform their dense counterparts.

To address this issue, structure can be enforced on the topology of nonzeros such that nonzero values are grouped into blocks [12]–[14]. While this approach is able to recover much of the performance achieved by dense computation, the constraint on the location of nonzeros can significantly degrade model quality relative to unstructured sparsity [14]–[16].

In this work, we develop an approach for computing SpMM and SDDMM on GPUs which is targeted specifically at deep learning applications. Our approach operates directly on the standard compressed sparse row (CSR) format and does not enforce any structure on the topology of nonzero values. **We make the following specific contributions:**

- We conduct a large-scale study of sparse matrices found in deep learning and identify favorable properties that can
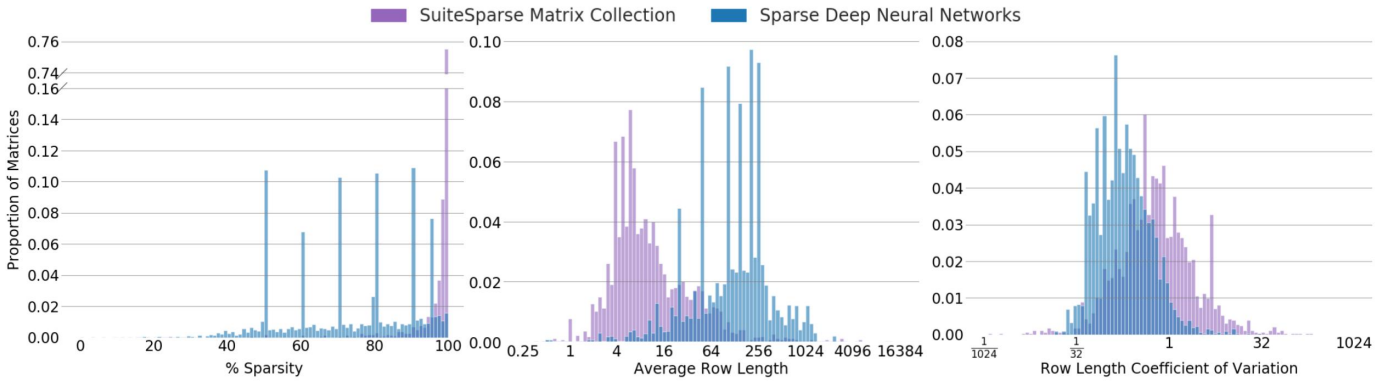
Fig. 2. **Properties of sparse matrices from scientific computing and deep learning applications.** Histograms are partially transparent to show overlapping regions. On average, deep learning matrices are 13.4× less sparse, have 2.3× longer rows, and have 25× less variation in row length within a matrix.

be exploited to accelerate sparse computation.

- We introduce a *1-dimensional tiling* scheme for decomposing the computation across processing elements that facilitates reuse of operands and lends itself to an extensible implementation.
- We develop two techniques, *subwarp tiling* and *reverse-offset memory alignment*, that enable the use of vector memory instructions on misaligned memory addresses in sparse data structures.
- We introduce *row swizzle load balancing*, an approach for load balancing computation between processing elements that is decoupled from the parallelization scheme.

On a large dataset of sparse matrices taken from state-of-the-art deep neural networks, we demonstrate geometric mean speedups of 3.58× and 2.19× over Nvidia cuSPARSE for SpMM and SDDMM respectively on Nvidia V100 GPUs. On the top performing problems, our kernels reach 27% of single-precision peak. Using our kernels, we demonstrate sparse Transformer and MobileNet models that achieve 1.2–2.1× end-to-end speedups and 12.8× reductions in memory usage while matching the accuracy of their dense counterparts. Our code is open-source and available at https://github.com/google-research/sputnik

II. SPARSE MATRICES IN DEEP LEARNING

To understand the properties of sparse matrices in deep learning, we constructed a dataset of sparse deep neural network weight matrices from the large-scale study of [17]. The dataset is composed of ResNet-50 [1] and Transformer [2] models trained on ImageNet [18] and WMT14 English-to-German [19] respectively, and includes models trained with four different algorithms for inducing sparsity in neural networks. For Transformer, we limit our analysis to models that achieve above 20 BLEU on the WMT14 English-German test set. For ResNet-50, we include models that achieve over 70% top-1 accuracy on the ImageNet validation set. In total, the collection includes 3,012 matrices from 49 different models.

Our analysis focuses on three properties of the matrices: row length (in number of nonzeros) coefficient of variation (CoV), average row length, and sparsity. The CoV of a matrix's row lengths is the standard deviation of the row lengths divided by their mean. A high CoV is indicative of load imbalance

across the rows of a sparse matrix. The average row length captures the average amount of work that will be done on each row of the sparse matrix. Longer row lengths are desirable as startup overhead and one-time costs can be amortized over more useful work. Sparsity measures the fraction of values that are zero valued in a matrix. Depending on the implementation, lower sparsity levels can be useful to increase the likelihood that nonzero values in different rows fall into the same columns, opening up opportunities for the reuse of operands through caches.

We contrast the properties of deep learning workloads with matrices from the SuiteSparse Matrix Collection [20], which is made up of 2,833 sparse matrices from a wide range of scientific workloads including circuit simulations, computational fluid dynamics, quantum chemistry, and more.

A. Results & Analysis

Statistics for our corpus of deep learning matrices and the SuiteSparse Matrix Collection are plotted in Figure 2. The difference between sparse matrices from scientific workloads and those from deep learning is considerable: on average, deep learning matrices are 13.4× less sparse, have 2.3× longer rows, and have 25× less variation in row length within a matrix. We find it likely that these differences are primarily driven by the desire to maintain high accuracy, which requires deep neural networks with a large number of parameters. This in turn leads to a higher number of nonzeros per row and a lower CoV, which is inversely proportional to average row length. For each of the metrics that we studied, deep learning matrices exhibit favorable properties that we can take advantage of to accelerate sparse matrix computations.

III. GRAPHICS PROCESSING UNITS BACKGROUND

This section provides a basic description GPU architecture and terminology. Our implementation is written in CUDA and thus we opt for the terminology used by Nvidia.

GPUs are made up of an array of *streaming multiprocessors* (SMs) and GPU kernels are made up of threads that are grouped into sets of 32 called *warps*. Warps are grouped into larger sets of threads called *thread blocks*. The set of thread blocks that make up a kernel is called a *grid*. When a kernel is launched to the GPU for execution, each thread
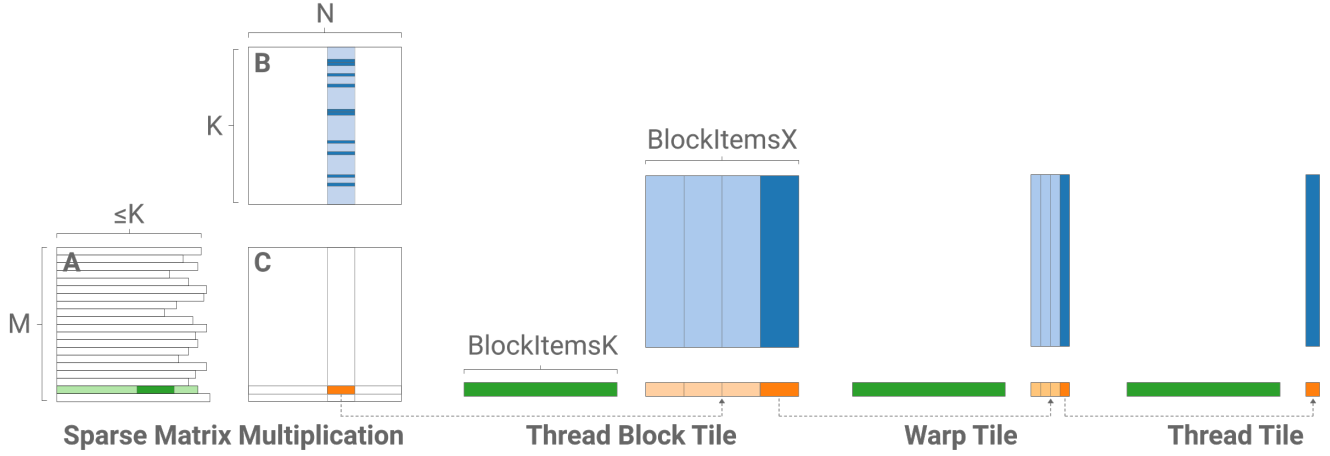
Fig. 3. **Hierarchical decomposition of SpMM with 1-dimensional tiling**. Visualized with 4 warps in a thread block and 4 threads in a warp for brevity. In each level of the decomposition, matrix A is a sparse matrix stored in compressed-sparse row format, marked in green and shown on the left. Matrix B is dense, marked in blue and shown on top of the output matrix. The output matrix C is dense, marked in orange and shown in the bottom right of each level. The dark regions at each level denote data used by the following level. **Far Left:** Each thread block computes a 1-dimensional tile of the output matrix. All values from the row of sparse matrix A are needed by all threads. We use all threads in the thread block to collaboratively load values and indices from A to shared memory where they can be quickly accessed for computation. For each column index loaded from A, the thread block will load a contiguous vector from matrix B (marked with dark blue horizontal stripes). **Center Left - Far Right:** Threads process independent outputs and thus need disjoint subsets of columns from dense matrix B. Each thread loads the values from B needed to compute it's outputs and stores them in thread-local registers.

block is scheduled onto an SM. A *wave* of thread blocks is a set of thread blocks that run concurrently on the GPU [21]. All threads within a thread block can communicate through fast, programmer-managed, shared memory that is local to the SM. All threads also have access to thread-local registers. The number of thread blocks that execute concurrently on an SM is referred to as the *occupancy* of the kernel. Higher occupancy is typically desirable, as thread-level parallelism can be exploited to hide the latency of memory and arithmetic operations. GPUs have a large but high-latency global memory that is accessible to all SMs, an L2 cache that is shared by all SMs, and L1 caches that are local to each SM. When a warp of threads access global memory, GPUs try to coalesce the accesses into as few transactions as possible.

IV. SPARSE MATRIX COMPUTATION

This section explains the operations implemented by our SpMM and SDDMM kernels.

A. Sparse Matrix–Matrix Multiplication Operation

Our SpMM kernel implements the computation $AB \Rightarrow C$, where A is sparse and stored in the standard compressed sparse row (CSR) format. In the following sections, we refer to matrices A, B and C as the sparse matrix, dense matrix, and output matrix respectively.

B. Sampled Dense–Dense Matrix Multiplication Operation

The SDDMM operation is defined as $AB \odot C \Rightarrow D$, where C and D are sparse and $\odot$ denotes the element-wise product of two matrices [22], [23]. Thanks to the element-wise scaling with a sparse matrix, dot-products for zero-valued locations of the output can be skipped to accelerate computation.

In sparse deep neural networks, SDDMM is necessary for a number of key computations. For example, in a weight sparse neural network the forward pass computes $WX \Rightarrow Y$, where W

is sparse. In the backward pass, the gradient w.r.t. the sparse weights is computed as $\delta Y X^T \odot \mathbb{I}[W] \Rightarrow \delta W$, where $\mathbb{I}[W]$ is an indicator function that returns 1 in the location of the nonzero values of the sparse matrix W. Transformer models with sparse attention similarly compute $QK^T \odot \mathbb{I}[Y] \Rightarrow Z$ in the forward pass, where Q and K are the query and key inputs to the attention mechanism respectively and Y is a sparse matrix that describes the connectivity of the attention mechanism.

These computations differ from the strict definition of SDDMM in two ways. First, they do not require the element-wise scaling by the sparse matrix values. Secondly, the B input operand to the SDDMM is typically transposed. With these applications in mind, our SDDMM implements the computation $AB^T \odot \mathbb{I}[C] \Rightarrow D$. While we specialize to the computation that arises in deep learning, we note that our approach is easily extensible to the general SDDMM computation [1].

C. Data Organization

To enable coalesced memory accesses into all input and output matrices, we store dense matrices in row-major layout and sparse matrices in CSR format [24]–[26]. We note that computing SpMM as $BA \Rightarrow C$, where A is the sparse matrix stored in compressed sparse column format and B and C are stored column-major would be equally efficient.

V. SPARSE MATRIX–MATRIX MULTIPLICATION

This section details the design of our SpMM kernel.

A. Hierarchical 1-Dimensional Tiling

Our scheme for SpMM on GPU is diagrammed in Figure 3 and presented in CUDA pseudo-code in Figure 4. The decomposition follows a row-splitting scheme [26], with one

---

[1]Element-wise scaling adds 1 load and 1 multiply instruction prior to storing the output. Non-transposed right-hand operand makes memory accesses trivially coalesced and simplifies the kernel relative to the transposed case.

key difference: rather than mapping a thread block to an entire row of the output matrix, we shard the output into 1-dimensional tiles and map independent thread blocks to each.

The motivation for this approach stems from the fact that the number of columns in the dense matrix can vary drastically in deep learning applications[2]. Consider various neural network architectures with sparse weight matrices and dense activations. When training RNNs this dimension corresponds to the batch size, which is typically between 16-128 elements [27]. In Transformer architectures, this dimension is the product of the batch size and sequence length, which can vary from 256 to over 2048 elements [2], [28]. In 1×1 convolutions, this dimension is the product of the image height and width with the batch size. In EfficientNet architectures, the product of the spatial dimensions alone range from 64 to 14,400 [29].

There are three main benefits to 1-dimensional tiling. Firstly, we can easily templatize our implementation for different tile sizes and generate specialized kernel variants for different regions of the problem space. Secondly, for problems with small M and K dimensions we launch more thread blocks than would otherwise be possible, enabling us to achieve higher occupancy and a higher fraction of peak throughput. Lastly, processing fixed-sized blocks enables us to aggressively unroll loops and compute offsets and constants at compile time. We similarly iterate through the reduction dimension in fixed-size steps, enabling further loop unrolling and static evaluation.

B. Vector Memory Operations

Vector memory instructions are an important tool for mitigating bandwidth bottlenecks and decreasing the number of instructions needed to express a computation [30]. However, it is non-trivial to use these operations in sparse matrix kernels.

First, using vector memory instructions increases the number of values loaded simultaneously by a thread block. For example, a thread block with a single warp using 4-wide vector loads would request 128 floats with a single instruction. In our 1D tiling scheme, this means that some loads from a sparse matrix row of length less than 128 would need to be predicated off. Similarly, problems with fewer than 128 columns in the dense matrix would execute with some threads in every thread block predicated off for the entirety of the kernel's execution. These constraints limit the utility of vector memory accesses, applied naively, to very large problems.

Secondly, vector memory accesses require that the target values be aligned to the vector width (2 or 4 32-byte values). For accesses into the dense matrix or output matrix this requires that the number of columns be divisible by the vector width such that the start of every row of values is properly aligned. The larger issue is with loads from the sparse matrix. With a 1-dimensional tiling or row-splitting scheme, accesses within a thread block begin at the start of a row of values in the sparse matrix. Because rows in a sparse matrix can have arbitrary lengths, these initial addresses have no alignment guarantees regardless of the problem dimensions.

---

[2]Existing work on SpMM often focuses on problems where the dense matrix is "tall and skinny" [24]–[26].

```
1  template <int kBlockItemsK, int kBlockItemsX>
2  __global__ void SpmmKernel(
3    SparseMatrix a, Matrix b, Matrix c) {
4    // Calculate tile indices.
5    int m_idx = blockIdx.y;
6    int n_idx = blockIdx.x * kBlockItemsX;
7
8    // Calculate the row offset and the number
9    // of nonzeros in this thread block's row.
10   int m_off = a.row_offsets[m_idx];
11   int nnz = a.row_offsets[m_idx+1] - m_off;
12
13   // Main loop.
14   Tile1D c_tile(/*init_to=*/0);
15   for(; nnz > 0; nnz -= kBlockItemsK) {
16     Tile1D a_tile = LoadTile(a);
17     Tile2D b_tile = LoadTile(b);
18     c_tile += a_tile * b_tile;
19   }
20
21   // Write output.
22   StoreTile(c_tile, c);
23 }
```

Fig. 4. **CUDA pseudo-code for SpMM with 1-dimensional tiling.** The output matrix is statically partitioned into 1-dimensional tiles. Independent thread blocks are launched to compute each output tile. On each iteration of the main loop, we load a 1-dimensional strip of the sparse matrix and a 2-dimensional tile of the dense matrix and accumulate the vector-matrix product. After processing all nonzero values in the row, the results are written to the output matrix.

1) Subwarp Tiling: To address the first issue, we extend our scheme to allow mapping of subsets of a warp (i.e., a subwarp) to independent 1D tiles of the output. This reduces the access width constraint by a factor of the number of subwarps used. This also gives us the flexibility to spread threads across more rows of the output matrix for problems with a smaller number of columns in the dense and output matrices.

With subwarp tiling, our scheme bears some resemblance to a standard two-dimensional tiling scheme at the warp level. The important difference is that subwarps processing different rows of the output matrix are not able to reuse values loaded from the dense matrix. However, depending on the sparsity level, accesses issued by different subwarps are likely to exhibit locality that could be serviced through caches.

The main drawback to this approach is that rows of variable length can result in warp divergence. We address the issue of load imbalance between threads in a warp in section V-C.

2) Reverse Offset Memory Alignment: A simple approach to address the second issue is to pad the rows of the sparse matrix with zeros such that all rows are a multiple of four in length. However, this limits the generality of the kernel. To enable the use of vector memory instructions on arbitrary sparse matrices, we introduce a simple trick in the setup portion of the kernel (AKA, the prelude): after loading the row offset and calculating the row length, each thread block decrements its row offset to the nearest vector-width-aligned address and updates the number of nonzeros that it needs to process. To maintain correctness, the threads mask any values that were loaded from the previous row prior to accumulating the result in the first iteration of the main loop.

We refer to this trick as *reverse offset memory alignment* (ROMA). Relative to the explicit padding scheme, ROMA

does not change the amount of work done by each thread block. The key difference is that instead of explicitly padding the matrix data structure, ROMA effectively pads the rows of the sparse matrix with values from the row before it[3].

ROMA can be implemented very efficiently. The alignment process adds 6 PTX instructions in the kernel prelude: 2 bitwise `and`, 1 `add`, 1 `setp` (set predicate), and 2 `selp` (select based on predicate). The masking process adds 1 `setp` and 2 `str.shared` (shared memory store) instructions to the first iteration of the main loop.

These techniques for enabling the use of vector memory instructions on sparse matrices are visualized in Figure 5. Figure 8 shows CUDA pseudo-code for our SpMM kernel with the necessary modifications for subwarp tiling and ROMA.

C. Row Swizzle Load Balancing

A number of approaches for handling load imbalance in sparse computation have been proposed [23], [26], [32]. However, existing approaches tightly couple load balancing to the parallelization scheme. While these schemes achieve good performance when load imbalance is significant, they typically introduce computational irregularity that can damage performance on more regular problems problems [26]. However, despite the regularity of sparse matrices found in DNNs, our kernels still suffer from load imbalance (see Figure 7).

When mapping sparse matrix operations to GPUs, there are two potential sources of load imbalance [26].

(a) **Load imbalance between warps or thread blocks:** Some warps/thread blocks may be assigned less work than others. This can lead to some SMs sitting idle while others do useful work.

(b) **Load imbalance within a warp or thread block:** Some threads within a warp may be assigned less work than others. This can lead to warp divergence and inefficient use of math units and memory bandwidth within an SM.

To address these issues, we make two observations. First, many units of work of varying sizes will be scheduled onto each SM over the course of kernel execution. Secondly, we can control what work is assigned to which SM by changing which threads are assigned to process each unit of work. Based on these observations, we can ensure the workload is balanced across the processing elements by remapping where work is scheduled such that each processing element is assigned a roughly equal amount of work. We refer to this remapping procedure as a *row swizzle*[4]. To address both sources of load imbalance, we introduce a two level re-ordering of work:

(a) **Row Binning:** Given an understanding of how thread blocks are mapped to SMs, alter the tile mappings such that each SM receives approximately the same amount of

---

[3]Note that the first row of the sparse matrix is guaranteed to be vector aligned, as all CUDA memory allocation routines allocate memory with at least 256-byte alignment [31]

[4]In reference to the similar approach of thread block swizzles, where thread blocks are reordered to improve cache reuse [33] as well as the fact that the reordering in our case is at the granularity of an output row.
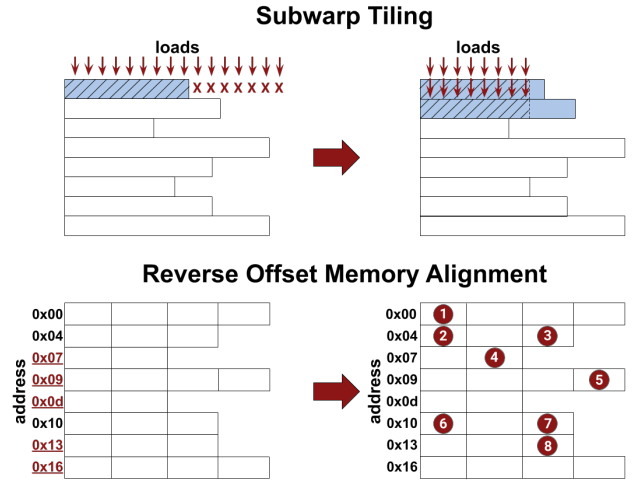

Subwarp Tiling

Reverse Offset Memory Alignment

Fig. 5. **Techniques for enabling the use of vector memory operations on sparse matrix data structures**. **Top:** Subwarp tiling maps subsets of a warp to independent 1-dimensional tiles of the output. By splitting warp accesses across multiple rows we reduce the amount of wasted work. Rows assigned to a warp/subwarp are marked in blue. Load addresses are indicated with arrows and predicated loads are indicated with "X". The hashed region denotes the values loaded by a single set of loads across the warp/subwarp. **Bottom:** Reverse offset memory alignment backs up the address of each row to the nearest aligned address. Values from the previous row are masked in the first loop iteration to maintain correctness. Misaligned addresses are underlined. Modified row start addresses are marked with circles and row index.

work to do. This helps to address load imbalance between warps/thread blocks.

(b) **Row Bundling:** For kernels where warps are split across multiple rows of the sparse matrix, alter the tile mappings such that each subwarp receives approximately the same amount of work to do. This helps address load imbalance within a warp.

1) Volta Thread Block Scheduler: The binning of rows such that SMs receive roughly the same amount of work is complex to implement, as it depends on the GPUs thread block scheduling algorithm, which is not public knowledge. We reverse engineer the Nvidia Volta thread block scheduler, following the same general approach as [34].

Overall, the Volta thread block scheduler is much simpler than the Fermi thread block scheduler. Thread blocks in the first wave are assigned to SMs based on their block index:

$$sm\_idx = 2(block\_idx \bmod 40) + \frac{block\_idx}{40} \bmod 2$$

where $block\_idx$ is calculated:

$$block\_idx = blockIdx.x + blockIdx.y * gridDim.x$$

This mapping distributes thread blocks round-robin over the SMs. After the first wave, thread blocks are scheduled in order of $block\_idx$ as resources become available.

2) Row Binning & Row Bundling Heuristics: A simple heuristic for binning rows is to select the first wave to be the heaviest N row bundles and then pair the following N heaviest row bundles with the previous bundles in reverse order of heaviness. To bundle the rows by size, we can greedily create bundles from consecutive rows ordered by size.

Given the online thread block scheduling algorithm used by Nvidia GPUs, these two heuristics can be implemented with a sort of the row indices by row length. Given a sorted array of the row indices in order of decreasing size, bundles consist of blocks of consecutive row indices. The first wave of bundles are scheduled round-robin across the SMs, and remaining bundles are scheduled in decreasing order of heaviness as bundles complete execution. We note that this heuristic for row binning is similar to guided self-scheduling [35].

An advantage of this approach is that we do not need to know the target bundle size to group similarly sized rows. This means that the heuristic does not need to have insight into any kernel selection heuristics used under the hood.

Since the topology of sparse matrices in DNNs is typically updated infrequently, the cost of the `argsort` of the row indices by their row lengths can be amortized over many training steps [6], [13], [17]. Implementing the swizzle in the kernel also requires the addition of a single load during the kernel prelude. The memory required to store the sorted indices for the matrix is negligible, as the number of rows in the matrix is typically much smaller than the number of nonzeros in the matrix for our target applications.

Figure 6 shows the high-level scheme for row swizzle load balancing. Figure 7 shows the performance of row swizzle load balancing for a sample problem as load imbalance increases. Figure 8 shows CUDA pseudo-code for our SpMM kernel with the necessary modifications for row swizzle load balancing.

D. Implementation Details

This section details additional low-level optimizations we applied to achieve good performance.

1) Index Pre-Scaling: In each iteration of the main loop of our kernel, we load the sparse matrix values and indices and store them in shared memory. Each index will be used by all threads to load from the dense matrix. To avoid redundant work each time an index is loaded, we have each thread scale its portion of the indices prior to storing to shared memory.

2) Residue Handling: Our kernel processes as many full tiles of nonzero values as possible and then executes a residue handler to accumulate the remaining products. As sparse matrix row lengths are rarely divisible by the tile size, it's important that the residue handling code is highly efficient.

To maximize shared memory bandwidth and minimize bank conflicts, we use 128-bit shared memory load instructions whenever possible [36]. This is trivial for the main loop, but difficult for the residue handling code as the number of nonzeros remaining is not necessarily divisible by four. To enable the use of wide shared memory instructions, we zero the shared memory buffers used for sparse matrix values and indices prior to loading the residual values and indices. We then split the loops for dense matrix loading and computation into two, and unroll the inner loop $4\times$ without bounds checks.

3) Mixed Precision: In addition to standard 32-bit floating-point kernels, we extended our SpMM implementation to support mixed-precision computation, as is commonly used in deep learning [37]. Our kernels support 16-bit floating-point
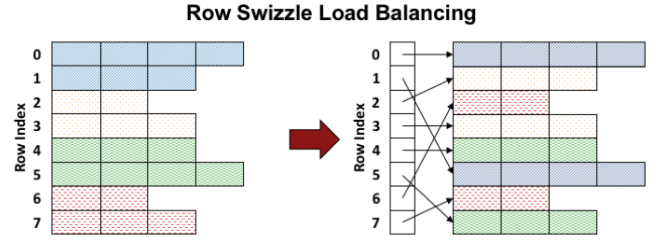

Fig. 6. **Row swizzle approach for load balancing sparse matrix computation**. Rows processed by the same warp are marked with the same pattern and color. We introduce a layer of indirection that re-orders when rows are processed. To balance work between threads in a warp, we group rows of similar length into *bundles*. To balance work between SMs, we process row bundles in decreasing order of size.
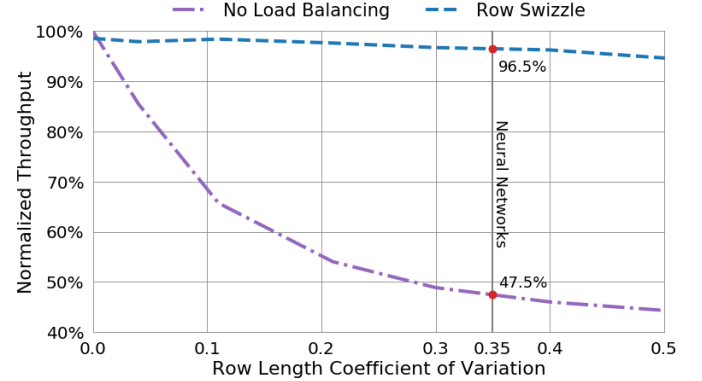

Fig. 7. **Sparse matrix–matrix multiplication runtime with varying levels of load imbalance**. M=8192, K=2048, N=128, sparsity=75% in single-precision on an Nvidia V100 GPU. Throughput measured as a percent of the throughput achieved with a sparse matrix where all rows have the same number of nonzero values. The gray line labeled "Neural Networks" marks the average CoV of sparse matrices in our dataset of deep neural networks. With this problem configuration, performance of the standard row ordering degrades to 47.5% of throughput with a perfectly balanced sparse matrix. Our row swizzle load balancing technique maintains 96.5% of the throughput with a perfectly balanced sparse matrix. We document the performance of our row swizzle load balancing technique further in Section VII-B

input/output data and use 16-bit integer indices for the sparse matrix meta-data. Inside our kernel, we convert FP16 data to FP32 and issue FP32 fused multiply-add instructions, as is standard. We convert the final outputs from FP32 to FP16 before writing the result. Due to the reduced representational capacity of 16-bit integers, we do not perform our index pre-scaling optimization for mixed-precision kernels.

VI. SAMPLED DENSE–DENSE MATRIX MULTIPLICATION

This section details the design of our SDDMM kernel.

A. Hierarchical 1-Dimensional Tiling

We use the same 1D tiling scheme for SDDMM as we do for SpMM, with two main differences. First, instead of mapping thread blocks to 1D regions of the output we map them to 1D strips of consecutive nonzeros. Because the output is sparse, this ensures better work distribution across thread blocks and is simpler to implement. Because the number of nonzeros in each row cannot be inferred without inspecting the sparse matrix, we launch the maximum number of thread blocks that could be needed. On startup, each thread block calculates if it has work to do and returns early if it is not needed. An alternative

approach would be to use dynamic parallelism [38]. However, we do not observe significant overhead from launching extra thread blocks in our benchmarks. For SDDMM targeting problems with very high sparsity, it's possible that dynamic parallelism would lead to better performance.

The second difference in our work decomposition is caused by the need to perform the computation with the transpose of the right-hand operand. With the dense matrices stored in row-major layout, naively partitioning the outputs across the threads would result in strided, uncoalesced memory accesses to the right-hand matrix. To avoid this issue, we alter our scheme so that each thread mapped to an output tile computes a portion of the results for all outputs in that tile. We then perform a reduction between these threads using warp shuffle instructions to compute the final results for each thread.

An alternative to this approach would be to perform the transpose of values loaded from the right-hand matrix in shared memory prior to computation. While this would use less registers per-thread, it would double shared memory usage. On Nvidia Volta GPUs shared memory and L1 cache use the same storage. Thus, using more shared memory reduces the size of the L1 cache. For these kernels, we found L1 cache capacity to be important for performance and thus decided against performing an explicit shared memory transpose.

B. Vector Memory Operations

Because both inputs are dense, it is trivial to use vector loads/stores for SDDMM problems where the inner dimension is divisible by the vector width. For all problems, we use scalar loads/stores on the sparse matrix. These operations only occur at the beginning and end of the kernel and do not significantly affect performance. To enable the use of vector loads/stores on a wider range of problems we process output tiles with subwarps, as explained in the context of SpMM.

C. Implementation Details

While we do use subwarp tiling to enable the use of vector memory instructions on a wider range of problems, load balancing in SDDMM is less critical due to the fact that all dot-products to be computed are of equal length. Additionally, problems from deep neural networks commonly have a dot-product length that is divisible by the SIMT width, making efficient residue handling less critical than in SpMM. For the SDDMM residual computation we use the same loop structure as the main loop and do not apply our loop-splitting optimization to enable wide shared memory loads.

VII. EXPERIMENTS

This section provides empirical results and analysis of our SpMM and SDDMM kernels. For SpMM we use a kernel selection heuristic where we select the n-dimension tile size to be N, rounded up to a power of 2, up to a maximum of 64. For SDDMM we use an n-dimension tile size of 32. For both kernels we use the widest vector memory operations possible. All benchmarks were conducted with CUDA 10.1.

```
1  template <int kBlockItemsY, int kBlockItemsK,
2      int kBlockItemsX>
3  __global__ void SpmmKernel(
4      SparseMatrix a, Matrix b, Matrix c) {
5      // Subwarp tiling: calculate tile indices
6      // for this subwarp.
7      int m_idx = blockIdx.y * kBlockItemsY +
8                  threadIdx.y;
9      int n_idx = blockIdx.x * kBlockItemsX;
10
11     // Row swizzle: load this subwarp's row
12     // index from the pre-ordered indices.
13     m_idx = a.row_indices[m_idx];
14
15     // Calculate the row offset and the number
16     // of nonzeros in this thread block's row.
17     int m_off = a.row_offsets[m_idx];
18     int nnz = a.row_offsets[m_idx+1] - m_off;
19
20     // ROMA: align the row pointer so that we
21     // can use vector memory instructions.
22     MemoryAligner aligner(m_off, nnz);
23     nnz = aligner.AlignedNonzeros();
24     m_off = aligner.AlignedRowOffset();
25
26     // First loop iteration.
27     Tile1D c_tile(/*init_to=*/0);
28     if (nnz > 0) {
29         Tile1D a_tile = LoadTile(a);
30         aligner.MaskPrefix(a_tile);
31         Tile2D b_tile = LoadTile(b);
32         c_tile += a_tile * b_tile;
33         nnz -= kBlockItemsK;
34     }
35
36     // Main loop.
37     for(; nnz > 0; nnz -= kBlockItemsK) {
38         Tile1D a_tile = LoadTile(a);
39         Tile2D b_tile = LoadTile(b);
40         c_tile += a_tile * b_tile;
41     }
42
43     // Write output.
44     StoreTile(c_tile, c);
45  }
```

Fig. 8. **CUDA pseudo-code for SpMM with subwarp tiling, ROMA and row swizzle load balancing.** To enable the use of vector memory instructions on a wider range of problem configurations, subsets of a warp are mapped to 1-dimensional output tiles, adding a template parameter that denotes the number of subwarps used (lines 1-2) and altering the index calculations (lines 5-8). To decrease load imbalance, we alter the order in which rows are processed (lines 11-13). To enable the use of vector memory instructions on misaligned addresses in sparse matrices, we back-up the row pointer to the nearest aligned address (lines 20-24). To maintain correctness, we mask any values loaded from the previous row in the first iteration of the main loop (line 30).

A. Kernel Benchmarks

1) Sparse Matrix Dataset: We evaluate the performance of our SpMM and SDDMM kernels by benchmarking on our dataset of sparse matrices from deep neural networks. For each of the 3,012 matrices in the dataset, we benchmark with both training and inference batch sizes. For SDDMM, we benchmark the problem corresponding to the gradient with respect to the sparse weight matrix. We benchmark convolution operations found in ResNet-50, as an `im2col` transform on the input data followed by SpMM or SDDMM [39]. We do not include the time of the `im2col` transform in our benchmarks. For ResNet-50 benchmarks with inference batch size, we pad the batch dimension to the nearest multiple of four
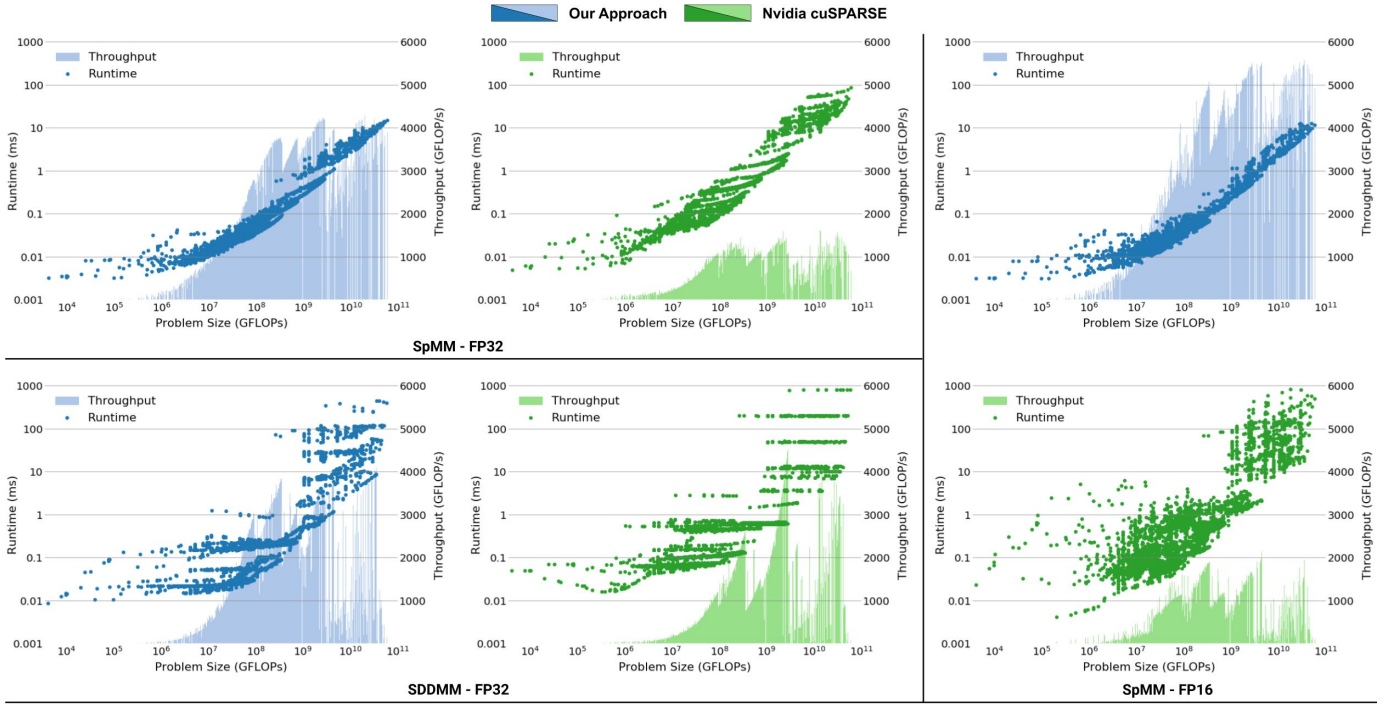
Fig. 9. **Benchmarks on our dataset of sparse matrices from deep neural networks.** Runtime (left y-axis) and throughput (right y-axis) plotted with increasing problem size for each kernel and precision. Benchmarked on an Nvidia V100 GPU. **Top Left:** SpMM benchmarks in single-precision. Across all problems, our approach achieves a geometric mean speedup of 3.58× and a peak speedup of 14.2× over Nvidia cuSPARSE. **Bottom Left:** SDDMM benchmarks in single-precision. Across all problems, our approach achieves a geometric mean speedup of 2.19× and a peak speedup of 6.58× over Nvidia cuSPARSE. **Right:** SpMM benchmarks in mixed precision with 16-bit data and 32-bit computation. Across all problems, our approach achieves a geometric mean speedup of 5.97× and a peak speedup of 297.5× over Nvidia cuSPARSE.

to enable vector memory instructions. All benchmarks are performed on an Nvidia V100 GPU. We use Nvidia cuSPARSE's `cusparseSpMM` and `cusparseConstrainedGeMM` as the baselines for our SpMM and SDDMM benchmarks respectively. Both cuSPARSE kernels use column-major layouts for dense matrices and CSR format for sparse matrices. Because `cusparseConstrainedGeMM` does not support transposition of the right-hand operand, we explicitly transpose the matrix using cuBLAS and include the transposition in our timing. We benchmark all problems on a single Nvidia V100 GPU. The results of all benchmarks are presented in Figure 9. We present statistics for these benchmarks in Table I.

Across all benchmarks, our SpMM and SDDMM kernels show significant advantages over Nvidia cuSPARSE. For single-precision SpMM, our kernel achieves a geometric mean speedup of 3.58× and reaches 4.29TFLOPs, representing 27.3% of single-precision peak. Our kernel outperforms cuSPARSE on 99.75% of the problems in our dataset. For single-precision SDDMM, our kernel achieves a geometric mean speedup of 2.19× and reaches 4.11TFLOPs, representing 26.2% of single-precision peak. Our kernel outperforms cuSPARSE on 93.34% of the problems in our dataset.

With mixed-precision, our SpMM kernel achieves a geometric mean speedup of 5.97× over Nvidia cuSPARSE and a peak throughput of 5.57TFLOPs. While our kernel uses 16-bit integers for the sparse matrix meta-data, cuSPARSE only supports 32-bit indices. We find it likely that this contributes to the increased performance gap. Our mixed-precision SpMM

outperforms cuSPARSE on 99.7% of the problems in our dataset. We note that cuSPARSE's mixed-precision SpMM performs inconsistently on some problems, leading to extreme slowdowns of as much as 297.5× relative to our kernel.

2) Sparse Recurrent Neural Networks: This section evaluates the performance of our kernels relative to the recently proposed techniques of [26] and [23]. The SpMM kernel provided by [26] only supports problems with batch sizes divisible by 32. [23] wrote SpMM and SDDMM kernels for batch size 32 and 128 and also require that the number of rows in the sparse matrix be divisible by 256. Given these constraints, we opt to benchmark these kernels on a dataset of problems from recurrent neural networks, where the problem configurations supported by the kernels from [26] and [23] are realistic for deep neural networks. We benchmark each kernel on RNN, gated recurrent unit (GRU) [40], and long short-term memory network (LSTM) [41] problems with sparse weights. We generated sparse matrices with random uniform sparsity. We benchmarked problems with state sizes 1k, 2k, 4k, and 8k, sparsities 70%, 80%, and 90% and batch sizes 32 and 128. All benchmarks were performed on an Nvidia V100 in single-precision. We do not include the time require for the pre-processing step used by the Adaptive Sparse Tiling (ASpT) approach of [23] in our benchmarks. We benchmark the row-splitting kernel from [26], as all of our benchmarks are beyond the threshold of average row length that the authors use to select between their row-splitting and nonzero-splitting kernels. Benchmark results are presented in Figure 10.
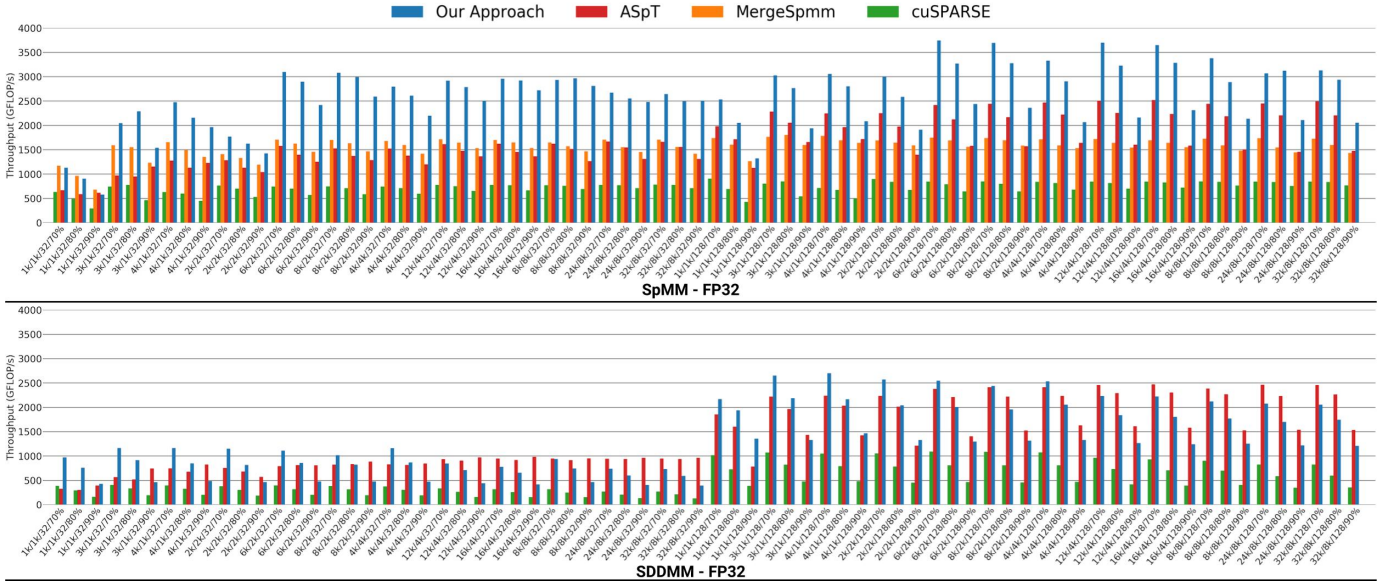
Fig. 10. **Benchmarks on sparse recurrent neural neural network problems.** Each problem is labeled M/K/N/sparsity. All benchmarks taken on an Nvidia V100 GPU in single-precision. **Top:** SpMM benchmarks. Compared to ASpt [23], our kernel achieves a geometric mean speedup of $1.56\times$ and a peak speedup of $2.4\times$. Compared to the merged-based approach of [26], our kernel achieves a geometric mean speedup of $1.59\times$ and a peak speedup of $2.15\times$. Compared to cuSPARSE, our kernel achieves a geometric mean speedup of $3.47\times$ and a peak speedup of $4.45\times$. **Bottom:** SDDMM benchmarks. Our kernel performs competitively with the adaptive sparse tiling approach of [23], achieving 92% of the throughput on average while using $3\times$ less memory and no-reordering of the sparse matrix. Compared to cuSPARSE, our kernel achieves a geometric mean speedup of $2.69\times$ and a peak speedup of $3.51\times$.

For SpMM, our approach significantly outperforms other methods. Our approach achieves geometric mean speedups of $1.56\times$, $1.59\times$, and $3.47\times$ over MergeSpmm [26], ASpT, and cuSPARSE respectively. For SDDMM, our approach significantly outperforms cuSPARSE and achieves performance on-par with ASpT. Our approach achieves geometric mean speedups of $2.69\times$ over cuSPARSE and 92% of the throughput of ASpT on average. While ASpT achieves good performance for SDDMM, it has a number of limitations. First, including the original CSR matrix, ASpT requires $3\times$ the memory to store the re-ordered matrix as well as meta-data needed for tiled execution. Second, the author's implementation uses different re-orderings of the sparse matrix for SpMM and SDDMM problems. For deep learning applications, this means that gradients calculated with respect to a sparse matrix will be in a different order than the sparse matrix used in the forward pass. In order to perform gradient updates or continue backpropagation, applications must pay the cost of re-ordering the sparse matrix on every training iteration.

B. Ablation Study

Table II shows the results of our ablation study on the optimizations we propose for each kernel. We benchmark both kernels on our dataset of sparse matrices from DNNs with both training and inference batch sizes. We report statistics for each model and batch size separately to show the effect of each technique on different portions of the problem space.

Across these benchmarks we find that techniques like row swizzle load balancing and residue unrolling are robust to varying problem configurations, while vector memory instructions show large benefits for compute heavy problems and less benefit for small problems. One outlier is the superior perfor-

mance of scalar memory operations for SDDMM. With the small weight matrices found in these models, these problems are largely occupancy-bound and thus benefit from the fact that our scalar kernels process fewer outputs per thread. On the dataset of RNN problems studied in Section 10, we observe our vector kernel achieve a geometric mean speedup of $2.45\times$ over the scalar variants. These results indicate that better kernel selection heuristics could greatly improve performance.

In addition to these techniques, our kernels benefit from the use of favorable data layouts and an efficient implementation enabled by our 1D tiling scheme.

C. Application: Sparse Transformer

Transformer models are a popular sequence modeling architecture, having been used to achieve state-of-the-art results on tasks such as machine translation [2], language modeling [42], and image generation [28]. Transformer models are made up of stacked layers, each of which contains a multi-head attention mechanism followed by a small fully-connected network. The attention mechanism used in Transformer takes in a query Q, key K, and value V and computes a weighted average of the input values based on the similarity of Q and K:

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Where d_k is the number of features for each element of the sequence. Despite the effectiveness of this architecture, QK^T computes the similarity of each token in the sequence with all other tokens, requiring computation and memory that grows quadratically with the sequence length. To alleviate this issue, recent work has explored the use of sparsity in the attention mechanism [6], [10], [11]. With sparse attention, we compute a subset of the outputs of QK^T and then multiply

Kernel	Single-Precision		Mixed-Precision
	SpMM	SDDMM	SpMM
Geo. Mean Speedup	3.58×	2.19×	5.97×
Peak speedup	14.2×	6.58×	297.5×
Peak throughput (TFLOPs)	4.29	4.11	5.57

the sparse output by V. <mark>With unstructured sparsity, these operations correspond to an SDDMM followed by an SpMM.</mark>

1) Experimental Setup: We trained a Transformer with sparse attention on the ImageNet-64x64 image generation dataset which has a sequence length of 12,288. Our model consists of 3 layers with 8 attention heads each, a hidden dimension of 1,024 and a filter size of 4,096 in the fully-connected network. We trained our models with a batch size of 8 for 140,000 training steps. For our sparse model, we simulate sparsity during training and convert to a sparse representation for benchmarking. While we train on an image generation task, we note that this architecture can be applied to other sequence learning tasks like language modeling without modification.

For our sparse model, we generated attention masks with a dense band of size 256 along the diagonal and random sparsity off-diagonal sampled with probability inversely proportional to the distance from the diagonal. We set off-diagonal sparsity to 95%. The sparse attention mask stays the same over the course of training and is shared by all attention heads and layers. The attention mask used by our model is visualized in Figure 11. We additionally wrote a kernel that computes the softmax function on a sparse matrix. For each model, we benchmark the forward pass in single-precision.

2) Results & Analysis: Benchmark results are reported in Table III. On a V100 GPU, our sparse model achieves a 2.09× speedup and 12.8× memory savings over the standard Transformer while matching accuracy. We report accuracy in bits per dimension, as is standard for this task. Note that lower bits per dimension is desirable. In addition to our results on V100, we exploit the memory savings of our sparse model to benchmark on an Nvidia 1080. On a significantly less powerful GPU, our sparse model is able to process 32,039 tokens per second while the standard Transformer runs out of memory. The memory savings of the sparse Transformer could also be used to train a much larger model, leading to higher accuracy.

D. Application: Sparse MobileNetV1

MobileNetV1 is an efficient convolutional neural network for computer vision tasks [43]. While originally designed for resource constrained settings, MobileNetV1 has been found to be highly efficient across platforms and has been influential in the design of computer vision models [29], [44], [45].

MobileNetV1 is made up of alternating depthwise and 1×1 convolutions. Each convolution is followed by batch normalization [46] and a ReLU non-linearity. MobileNetV1 defines a range of models with size controlled by a width multiplier. The 1×1 convolutions in these models are responsible for the large majority of the FLOPs and can be computed as matrix multiplication if the input data is stored in CHW format.

SpMM				
Model	Transformer	Transformer	ResNet-50	ResNet-50
Batch Size	1	8	1	256
-Load Balancing	96.1%	88.9%	91.7%	78.5%
-Vector Inst.	100.1%	80.9%	87.9%	64.8%
-Residue Unroll	92.0%	94.1%	87.8%	92.6%
-Index Pre-Scale	100.6%	100.6%	98.2%	100.3%

SDDMM				
Model	Transformer	Transformer	ResNet-50	ResNet-50
Batch Size	1	8	1	256
-Load Balancing	101.1%	97.1%	100.9%	96.8%
-Vector Inst.	98.3%	132.0%	120.2%	170.6%

1) Experimental Setup: We introduce sparsity into the 1×1 convolutions of MobileNetV1 using magnitude pruning [5]. We prune all models to 90% sparsity. We leave the first layer dense, as we found it to be bandwidth bound by the activation matrix and thus saw less benefits from weight sparsity. We trained our baseline models on the ImageNet [18] dataset with 32 accelerators for 100 epochs. As we target efficient inference in the regime where inference costs outweigh training costs, we increase training time for our sparse models by 10× which helps the sparse models converge while being pruned.

At inference time, batch normalization can be fused into the preceding linear operation. We do this for all depthwise and 1×1 convolutions. For depthwise convolution, we wrote kernels that support fused bias and ReLU operations. We similarly fuse bias and ReLU into our sparse 1×1 convolutions. For the 1×1 convolutions in our dense baselines we use Nvidia cuBLAS, which is backed by highly-tuned assembly kernels. We additionally wrote a fused bias + ReLU kernel, which we use following these linear operations. For our sparse models, we use an oracle kernel selector for four 1×1 convolutions where our heuristic was sub-optimal. For each model, we benchmark inference in single-precision on an Nvidia V100 GPU with a batch size of 1 image, as is common in online inference applications like self-driving cars.

2) Results & Analysis: The results of our benchmarks are shown in Figure 12 and in Table IV. Across the board, our sparse models offer speedups of 21-24% for a given accuracy, or equivalently, 1.1% higher accuracy for the same throughput.

Because of the accuracy loss from pruning, our sparse models are wider than the dense ones they match accuracy with. Increased width also increases the cost of the non-sparse operations relative to those in the dense baseline. Better pruning algorithms would help alleviate this and enable further speedups. Additionally, the depthwise convolutions become a significant bottleneck after the 1×1 convolutions are pruned. Tuning these kernels would yield further gains for our sparse models relative to their dense counterparts.

VIII. Related Work

[47] and [25] propose efficient SpMM kernels based on alternative sparse matrix formats designed for GPUs [48], [49]. [26] discuss the design of high-performance SpMM on

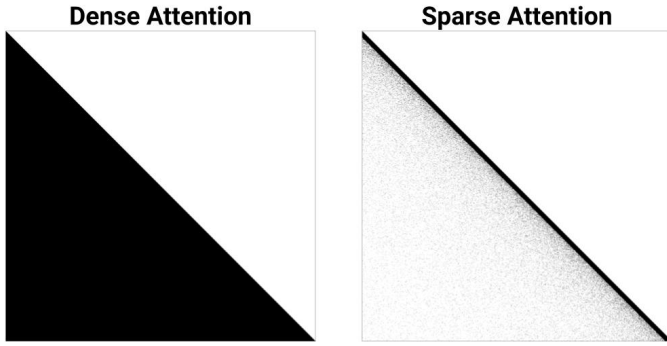**Dense Attention** | **Sparse Attention**

Fig. 11. **Transformer attention mechanism connectivity.** The upper diagonal is masked so that tokens only attend to those that came before them. **Left:** Dense all-to-all attention. **Right:** Sparse attention with a small dense band and random off-diagonal sparsity sampled with probability inversely proportional to distance from the diagonal.

TABLE III
SPARSE TRANSFORMER RESULTS

	Model	Transformer	Sparse Transformer
	Bits Per Dimension	3.76	3.77
V100	**Throughput (tokens/s)**	32,477	67,857
	Memory Usage (GB)	9.88	0.77
1080	**Throughput (tokens/s)**	out-of-memory	32,039
	Memory Usage (GB)	out-of-memory	0.88

GPUs. We compare to their approach for computing SpMM in Section VII-A and reference their taxonomy for SpMM design throughout the text. [23] propose an adaptive tiling technique, where CSR matrices are partitioned into sets of rows. Within each set, the columns are re-ordered such that columns with more nonzeros are grouped. These "heavy" groups are processed together and exploit tiled execution to enable more reuse of operands. The remaining columns are processed with a standard row-splitting scheme. We benchmark and discuss limitations of this approach in Section VII-A.

[50] implement an efficient direct sparse convolution for CPUs and demonstrate performance gains relative to dense baselines. [51] develop a technique for inducing sparsity in Winograd convolutions [52] and design and efficient implementation for CPUs. [16] design efficient SpMM kernels for CPUs and demonstrate significant performance improvements for highly efficient neural networks on mobile processors.

[13] and [15] enforce different forms of structure on sparse matrices to enable efficient mapping to GPUs. [13] develop efficient GPU kernels for block-sparse matrices and apply them to neural networks on a range of different tasks. [15] propose fixing the number of nonzeros in small regions of the sparse matrix to balance between performance and accuracy loss from enforcing structure on the topology of nonzeros.

IX. DISCUSSION & CONCLUSION

In addition to the kernels we discuss, training DNNs requires the computation $A^T B \Rightarrow C$, where A^T is the transpose of a sparse matrix. It's difficult to fuse the transpose into the SpMM for CSR matrices. However, for DNN training it's possible to cache the row offsets and column indices for A^T when the sparse matrix topology is updated and perform the transpose as an `argsort` of the matrix values. Alternative
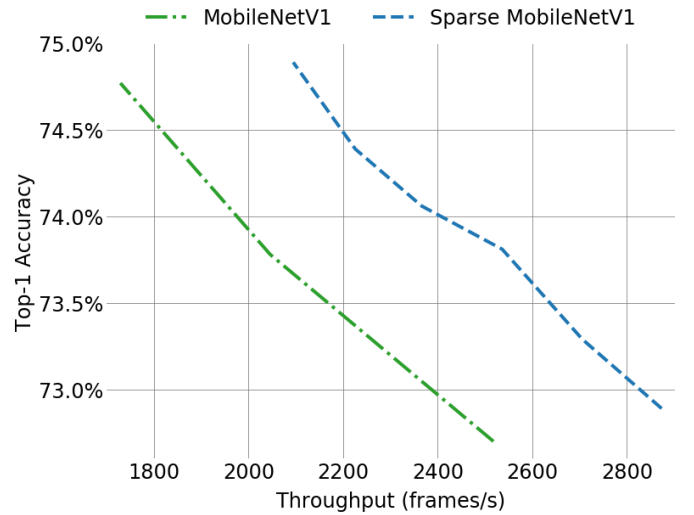

Fig. 12. **MobileNetV1 accuracy-runtime tradeoff curves.** Sparse modes are more efficient, achieving speedups of 21-24% for a given accuracy across all model sizes, or equivalently, ~1.1% higher accuracy for the same throughput.

TABLE IV
SPARSE MOBILENETV1 RESULTS

Model	Width	Accuracy	Throughput (frames/s)
Dense	1	72.7%	2,518
	1.2	73.8%	2,046
	1.4	74.8%	1,729
Sparse	1.3	72.9%	2,874
	1.4	73.3%	2,706
	1.5	73.8%	2,537
	1.6	74.1%	2,366
	1.7	74.4%	2,226
	1.8	74.9%	2,095

sparse matrix formats are an interesting direction to enable transposed and non-transposed computation [53], [54]

On large problems, the performance of our kernels is limited by shared memory bandwidth. One direction for alleviating this bottleneck is to exploit reuse of values loaded from the right input across multiple rows of the left input matrix.

While our kernels are highly efficient, they are not able to take advantage of dedicated matrix-multiply hardware. For unstructured sparsity, it's possible that unpacking sparse tiles in shared memory could enable the use of these operations. New advances in hardware are likely to enable this further [55]. Despite model quality loss, it remains possible to exploit this hardware with vector and block sparsity [12], [13], [16].

In this work, we demonstrate that the sparse matrices found in deep neural networks exhibit favorable properties that can be exploited to accelerate computation. Based on this insight, we design high-performance SpMM and SDDMM kernels targeted specifically at deep learning applications. Using our kernels, we demonstrate sparse Transformer and MobileNet models that achieve 1.2–2.1× speedups and up to 12.8× memory savings without sacrificing accuracy. We hope that our findings facilitate better support for sparsity in deep learning frameworks and more broadly enable widespread use of sparsity in deep learning.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, 2017.

[3] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient neural network," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2015.

[4] S. Narang, G. Diamos, S. Sengupta, and E. Elsen, "Exploring Sparsity in Recurrent Neural Networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[5] M. Zhu and S. Gupta, "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018.

[6] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating Long Sequences with Sparse Transformers," *CoRR*, vol. abs/1904.10509, 2019.

[7] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage," in *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, 1989.

[8] D. Molchanov, A. Ashukha, and D. P. Vetrov, "Variational dropout sparsifies deep neural networks," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 2017.

[9] C. Louizos, M. Welling, and D. P. Kingma, "Learning Sparse Neural Networks through L_0 Regularization," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[10] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The Efficient Transformer," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rkgNKkHtvB

[11] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, "Efficient content-based sparse attention with routing transformers," *CoRR*, vol. abs/2003.05997, 2020.

[12] S. Narang, E. Undersander, and G. F. Diamos, "Block-Sparse Recurrent Neural Networks," *CoRR*, vol. abs/1711.02782, 2017.

[13] S. Gray, A. Radford, and D. P. Kingma, "Block-sparse gpu kernels," https://blog.openai.com/block-sparse-gpu-kernels/, 2017.

[14] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. van den Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient Neural Audio Synthesis," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 2018.

[15] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient DNN inference on GPU," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, 2019, pp. 5676–5683.

[16] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast Sparse Convnets," *CoRR*, vol. abs/1911.09723, 2019.

[17] T. Gale, E. Elsen, and S. Hooker, "The State of Sparsity in Deep Neural Networks," *CoRR*, vol. abs/1902.09574, 2019.

[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[19] O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Baltimore, Maryland, USA: Association for Computational Linguistics, June 2014, pp. 12–58. [Online]. Available: http://www.aclweb.org/anthology/W/W14/W14-3302

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.

[21] P. Micikevicius, "GPU Performance Analysis and Optimization," http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf, 2012.

[22] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, "Sampled dense matrix multiplication for high-performance machine learning," in *25th IEEE International Conference on High Performance Computing, HiPC 2018, Bengaluru, India, December 17-20, 2018*. IEEE, 2018, pp. 32–41.

[23] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 300–314.

[24] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi, "cuSPARSE Library," https://www.nvidia.com/content/GTC-2010/pdfs/2070_GTC2010.pdf, 2010.

[25] H. Anzt, S. Tomov, and J. J. Dongarra, "Accelerating the LOBPCG method on gpus using a blocked sparse matrix vector product," in *Proceedings of the Symposium on High Performance Computing, HPC 2015, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015*. SCS/ACM, 2015, pp. 75–82.

[26] C. Yang, A. Buluç, and J. D. Owens, "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*, 2018, pp. 672–687.

[27] S. Narang, "DeepBench: Benchmarking Deep Learning Operations on Different Hardware," https://github.com/baidu-research/DeepBench, 2016.

[28] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, "Image Transformer," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 2018.

[29] M. Tan and Q. V. Le, "Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019.

[30] J. Luitjens, "CUDA Pro Tip: Increase Performance with Vectorized Memory Access," https://devblogs.nvidia.com/

cuda-pro-tip-increase-performance-with-vectorized-memory-access/, 2013.

[31] Nvidia, "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2019.

[32] D. Merrill and M. Garland, "Merge-Based Parallel Sparse Matrix-Vector Multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016.

[33] A. Kerr, D. Merrill, J. Demouth, and J. Tran, "CUTLASS: Fast Linear Algebra in CUDA C++," https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/, 2017.

[34] S. Pai, "How the Fermi Thread Block Scheduler Works," https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html, 2014.

[35] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1425–1439, 1987.

[36] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[37] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[38] A. Adinets, "Adaptive parallel computation with cuda dynamic parallelism," https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/, 2014.

[39] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[40] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, A. Moschitti, B. Pang, and W. Daelemans, Eds. ACL, 2014, pp. 1724–1734.

[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 2978–2988.

[43] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[44] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 4510–4520.

[45] A. Howard, R. Pang, H. Adam, Q. V. Le, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, and Y. Zhu, "Searching for mobilenetv3," in *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 2019, pp. 1314–1324.

[46] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 448–456.

[47] F. Vázquez, G. O. López, J. Fernández, I. García, and E. M. Garzón, "Fast sparse matrix matrix product based on ELLR-T and GPU computing," in *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012, Leganes, Madrid, Spain, July 10-13, 2012*. IEEE Computer Society, 2012, pp. 669–674.

[48] F. Vázquez, J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on NVIDIA gpus," *Concurr. Comput. Pract. Exp.*, vol. 23, no. 8, pp. 815–826, 2011.

[49] H. Anzt, S. Tomov, and J. J. Dongarra, "Implementing a sparse matrix vector product for the sell-c / sell-c-σ formats on nvidia gpus," 2014.

[50] J. Park, S. R. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster cnns with direct sparse convolutions and guided pruning," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[51] S. R. Li, J. Park, and P. T. P. Tang, "Enabling sparse winograd convolution by native pruning," *CoRR*, vol. abs/1702.08597, 2017.

[52] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 4013–4021.

[53] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, F. M. auf der Heide and M. A. Bender, Eds. ACM, 2009, pp. 233–244.

[54] J. Li, J. Sun, and R. W. Vuduc, "Hicoo: hierarchical storage of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 2018, pp. 19:1–19:15.

[55] Nvidia, "NVIDIA A100 tensor core gpu architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.