Spartan: A Sparsity-Adaptive Framework to Accelerate Deep Neural Network Training on GPUs

Shi Dong, *Member, IEEE,* Yifan Sun, *Member, IEEE,* Nicolas Bohm Agostini, *Student Member, IEEE,* Elmira Karimi, *Student Member, IEEE,* Daniel Lowell, Jing Zhou, *Member, IEEE,* José Cano, *Senior Member, IEEE,* José L. Abellán, *Member, IEEE,* and David Kaeli, *Fellow, IEEE*

Abstract—Deep Neural Networks (DNNs) have emerged as an important class of machine learning algorithms, providing accurate solutions to a broad range of applications. Sparsity in activation maps in DNN training presents an opportunity to reduce computations. However, exploiting activation sparsity presents two major challenges: i) profiling activation sparsity during training comes with significant overhead due to computing the degree of sparsity and the data movement; ii) the dynamic nature of activation maps requires dynamic dense-to-sparse conversion during training, leading to significant overhead.
In this paper, we present *Spartan*, a lightweight hardware/software framework to accelerate DNN training on a GPU. Spartan provides a cost-effective and programmer-transparent microarchitectural solution to exploit activation sparsity detected during training. Spartan provides an efficient sparsity monitor, a tile-based sparse GEMM algorithm, and a novel compaction engine designed for GPU workloads. *Spartan* can reduce sparsity profiling overhead by 52.5× on average. For the most compute-intensive layers, i.e., convolutional layers, we can speedup AlexNet by 3.4×, VGGNet-16 by 2.14×, and ResNet-18 by 2.02×, when training on the ImageNet dataset.

Index Terms—DNN, Sparsity, GPU.

1 Introduction

Over the past decade, Deep Neural Networks (DNNs) have become a popular approach to address such demanding applications as image/speech recognition, object localization/detection, natural language processing, and guidance/navigation [1]. Researchers have been able to achieve high inference accuracy for many of these applications, inspiring new classes of smart products, reshaping our society and our daily lives. Although many variants of DNN models exist (e.g., Convolutional Neural Networks and Recurrent Neural Networks [1]), the dominant computations used during their training are matrix-based operations (General Matrix Multiplication or GEMM). Spurred on by the emergence of high performance platforms that are able to perform billions of matrix-based operations efficiently, training a large-scale DNN has become a reality [2], [3], [4]. GPUs have been used effectively to efficiently train a DNN [5], [6], [7], thus enabling this class of algorithms to reach unprecedented popularity.

The computer architecture community has explored methods to improve execution efficiency and memory usage when processing DNNs. Of the many approaches pursued, leveraging weight/activation sparsity has attracted a lot of attention [8], [9], [10], [11], [12], [13]. Prior studies have explored exploiting sparsity to accelerate DNN computations during both training and inference on customized platforms (e.g., FPGAs and ASICs) [4], [8], [9], [12]. For inference, the major source of sparsity occurs after applying weight sparsification and quantization [14]. In contrast to inference, for training, sparsity is produced by the ReLU activation function during both forward and backward propagation, and in the Max Pooling layer during backward propagation [10]. A recent study found that there can be as much as 90% activation sparsity in AlexNet [15].

Currently, leveraging sparsity for accelerating training mainly targets customized platforms [4]. Sparsity during training on a GPU is largely under-explored, even though GPUs still serve as the primary platform for training large-scale DNNs [16]. GPUs can effectively accelerate both dense [17], [18] and sparse [19], [20] matrix operations. Given the large number of zero values detected during training, employing sparse matrix operations should be able to further reduce training time on a GPU. We focus on leveraging activation sparsity for the following reasons: 1) we observe limited weight sparsity (less than 1%) during DNN training, and 2) we do not explore weight sparsification and quantization methods in order to keep training lossless. We focus on accelerating convolutional layers, as they are the most compute-intensive layers in a DNN and take over 90% of the overall execution time [21].

DNN training presents many challenges when attempting to leverage activation sparsity. The DNN training is an iterative and dynamic process. The contents of the activation maps keep changing due to the randomly selected inputs (e.g., a batch of images) throughout the training. Therefore, efficiently leveraging activation sparsity during training presents the following challenges: i) tracking and profiling

- *S. Dong, Y. Sun, N. Agostini, E, Karimi and D. Kaeli are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, 02115.*
  E-mail: {dong.sh, sun.yifa, bohmagostini.n, karimi.e}@northeastern.edu and kaeli@ece.neu.edu
- *D. Lowell and J. Zhou are with AMD.*
  E-mail: {Daniel.Lowell, Jing.Zhou2}@amd.com
- *J. Cano is with University of Glasgow, United Kingdom.*
  E-mail: Jose.CanoReyes@glasgow.ac.uk
- *J. L. Abellán is with UCAM Universidad Catolica de Murcia, Spain.*
  E-mail: jlabellan@ucam.edu

data sparsity introduces high overhead due to the need to compute the degree of sparsity and the data movement between the CPU and the GPU; ii) the contents of activations change dynamically throughout every iteration, requiring low-overhead dynamic sparse format conversion. The conversion overhead of current popular sparse matrix formats is too high for dynamic conversion because: 1) the generation of the indexing information is an inherently serial process, and 2) multiple data structures are needed, involving many write operations. As a consequence, the overhead of dense-sparse matrix format conversion can cancel out the benefits of using sparse matrix operations.

In this paper, we present *Spartan*, a sparsity-adaptive framework to accelerate training of DNNs on a GPU. We first characterize sparsity patterns present in activations used in DNN training. We also consider the current state-of-the-art approaches for managing sparse matrices. Then, we highlight the challenges of leveraging the sparse data while trying to accelerate DNN training. Based on key observations from our characterization results, we propose the Spartan framework to address these challenges. This paper makes the following contributions:

- We propose a novel sparsity monitor that intelligently acquires and tracks activation sparsity with negligible overhead. Using our monitor, we can significantly reduce sparsity profiling overhead by 52.5×, on average. We adopt a periodical monitoring technique, whose behavior is regulated by two algorithms: i) a Dynamic Period Adjustment Algorithm (DPAA), and ii) a Sparsity Stability Detecting Algorithm (SSDA). The former dynamically adjusts the monitoring period, while the latter detects the sparsity stability level.
- We propose a novel sparse format ELLPACK-DIB (Data Index Bundling) based on ELLPACK-R [22], and design a customized tile-based sparse GEMM algorithm that uses this format.
- We propose a novel compaction engine located between the L2 cache and main memory of the GPU, enabling dynamic compaction/conversion. It serves as a near memory processing unit, responsible for compacting sparse data into the ELLPACK-DIB format during kernel execution. The compaction engine consists of three major components: i) a Sparsity Information Block (SIB), ii) a Compacting Processor, and iii) a Prefetch Buffer. We further customize the GEMM algorithm to utilize our enhanced GPU architecture incorporating the Spartan compaction engine.
- We evaluate our sparsity monitor during the training process with five commonly-used DNN models, using both CIFAR-10 and ImageNet datasets. We find that the average overhead introduced by profiling is only 1.7%. We evaluate the compaction engine using our customized sparse GEMM algorithm. For convolutional layers, we can achieve an average speedup of 3.4× for AlexNet, 2.14× for VGGNet-16, and 2.02× for ResNet-18, when training on the ImageNet dataset.

To our best knowledge, this is the first work that exploits activation sparsity to accelerate DNN training on a GPU. Our proposed framework can be generalized to all GPU types, and any DNN model that uses ReLU activation functions, pooling layers, and other types of layers that generate zeros.

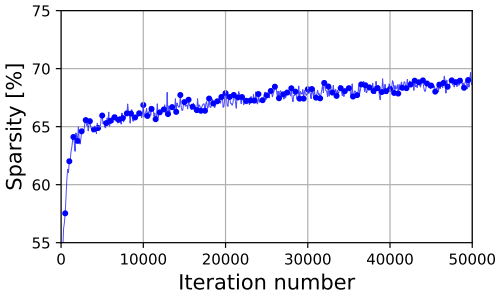


Fig. 1: The sparsity trend from one activation map of CifarNet.

2 Motivation

The degree of sparsity present in data enables us to leverage sparse computations, which have been widely used in HPC applications [23] and DNN models [4], [8]. Sparse computations can significantly improve execution performance, since they can skip zeros when performing multiplication and summation operations, reducing the total number of operations executed. In this section, we present our characterization of activation sparsity observed during DNN training. We also review the performance associated with state-of-the-art sparse matrix multiplication solutions. Given the high sparsity level (up to 80%) observed during the training process, and the inherent inefficiencies of existing sparse matrix multiplication solutions [19], [20], [24], we are motivated to develop our Spartan framework that incorporates both software and hardware features. We focus on characterizing activation sparsity for two reasons: 1) we observe limited weight sparsity during DNN training, and 2) we do not explore weight sparsification and quantization methods in order to keep training lossless.

2.1 Activation Sparsity in DNN Training

Popular DNN models [25], [26], [27], [28] employ linear layers that include convolutional and fully-connected layers, as well as non-linear layers that include max-pooling layers and ReLU activation functions. The ReLU activation function outputs zeros if the input is negative, creating sparsity. Also, a max-pooling layer performs a downsampling operation during forward propagation and an upsampling operation during backward propagation. The upsampling operation also generates a large number of zeros.

Figure 1 illustrates the sparsity trend of one activation map generated by a ReLU activation function after the second convolutional layer of CifarNet [29], trained using the CIFAR-10 dataset. From the figure, we notice significant sparsity (around 70%), even though there is great diversity in the input values across training iterations. We observe similar trends across different variants of DNN models, such as AlexNet, VGGNet, and ResNet. These trends and patterns are also preserved when we randomly shuffle the inputs. We can summarize the sparsity patterns observed as follows:

1) The sparsity patterns and distribution of zeros change over time. This is because the training input changes on each iteration.

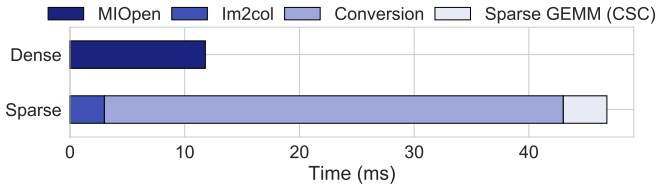


Fig. 2: Execution time breakdown for convolution using clSparse and rocSparse, compared with MIOpen.

2) Activation maps from different layers contain different degrees of sparsity and sparsity trends over time.
3) The sparsity exhibits negligible variance during short periods across consecutive training iterations.
4) The sparsity level increases gradually, then remains stable across many training iterations.

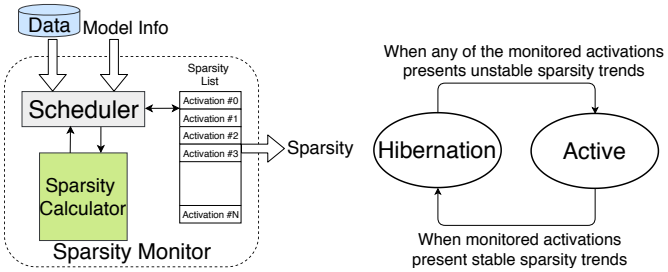
According to observations 1 and 2, we need an *efficient sparse format conversion* to *dynamically* convert data stored in a dense format to its sparse format, enabling efficient usage of sparse matrix operations on demand. In addition, observations 3 and 4 suggest that *exhaustive profiling of sparsity during every iteration is not needed*, motivating us to develop an efficient profiling mechanism.

2.2 Characterization of Sparse Matrix Operations

Both convolutional and fully-connected layers use General Matrix Multiplication (GEMM) as their primary computational kernel. The computation in the fully-connected layers can be directly represented by GEMM, while the computation in the convolutional layers can be a combination of an *im2col* operation (transforming high-dimensional activation/feature maps into a 2D matrix) and a GEMM operation [5]. Convolutional layers dominate the overall DNN training time. In particular, the convolutional layers alone can contribute to approximately 90% of the training time [21], [30]. Therefore, in this paper we are focused on improving GEMM-based convolutional layer performance, given its dominance on training performance.

Next, we capture the execution time of a convolutional layer during the forward propagation operation, comparing the performance when using a popular Compressed Sparse Column (CSC) [31] and a dense format. We configure the convolutional layer using a filter size of $5 \times 5 \times 64 \times 64$ (CifarNet [29]). We evaluate the convolutional layer using a dense format with AMD's MIOpen, using both *im2col* and dense GEMM kernels [32]. When using a sparse format, we include the dense-to-sparse conversion (implementation provided by the clSparse [24] library) between the *im2col* and the sparse GEMM (using the implementation in the rocSparse library [20]).

The execution time breakdown, as shown in Figure 2, suggests that using a sparse matrix multiplication can significantly reduce the GEMM execution time. However, as the dense-to-sparse format conversion introduces large overhead (approximately 85% of the overall execution time), the overall execution time of the convolutional layers increases when using a sparse format. To exploit the sparsity present in DNN training, we need a new solution that can hide the dense-to-sparse conversion overhead and accelerate the convolutional layers.



(a) (b)

Fig. 3: (a) The overview of the sparsity monitor. (b) State transition of Dynamic Monitoring Period Management.

3 Sparsity Monitor

In this section, we present our sparsity monitor design, providing efficient and flexible sparsity monitoring during DNN training. The sparsity monitor detects activation sparsity before each convolutional layer in a DNN model, determining when to leverage our sparse GEMM kernel acceleration dynamically, depending on the detected sparsity level. Running on the CPU, our monitor is designed to reduce the overhead of computing the degree of sparsity and the data movement between the CPU and GPU. Figure 3a presents an overview of the sparsity monitor, which consists of three components: 1) a scheduler, 2) a sparsity list, and 3) a sparsity calculator. The regular workflow of the sparsity monitor is as follows. First, the model information (i.e., the structure of the DNN model and a list containing which activation maps we select to monitor) is sent to the scheduler to initiate the monitoring process. Next, the scheduler determines when to profile the activation maps and enables the sparsity calculator to compute the degree of sparsity based on the selected activation maps (Data). Then the scheduler manages and updates the sparsity list that contains the sparsity for individual activation maps being monitored.

The scheduler manages and monitors each individual activation map by regulating two important parameters: 1) the monitoring period, and 2) the monitoring duration. The former determines the timing gap between two monitoring processes, while the latter indicates the length of the monitoring process. To avoid exhaustive monitoring, the monitoring duration should be smaller than the monitoring period. Once determined by user, the monitoring duration remains the same across all monitoring processes. The monitoring period, on the other hand, can be dynamically changed to further reduce the profiling overhead. Specifically, we consider the importance of observations 2, 3 and 4 presented in Section 2.1. We propose multiple mechanisms to assist with periodic monitoring, dynamically adjusting the monitoring period. The mechanisms are: i) flexible monitoring, ii) fast termination, and iii) dynamic management of the monitoring period management.

i) Flexible Monitoring provides a mechanism to regulate the monitoring period in a flexible manner. For each individual activation map, the monitoring process can have a different monitoring period. As per observation 2, activation maps from different layers present different levels of sparsity and the sparsity associated with these activation maps may change over time. Therefore, flexible monitoring can be more

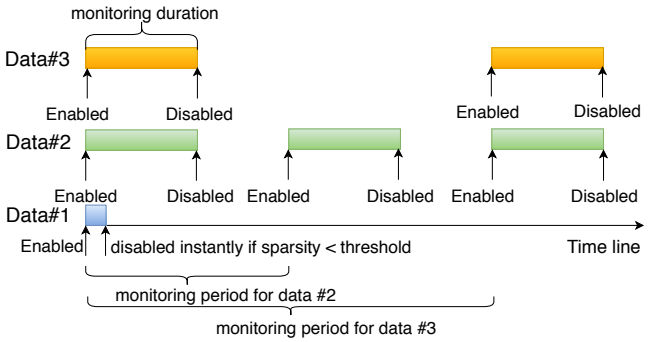


Fig. 4: Three monitoring processes with flexible monitoring and fast monitoring disabled.

efficient in managing the monitoring process than using only a single monitoring period. Figure 4 shows two examples of a flexible monitoring process. In particular, the Data#2 (green monitoring process) and Data#3 (orange monitoring process) are two monitored activation maps that have different monitoring periods.

ii) Fast Termination stops the monitoring process if the detected sparsity level is below a user-defined threshold. By doing this, the monitoring duration is reduced to avoid useless monitoring. Figure 4 also shows an example of fast termination where the monitoring process stops for Data#1 (blue monitoring process) immediately after a low sparsity level is detected.

iii) Dynamic Monitoring Period Management is the most effective mechanism to reduce the overhead associated with the sparsity computation and data movement, tuning the monitoring period dynamically according to sparsity variance throughout many training iterations. Dynamic Monitoring Period Management maintains a two-state finite state machine: 1) *Active* and 2) *Hibernate*. We introduce these two states to handle two possible scenarios according to observation 4: 1) when sparsity changes gradually and 2) when sparsity remains at a stable level. In the Active state, the monitoring period is shorter and can dynamically respond based on a Dynamic Period Adjustment Algorithm (DPAA), depending on the current sparsity trends. While in the Hibernate state, the monitoring period is longer and shared by all monitored activation maps for simplicity (further details below). The transitions between the two states are shown in Figure 3b. The conditions for transitions are regulated by Algorithm 2, a Sparsity Stability Detecting Algorithm (SSDA).

DPAA is enabled only in the Active state, adjusting the monitoring period of each individual activation map. The DPAA (Algorithm 1) can adjust the monitoring period by maintaining history and storing the most recent measured sparsity for every monitored activation map managed in the sparsity list. The algorithm first collects the measured sparsity and adds it to a history list, saving only a number of the most recent measured sparsity. Then it checks the difference between the most recent sparsity and the oldest in the history list. If the absolute difference is smaller than a threshold, we double the length of the monitoring period in the Active state. The process continues until the monitoring period is larger than the one used in the Hibernate state.

Algorithm 1 Dynamic Period Adjustment Algorithm (DPAA)

```
Inputs: sparsity, history, max_length, threshold, active_period, hibernate_period
Outputs: N/A
if sizeof(history) < max_length then
                                   ▷ Collecting the sparsity history
    idx = sizeof(history)
    history[idx] = sparsity
else
                                   ▷ Increase the monitoring period
    if abs(sparsity - history[0]) < threshold then
        active_period *= 2
    end if
    if active_period > hibernate_period then
        active_period = hibernate_period
    end if
    history.Update(sparsity)
end if
```

SSDA detects the stability level of sparsity, determining when to transition between the Active and Hibernate state. As described in Algorithm 2, in the Active state, the algorithm polls through the sparsity list in Figure 3a. If the sparsity levels of all selected monitored activation maps become stable, the state can transit to Hibernate, a state where all activation maps share the same monitoring period. While in the Hibernate state, if any of the monitored activation maps exhibit unstable sparsity trends, the state transits to Active, resetting the monitoring period for all monitored activation maps to the initial monitoring period.

4 ELLPACK-DIB Based GEMM

In this section, we explore a novel sparse format named *ELLPACK-DIB* and a tile-based GEMM algorithm designed to effectively exploit this format.

As discussed in Section 2, when using the CSC format, the dense-to-sparse conversion is costly in terms of execution time. Regular sparse formats, such as CSR/C [31] and COO [31], use multiple data structures to store the non-zero elements and indexing information. From our analysis of these formats, we have identified two major factors contributing to the dense-to-sparse conversion overhead. First, given a sparse matrix with size $M \times N$, storing the non-zero elements and calculating indexing information comes with a time complexity of $O(M \times N)$. This is because storing a non-zero element and calculating the associated indexing information depends on the location and index of the previous non-zero element. This dependency leads to a serialized conversion process that lacks any parallelism. Second, we encounter a number of writes needed to update multiple data structures for the non-zero elements and associated indexing information. Note that writes are commonly expensive and should be avoided as much as possible.

We find that one variant of ELLPACK, ELLPACK-R [22], has a structure wherein conversion can be parallelized. ELLPACK-R in row-major order requires three data structures. The first one stores non-zero elements. The second one stores the column index. The third one stores the number of

Algorithm 2 Sparsity Stability Detecting Algorithm (SSDA)

```
Inputs: state, sparsity_list, sparsity, history, max_length, threshold, active_period, hibernate_period
Outputs: N/A
if state == ACTIVE then
    hibernation_ready_count = 0
    for i = 1 to sizeof(sparsity_list) do
        if active_period == hibernate_period then
            hibernation_ready_count++
        end if
    end for
    if hibernation_ready_count == sizeof(sparsity_list) then
        state.transit(HIBERNATE)
    end if
end if
if state == HIBERNATE then
    if sizeof(history) < max_length then
        idx = sizeof(history)
        history[idx] = sparsity
    else
        if abs(sparsity - history[0]) >= threshold then
            active_period.reset(all)
            state.transit(ACTIVE)
            history.clear()
        end if
    end if
    history.Update(sparsity)
end if
```

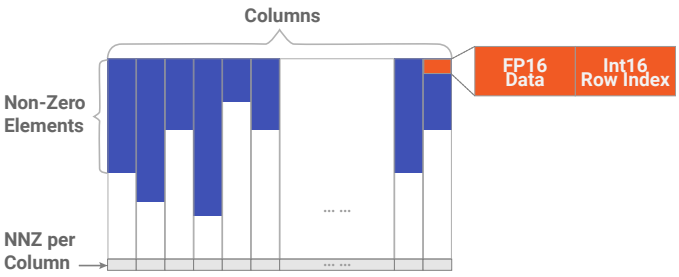


Fig. 5: The format of ELLPACK-DIB.

non-zero elements per row. The data structures for each row are completely independent of each other, meaning that the dense-to-sparse conversion can be parallelized, reducing the time complexity to $O(N)$.

Although the time complexity of the conversion is significantly reduced, there are still three data structures. To simplify the data structures, we propose ELLPACK-DIB (Data Index Bundling). Considering that the data structures for storing non-zero elements and column indices are exactly the same size in ELLPACK-R, ELLPACK-DIB bundles the data and the indices together, storing them in a single data entity. Since we only use ELLPACK-DIB for activation maps rather than weights, we use a *column-major* ELLPACK-DIB format, as shown in Figure 5, producing an efficient data access pattern for sparse GEMM.

Figure 5 presents the ELLPACK-DIB format in detail. As indicated in the figure, we only need two data structures to store the non-zero elements bundled the row index, and the

Algorithm 3 Tile-based GEMM algorithm Using ELLPACK-DIB (GPU kernel)

```
Inputs: A, lda, B, ldb, ldc, nnz_col, tile_size
Outputs: C
row = MappingRow(group_id, local_id)
col = MappingCol(group_id, local_id)
sum = 0
local_B[tile_size][tile_size]
local_A[tile_size][tile_size]
for i = 1 to nnz_col[col] with step tile_size: do
    idx = MappingTile(i, col, ldb)
    x = local_id / tile_size
    y = local_id mod tile_size
    local_B[x][y] = low2float(B[idx])
    row_idx = high2int(B[idx])
    local_A[x][y] = A[row + row_idx * lda]
    Synchronization
    for j = 1 to tile_size do
        b_idx = MappingLocalB(j, local_id)
        a_idx = MappingLocalA(j, local_id)
        sum += local_B[b_idx] * local_A[a_idx]
    end for
end for
C[row + col * ldc] = sum
```

number of non-zero elements (NNZ) per column. We use IEEE half-precision floating point format (FP16) for the data and a 16-bit unsigned integer for the row index. We only use this format for converting activations during DNN training. The actual computation in the sparse GEMM (described below) still uses single precision (FP32) by converting FP16 data back to FP32. Therefore, the precision lost during conversion has little impact on the overall training performance [10], [33]. We use the second 16 bits to represent the row index. For the index field, we maintain positional information so that ELLPACK-DIB-based GEMM operations can generate the same output as the dense GEMM operations. For the row index, a 16-bit unsigned integer representation is sufficient for all possible problem sizes in commonly-used DNN models. In particular, the largest convolutional layer in both ResNet and VGGNet has 4,608 elements ($3 \times 3 \times 512$) in one column after performing the *im2col* operation [26], [28].

We propose a tile-based GEMM algorithm using ELLPACK-DIB on GPUs. Algorithm 3 presents the kernel details. Note that we use some OpenCL kernel parameters, e.g., local_id and group_id [34]. The mapping functions (i.e., MappingRow, MappingCol, MappingTile, etc.) shown in the algorithm are used to map the kernel parameters to a specific location for memory access.

Our algorithm involves three matrices: 1) A, a dense matrix for weights, 2) B, a sparse matrix for activations in ELLPACK-DIB format, and 3) C, a dense matrix which is the output. The parameters lda, ldb, ldc correspond to the leading dimension of matrices A, B, and C, respectively. Usually, the leading dimension is equal to the number of rows of the matrix. Each tile of matrix C is mapped to a workgroup [34], and each element of the tile is mapped to a workitem/thread [34]. The threads in one workgroup first load a tile of B in ELLPACK-DIB format, unpacking the data and

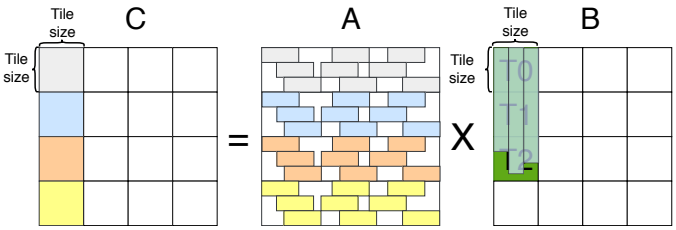


Fig. 6: A tile-based GEMM using ELLPACK-DIB.

row index accordingly. Then a tile of A is loaded, based on the row index. Once tiles of A and B are ready, multiplication and accumulation (MAC) are performed. This process is done iteratively until we reach the number of non-zero elements per column. To avoid load imbalance due to the varying number of non-zero elements, we zero-pad the last tile of B, thus avoiding potential branch divergence within a wavefront.

Figure 6 shows an example of this algorithm. In the example, we have four tiles in C, colored with gray, blue, orange and yellow. These four tiles share the three tiles (named T0, T1 and T2) of matrix B in ELLPACK-DIB format. Only three tiles are needed, as the non-zero elements only occupy three tiles in this example. To compute the gray tile, the data from matrix A (marked with gray to match the color of the gray tile in matrix C) is selected based on the row index loaded from tile T0, T1 and T2. The white area in A indicates the unselected data. We follow a similar process for the blue, orange and yellow tiles.

We evaluate the performance of dense-to-sparse conversion of ELLPACK-DIB and the customized GEMM algorithm using this format. Compared with CSC, the conversion time is reduced by over 10× and the time for GEMM is also reduced. Using the ELLPACK-DIB, we can achieve 19% speedup over MIOpen given a 90% sparsity, for the same problem size as indicated in Section 2. However, when we decrease the sparsity (e.g., 80%), the speedup is gone. As such, the benefits are very limited due to the conversion overhead. To address this overhead, we develop a hardware-based conversion mechanism, which is described in the next section.

5 Compaction Engine

In this section, we present the design of a novel hardware component named the *compaction engine*. The compaction engine is designed as a near-memory processing unit. The main purpose of it is to convert/compact sparse data to the ELLPACK-DIB format during kernel execution, hiding conversion overhead. The compaction engine consists of three major sub-components: the sparsity information block or SIB (Section 5.2), the compaction processor (Section 5.3), and the prefetch buffer (Section 5.4).

5.1 Overview

Figure 7a presents a logical view of a GPU equipped with a compaction engine. As shown in the figure, the compaction engine logically sits between the L2 cache and main memory, serving all banks of main memory/L2 and all Compute Units (CUs) of a GPU.

The compaction engine provides compacted data in the ELLPACK-DIB format in a programmer-transparent manner.

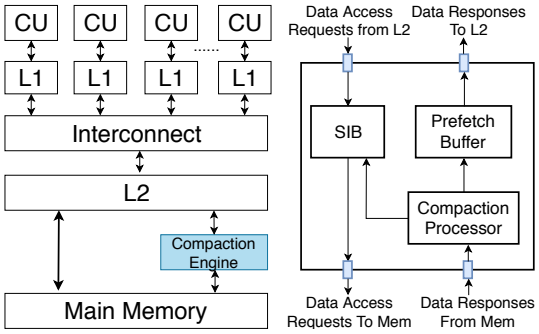


Fig. 7: (a) GPU with a compaction engine. (b) Overview of the compaction engine.

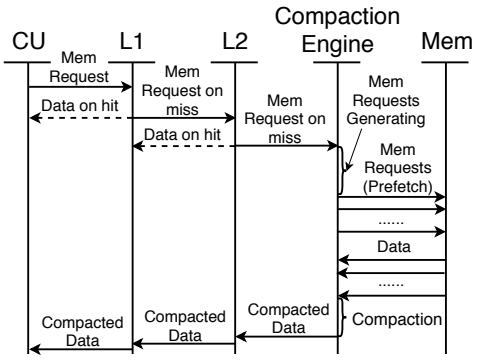


Fig. 8: An example of handling a data read request.

Figure 8 shows an example of handling a single memory request in our modified GPU with the compaction engine. Note that the compaction engine only serves data requests for reads, since only activations represented in matrix B during DNN training need compaction/conversion when running sparse GEMM. The reasons why we do not consider converting the activation maps through writes are twofold: 1) writes are costly, 64.3% slower than reads [35]. As such, converting the activation maps on writes leads to a performance degradation, and 2) dense GEMM is still used during training when the sparsity level is low. Spartan only enables sparse GEMM and the compaction engine after detecting significant levels of sparsity. From Figure 8, the steps to handle a memory request before the compaction engine are no different than those present in a regular cache system. The only difference is that memory requests issued from components above the compaction engine (i.e., CUs, L1s and L2) expect data in ELLPACK-DIB format. The compaction engine is responsible to respond to memory requests for loading sparse data, prefetching data stored in the original matrix format, compacting the data using the ELLPACK-DIB format and then returning them.

Prefetching overhead may lead to long latency when the requested compacted data requires a large amount of data from main memory. The worst-case scenario occurs when we need to load an entire column to service a single memory request. To avoid long-latency prefetching, we introduce a sparsity information block (SIB), a hardware component to store the profiled sparsity information while performing prefetching.

The primary sparsity information stored in the SIB is a series of bitmasks, where a value of "1" indicates a cache line with non-zero elements and a value of "0" corresponds to a cache line with only zeros. We describe the details in Section 5.2. With the sparsity information stored in the SIB, the compaction engine can carry out a more intelligent and efficient way to issue prefetches, reducing prefetching latency. In addition to the SIB, we also add a compaction processor and a prefetch buffer. The former compacts the prefetched data into ELLPACK-DIB format and profiles the sparsity information at the same time. The latter stores the compacted data and streamlines the process of sending data to L2.

Figure 7b presents an overview of the compaction engine with the added components. To redirect data fetches of sparse data to use the compaction engine, we add an additional bit in the memory request, and extend an existing vector load instruction (FLAT_LOAD [36]) to indicate this special data-load type. Upon receiving a memory read request for the sparse activation maps (sparse memory request), the SIB issues memory transactions for prefetching, based on the data search algorithm we propose (further details in Section 5.2.1). When the data arrives from main memory, the compaction processor is activated to perform: 1) compaction, and 2) profiling. After that, it updates the SIB using the profiled sparsity information and stores the compacted data in the prefetch buffer. Once the data is ready for the corresponding request, the prefetch buffer will send it to the L2. We describe the details of the SIB, the compaction processor and the prefetch buffer in Sections 5.2, 5.3 and 5.4, respectively.

With the compaction engine, we no longer need a costly SW-based dense-to-sparse conversion, and we do not need the data structure for storing the number of non-zero elements per column, as shown in Algorithm 3. To achieve the latter, we add an additional bit in the data response to indicate whether or not the compacted data is the last one in the column. Instead of using an additional structure nnz_col, the program uses this additional bit (set by the compaction engine) to terminate.

5.2 Sparsity Information Block

The sparsity information block (SIB) consists of multiple entries for storing sparsity profiles for each column. Each SIB entry contains two types of sparsity information (further details in Section 5.2.2): 1) the sparsity bitmask, and 2) the cache line mapping information (CMI). Because the basic unit managed in the memory system is a cache line (i.e., 64 bytes or 16 single precision (FP32) elements) [36], the SIB also manages the sparsity information on a cache line basis.

The sparsity bitmask contains a series of bits, representing the sparsity pattern present in the uncompacted data in a column. For each bit, a value of 1 indicates a non-zero cache line which has non-zero elements, whereas a 0 value indicates a cache line with only zero elements. The position of the bits is relative to the position of the cache line in the associated column. For example, the first bit indicates the cache line starting at row number 0, the second bit indicates the cache line starting at row number 16, and so on.

The CMI records the mapping information between the compacted data (to L2) and the uncompacted data (from main memory). Each mapping entry is associated with one compacted cache line (16 non-zero FP32 elements) and has two parameters: 1) the number of non-zero cache lines needed for filling 16 non-zero elements (NNC_needed), and 2) the offset indicating the last non-zero element in the last non-zero cache line. The CMI has multiple entries, each of which corresponds to a compacted cache line.

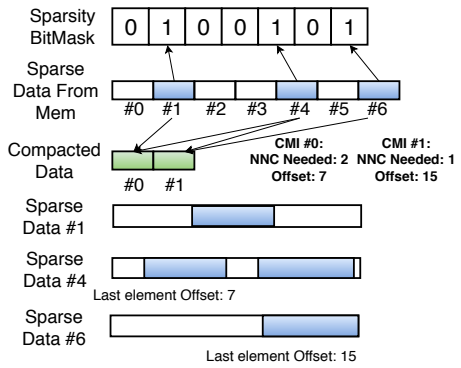


Fig. 9: An example of updating sparsity bitmask and cache line mapping information.

Figure 9 shows an example of setting the sparsity bitmask and CMI entries. In this example, we have 7 sparse cache lines (numbered #0 to #6) from main memory. Among them, only three are non-zero cache lines, shown in blue (#1, #4 and #6). The sparsity bitmask can be updated based on this layout. These 7 sparse cache lines can be compacted into 2 compacted cache lines, shown in green (#0 and #1). The compacted cache line #0 has data from sparse cache lines #1 and #4. The last element can be found in cache line #4, with an offset of 7. As such, in CMI #0, the NNC_needed value is set to 2 and the offset is set to 7. The compacted cache line #1 holds data from cache lines #4 and #6. Note that this compacted cache line also requires two sparse cache lines. However, the previous cache line (#4) is prefetched before setting CMI #1. As a result, we set the NNC_needed to 1 and the offset to 15.

5.2.1 *Data Search Algorithm*

In order to avoid long-latency prefetches, we propose a data search algorithm with three search modes that leverages the sparsity information of the SIB (Algorithms 4 and 5). Guided by the sparsity bitmask and CMI, the algorithm is able to issue memory transactions in a more intelligent manner, avoiding cache line accesses that do not contain the requested data.

The three search modes of the algorithm are as follow: 1) accurate search, 2) prudent search and 3) hasty search. Each of these modes corresponds to one potential scenario. For example, accurate search is used when the SIB holds the target CMI entry. The data search algorithm can find the locations of sparse data in main memory based on the information stored in the CMI entry. The prudent search is used when no CMI entry can be found, but a sparsity bitmask exists. This scenario can happen when a CMI entry is replaced by a newer one. In this scenario, memory transactions are issued based only on the bitmask. The hasty search mode is used when neither a sparsity bitmask nor CMI entry exists, i.e., the application is in the warm-up phase.

Function Name	Description
FindFirstNonZeroBit	Find the location of the first non-zero bit.
FindNextBitLocation	Find the location of a bit which is N non-zero bits away from a given bit location. N is an integer number.
FindNextNonZeroBit	Find the location of a bit which is the next non-zero bit from a given bit location.

TABLE 1: Descriptions of functions used in the data search algorithm.

Algorithm 4 Data Searching Algorithm (Accurate Search)

```
1: Inputs: bitmask, CMI, address
2: Outputs: mem_trans[ ]
3: entry = GetCMIEntry(CMI, address)
4: bit_location = FindFirstNonZeroBit(bitmask)
5: for i = 0 to entry.idx-1 do
6:     bit_location = FindNextBitLocation(bitmask, bit_location, CMI[i].NNCNeeded)
7: end for
8: for j = i to entry.NNCNeeded do
9:     mem_tran = AddrCalc(bit_location)
10:    mem_trans.append(mem_tran)
11:    bit_location = FindNextNonZeroBit(bitmask, bit_location)
12: end for
```

Table 1 lists the descriptions of bitmask functions used in Algorithms 4 and 5. The major operation needed for implementing these functions is a bit shift.

The accurate search accurately locates the bit locations identifying where the memory transactions should be issued, according to the cache line mapping information. The algorithm loops through all of its previous CMI entries, using the parameter NNC_needed to arrive at the desired location. Then it issues the memory transactions based on its own NNC_needed value. The prudent search first issues memory transactions from the location of the first non-zero bit in the bitmask. Then it traverses through the bitmask, issuing memory transactions only based on the non-zero bits. The hasty search issues memory transactions in a consecutive manner from the first bit of the bitmask.

5.2.2 SIB Management

Considering that the size of the compacted data in a column can vary with different column sizes, as well as the sparsity level, we adopt an approach to dynamically determine the entry size and allocate space for the sparsity bitmask and CMI. Figure 10 shows the contents of a SIB entry.

From the figure, the SIB entry is composed of three parts: 1) metadata, 2) a sparsity bitmask and 3) cache line mapping information. The metadata has fixed 6 bytes used for storing meta-information, including column number, current bit count in the bit mask and current number of valid CMI entries. We define K as the entry size in bytes, N the sparsity bitmask size in bytes, and M the CMI size in bytes. N can be determined by the column size col_size using the equation: $\lceil col\_size/128 \rceil$. K can be determined based on the column

Algorithm 5 Data Searching Algorithm (Prudent Search and Hasty Search)

```
1: Inputs: bitmask, prefetch_length, address, search_mode
2: Outputs: mem_trans[ ]
3: if search_mode == Prudent then
4:     bit_location = FindFirstNonZeroBit(bitmask)
5: end if
6: if search_mode == Hasty then
7:     bit_location = 0
8: end if
9: for j = i to prefetch_length do
10:    mem_tran = AddrCalc(bit_location)
11:    mem_trans.append(mem_tran)
12:    if search_mode == Prudent then
13:        bit_location = FindNextNonZeroBit(bitmask, bit_location)
14:    end if
15:    if search_mode == Hasty then
16:        bit_location++
17:    end if
18: end for
```

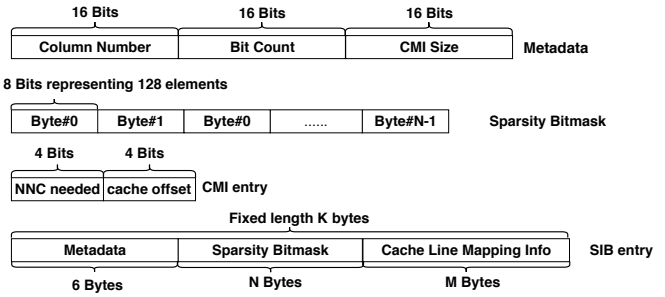


Fig. 10: The overview of a SIB entry.

size as well. M can be calculated using $K - 6 - N$, which is also the maximum allowable number of CMI entries (a CMI entry only needs 1 byte). To determine K, we heuristically categorize the column size into one of four classes: 1) larger than 4096, 2) between 2048 and 4096, 3) between 1024 and 2048 and 4) smaller than 1024. We set K to 256, 128, 64 or 32 if the column size fits in category 1), 2), 3), or 4), respectively.

Given that CMI has a limited number of entries, we propose a sliding window replacement scheme. When the CMI is full, we remove the oldest CMI entry and keep the latest entries. By using this scheme, we keep the most recent records, taking advantage of the temporal locality.

5.3 Compaction Processor

The compaction processor contains multiple compaction processing units, servicing multiple prefetched data from main memory in parallel. Figure 11 shows the organization of the compaction processor. When activated, each compaction processing unit processes the data from an associated buffer (i.e., the data buffer shown in the figure) and then sends the compacted data to the prefetch buffer and the sparsity information to the SIB.

Figure 12 shows a diagram of the compaction processing unit (ComPU). The ComPU is a special purpose SIMD unit that performs two tasks: 1) FP32 to FP16 conversion, 2)

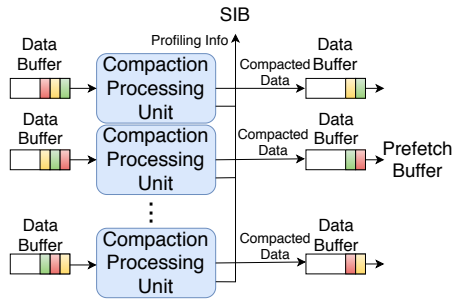


Fig. 11: The overview of the compaction processor.

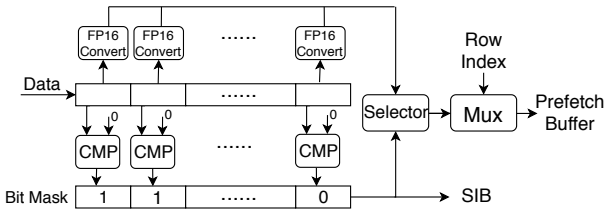


Fig. 12: The overview of the compaction processing unit.

comparisons with 0. The SIMD width is set to 16 as the memory requests to main memory are issued on a cache line basis. A fine-grained bitmask is generated according to the comparison results, reflecting the sparse pattern within the processed cache line. The ComPU uses this bitmask to select the non-zero elements, which are later bundled with the corresponding row index according to the ELLPACK-DIB format. In addition, the fine-grained bitmask is sent to the SIB as profiled sparsity information. Note that the row index can be calculated based on the memory transaction SIB issues.

5.4 Prefetch Buffer

The prefetch buffer has a multi-lane FIFO structure for storing data from the compaction processor. Each lane corresponds to an active SIB entry. During prefetching, the compacted data are stored in-order in the prefetch buffer. Once the prefetch buffer has collected one compacted cache line (i.e., 16 bundled elements) for a memory request, the data is sent to the L2. Due to prefetching, sometimes the compacted data is stored in the prefetch buffer before the sparse memory request arrives. We keep this data in the buffer. By doing so, we can take advantage of the earlier prefetch to further reduce data fetching latency.

6 Experimental Methodology

We evaluate the overhead of the software-based sparsity monitor on real hardware. Besides, we evaluate the improvements due to the hardware-based compaction engine using a state-of-the-art GPU simulator.

On the one hand, our experimental setup for evaluating the sparsity monitor uses two different classes of heterogeneous systems, targeting the training of DNN models at different scales. Both systems are equipped with CPU and GPU. The first one is a desktop-grade system, equipped with an AMD Radeon RX Vega56 [37] as the GPU platform and an Intel(R) Core(TM) i7-8700 as the CPU platform. We select VGGNet-11 [26] and ResNet-10 [28] as the DNN models and use the CIFAR-10 dataset [38] for training. The second one is a server-grade system, equipped with an NVIDIA Tesla V100 [39] as the GPU platform and an Intel(R) Xeon(R) E5-2630 as the CPU platform. We select AlexNet [25], VGGNet-16 [26] and ResNet-18 [28] as the DNN models, and use the ImageNet dataset [40] for training. We implement the sparsity monitor on TensorFlow 1.4 [41]. Table 2 shows the values for the parameters of the sparsity monitor that were described in Algorithms 1 and 2. We use the rule of thumb to determine a minimum sparsity threshold that presents performance gain using our method.

Parameter	Size/Number
Maximum History Length	10
Sparsity Threshold	0.3
Initial Monitoring Period	500
Hibernation Monitoring Period Unit	10000

TABLE 2: Specifications of the sparsity monitor.

On the other hand, we use MGPUSim [42] to model the compaction engine in a baseline AMD Radeon Instinct MI6 [43] GPU. Table 3 lists the details of the MI6. Table 4 lists the specification of the modeled compaction engine. We consume half of the space (1MB) of the baseline L2 cache for the storage of data related to the compaction engine, introducing no additional overhead for storage. In practice, the L2 space partition and the compaction engine are enabled only when launching a sparse GEMM kernel. When launching a dense GEMM kernel, the entire L2 space is used by the kernel. In our experiments with the compaction engine enabled, the portion of the L2 space devoted to compacted data is static. However, the size of the partition could be set dynamically – a direction for future work.

Next, we present the methodology used to evaluate the compaction engine. Our evaluation is presented on a layer-by-layer basis. First, we select only the computation of the convolutional layers in forward propagation in our experiments, because the computations of convolutional layers in backward propagation are transposed convolutions [44], resulting in a very similar pattern as compared to forward propagation. To demonstrate the effectiveness of the compaction engine, we first select the convolution filter with the largest filter size ($3\times3\times512\times512$) and three different activation sparsity levels (65%, 70% and 85%, which are levels observed during the training of VGGNet-11 and ResNet-10 on CIFAR-10 dataset). Then, we conduct an extensive evaluation of all convolutional layers of AlexNet, VGGNet-16, and ResNet-18, using the sparsity levels observed during the training on the ImageNet dataset. In these experiments, we collect data during training across a number of epochs (one epoch indicates one round of training involving all images in the dataset). We pause sparsity profiling of each monitored activation map when we detect stability (5 epochs when training with ImageNet, 2 epochs with CIFAR-10).

In our experiments we use two different types of input data, but with the same sparsity levels. The first is synthesized data that has random locality (i.e., *Synthetic*), where the non-zero data is uniformly distributed. The second is obtained from training on real world dataset (i.e., *Real*). Table 5 provides the details of our experiments. We select three layers

Parameter	Size/Number
CU	36
Shader Array	9
L1 Vector Cache	16KB 4-way per CU
L2 Cache	2MB (without compaction engine) 1MB (with compaction engine)
DRAM	4GB

TABLE 3: Specifications of the baseline Instinct MI6 GPU.

Parameter	Size/Number
SIB	512KB
Prefetch Buffer	512KB
Compaction Processing Unit	32

TABLE 4: Specifications of the Spartan compaction engine.

Model	Name	Sparsity	Batch Size
ResNet-10	Layer A	65%	128/256
VGGNet-11	Layer B	70%	128/256
VGGNet-11	Layer C	85%	128/256
AlexNet	Convx-y	45%-65%	128
VGGNet-16	Convx-y	35%-55%	64
ResNet-18	Convx_y-z	30%-50%	64

TABLE 5: Experimental setup for evaluating the compaction engine.

Model	Scheduler	Data Transfer	Sparsity Calculation
AlexNet	2.8%	47.2%	50%
VGGNet-16	0.2%	46.1%	53.7%
ResNet-18	0.4%	56.4%	43.2%

TABLE 6: Average execution time breakdown of the sparsity monitor (%).

(Layer A, Layer B and Layer C) as representative layers for the models, all with the same filter size. Layers A, B, and C correspond to the second convolutional layer from the fourth residual block of ResNet-10, the fifth convolutional layer of VGGNet-11 and the seventh convolutional layer of VGGNet-11, respectively. For AlexNet and VGGNet-16, we use Convx-y, where the x and y values index the convolutional layer and sparsity level, respectively. For ResNet-18, we use Convx_y-z to represent the layers, where the x, y and z values correspond to the residual block, the convolutional layer within the residual block, and sparsity level, respectively. Our baseline is a highly optimized dense matrix multiplication kernel we have selected from the AMD APP SDK [45].

7 Results and Analysis

Next, we present the evaluation results for both the sparsity monitor and the compaction engine.

Table 7 provides the execution time overhead of using exhaustive profiling (i.e., measuring sparsity every training iteration) and profiling with our sparsity monitor. Our baseline is a DNN training process without performing any sparsity profiling. The training process runs for 50,000 training iterations for VGGNet-11 and ResNet-10, and 150,000 training

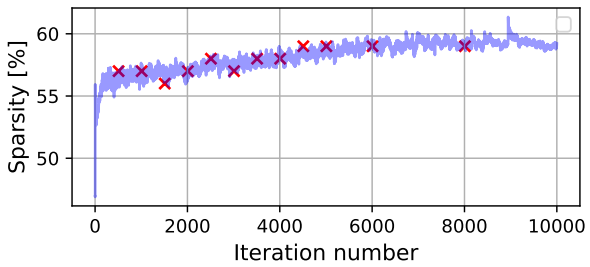


Fig. 13: The profiling process of the sparsity monitor (VGGNet-11).

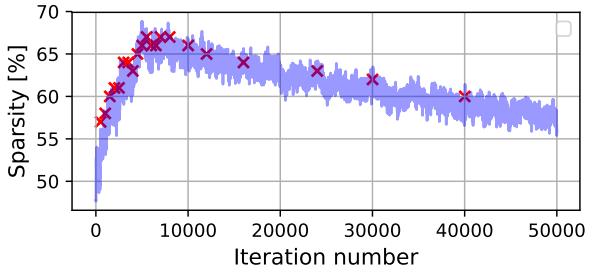


Fig. 14: The profiling process of the sparsity monitor (AlexNet).

iterations for AlexNet, VGGNet-16, and ResNet-18. From the table, we can see that profiling with the sparsity monitor has negligible overhead when training the DNN models (a maximum of 5.7%).

Table 6 shows the average execution time breakdown of the sparsity monitor in percentage of the total execution time, while monitoring AlexNet, VGGNet-16, and ResNet-18. From the table, the sparsity calculation overhead and data transfer time between CPU and GPU dominate execution. The overhead of the scheduler is negligible.

Figure 13 shows the data sparsity monitoring over time for the input of the fourth convolutional layer in VGGNet-11. Figure 14 shows the same profiling process at the input of the second convolutional layer in AlexNet. In these figures, the blue line shows the sparsity trend. The $\times$ symbols show when discrete sparsity measurements are collected by the sparsity monitor. From these results, we can see that the sparsity monitor is able to capture sparsity trends quite closely. The monitoring period (i.e., the distance between two $\times$'s) is extended whenever the sparsity monitor detects more stability in the degree of sparsity.

In terms of the performance improvements obtained using the compaction engine, we first present the speedup shown in Figure 15. Monitoring the activation maps from training VGGNet-11 and ResNet-10 with the CIFAR-10 dataset (real data), we can achieve a 1.24$\times$ average speedup using a batch size of 128, and achieve 1.56$\times$ average speedup with batch size of 256.

In Figure 15 we can also see that the benefits of our compaction engine are limited when using small image sizes (32×32) and small batch sizes. This is because the GPU suffers less memory pressure when using small image or batch sizes. In Figure 19b, the L2 cache hit rates for dense GEMM

Model Type	VGGNet-11	ResNet-10	AlexNet	VGGNet-16	ResNet-18
Dataset	CIFAR-10		ImageNet		
Exhaustive Profiling	14.9%	49.4%	21.4%	46.4%	94.2%
Profiling with the Sparsity Monitor	0.3%	0.6%	1.2%	5.7%	0.9%

TABLE 7: Overhead of two types of sparsity profiling: 1) exhaustive profiling and 2) sparsity monitor profiling. Overhead is reported relative to DNN training without any sparsity profiling.

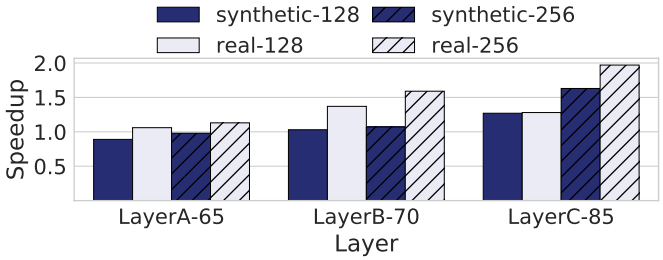


Fig. 15: The performance improvements for layers A, B, and C, using the configurations described in Table 5.

have already reached over 95% when the batch size is 128. The compaction engine has little impact on the overall performance as the memory performance of dense GEMM is already close to the upper-bound when batch size is 128. However, as we increase the batch size to 256, the L2 cache hit rate of dense GEMM drops to 83%. Then the compaction engine becomes more beneficial.

Next, in Figures 16, 17 and 18 we show the performance improvements obtained for all the convolutional layers presenting sparse inputs for AlexNet, VGGNet-16 and ResNet-18, respectively. From the figures, when training with the activation maps using the ImageNet dataset (real data), we can achieve an average 3.4× speedup for AlexNet, a 2.14× speedup for VGGNet-16, and 2.02× speedup for ResNet-18.

From these results, we highlight two interesting observations: 1) Given the same problem size (same filter size and the number of input and output channels), the speedup increases when the batch size grows, and 2) using data from the actual training has a better performance than using synthetic data. We will present a detailed analysis of these results in the next section.

7.1 Performance Analysis

To demonstrate the effectiveness of the compaction engine, we now present our performance analysis using three L2-related performance metrics: i) the number of memory transactions performed in L2, ii) the L2 hit rate, and iii) L2 read latency. We use normalized values for metrics i) and iii), relative to the value obtained when using dense GEMM (labeled as "dense"). We use the absolute value for metric ii). For demonstration purposes, we only show results from the Layer C in VGGNet-11 and the Conv4 layer in AlexNet. Other layers show similar trends. For layer C, we vary the batch size from 128 to 256. For the Conv4 layer in AlexNet, we vary the batch size from 16 to 128. Inspecting Figures 19a-19c, we can see that the compaction engine reduces the number of memory transactions and reduces the memory access latency in L2, increasing the L2 hit rate, especially when the batch size grows. The trend becomes more evident in Figures 20a-20c, where we use the ImageNet dataset. First, the compaction engine enables our proposed sparse GEMM, which has fewer memory accesses. Even though the sparse GEMM algorithm leads to poorer data locality, it has little impact on the overall performance, especially when using real-world datasets for training. Second, the pattern of non-zero elements in the activation maps is regular and consecutive, as a result of the compaction engine, leading to a higher L2 hit rate with fewer memory accesses. Last, the compaction engine's prefetching mechanism significantly reduces the average access latency, enabling the compacted data to be efficiently stored in a prefetch buffer. From our results we see that the compaction engine can achieve better memory performance when trained using a real-world dataset. This can be explained by the fact that the real data captures data locality better than the synthetic data. In Figure 19a and 20a, we noticed that the normalized number of memory transactions drops when the batch size increases. We find that the dense GEMM suffers more with increased batch size. When we increase the batch size to be n times larger, the activation maps grow by a factor of n. The compaction engine can effectively alleviate the extra pressure placed on the memory system given the same hardware configuration, reducing the number of memory accesses and caching the compacted data in the prefetch buffer.

7.2 DNN Training Speedup Modeling and Estimation

In this section, we discuss our model to accelerate DNN model training and estimate the benefits of Spartan for DNN training.

One challenge we experienced in this work is that the simulation of a full model training is extremely time consuming. Even equipped with a state-of-the-art GPU simulator such as MGPUSim [42], one training iteration of a full DNN model can take more than a day to complete. However, simulating one epoch of training using the ImageNet dataset (10,000 iterations) would take more than 27 years to complete. Second, running simulation of a full model is not needed. As pointed out in previous DNN acceleration studies [21], the convolutional layers dominate the execution time. The speedup of the convolutional layers can alone serve as a reliable predictor of the overall training performance.

To estimate the overall training performance, we capture a model to calculate the speedup for DNN training using the compaction engine. We formalize the model in Equation 1. Equipped with this model, we can easily compute the potential benefits of Spartan across an arbitrary number of training iterations.

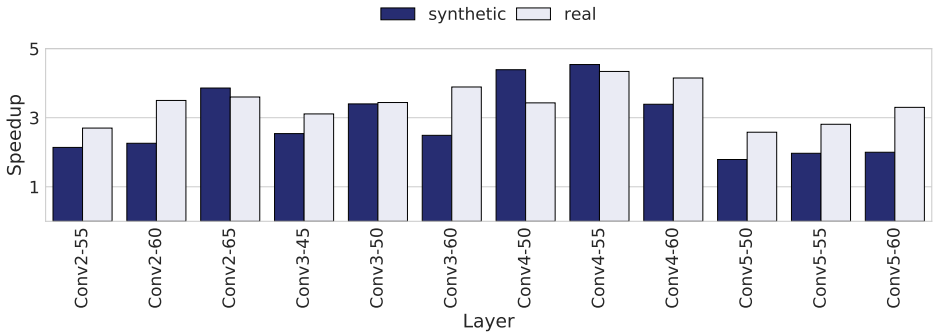


Fig. 16: Performance improvements of convolutional layers in AlexNet.

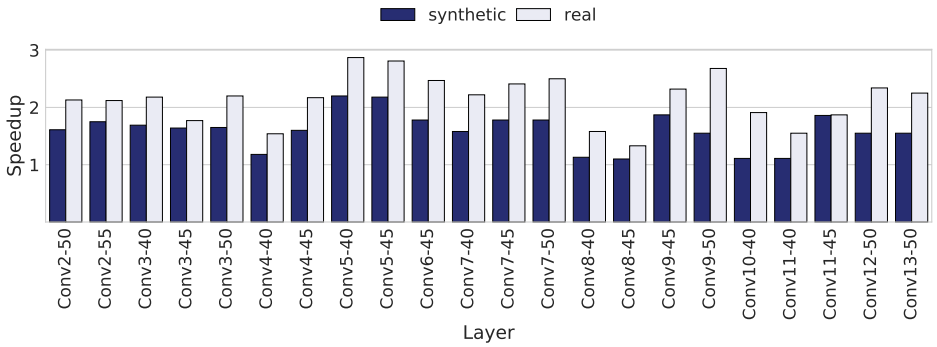


Fig. 17: Performance improvements of the convolutional layers in VGGNet-16.

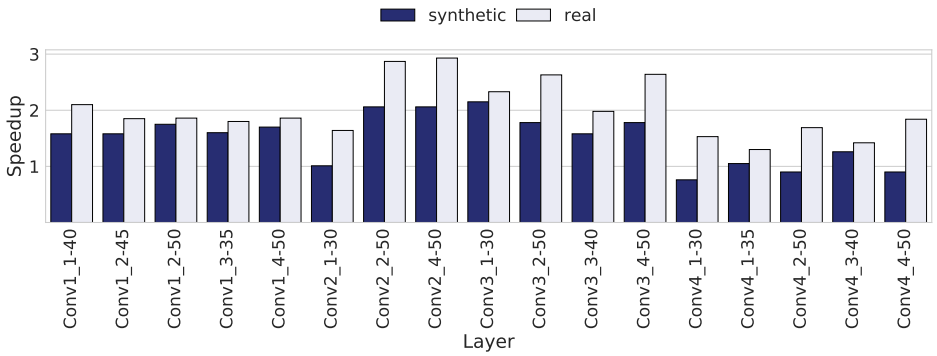


Fig. 18: Performance improvements of the convolutional layers in ResNet-18.

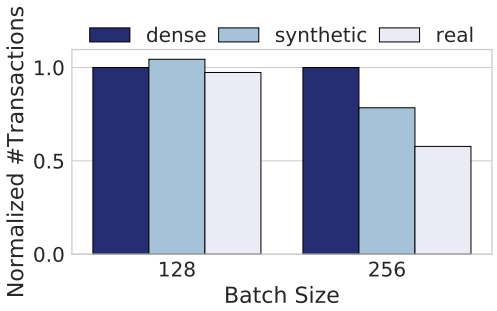
$$\frac{1}{O + \sum_{i}^{N} \sum_{j}^{M_i} P \frac{C_i R_{ij}}{S_{ij}}} \quad (1)$$

Similar to the strong scaling of Amdahl's law, we propose a speedup model that also consists of two terms, O and P. The P term represents the percentage of total GPU kernel execution that is due to the convolutional layers in the DNN. The O term is the contributions due to other execution, including time spent in other layers (e.g., reLU, batch normalization, pooling, and softmax), overhead of the GPU kernel launch, and overhead of the sparsity monitor. C_i is the proportion of execution time for each convolutional layer, where $\sum_i^N C_i = 1$ and N is the number of convolutional layers in a DNN. R_{ij} is the detected sparsity level across the entire training, where $\sum_j^{M_i} R_{ij} = 1$ is for the i^{th} convolutional layer and M_i is the detected sparsity level. S_{ij} is the speedup of the i^{th} convolutional layer given j^{th} sparsity level.

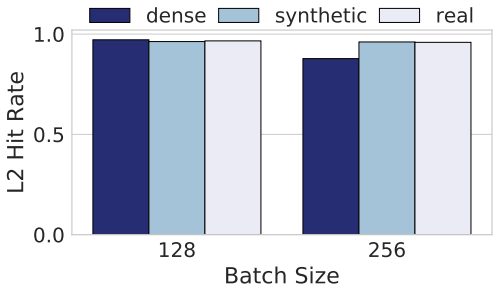
We measure O and P values using DNNMark [46], while considering the overhead of the sparsity monitor. Table 8 shows the measured P values when training AlexNet, VGGNet-16, and ResNet-18 for the ImageNet dataset, across 5 epochs. The O values can be calculated by computing $1 - P$. We also measure the C_i using DNNMark and R_{ij} using the sparsity monitor. We use the speedup obtained from the simulator as S_{ij}. Table 8 also includes the estimated speedup (2.29×, 1.87×, and 1.55×) of the training for AlexNet, VGGNet-16, and ResNet-18, with ImageNet dataset, across 5 epochs.

7.3 Hardware Cost Estimation

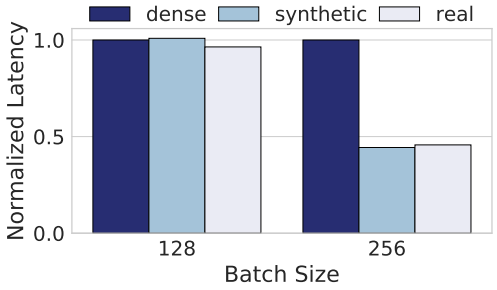
Given that Spartan is a software-hardware co-design solution, we also need to evaluate the hardware cost involved.



(a)

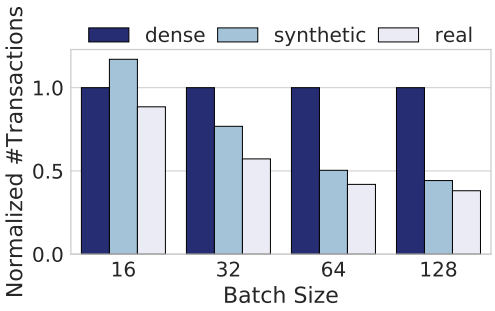


(b)

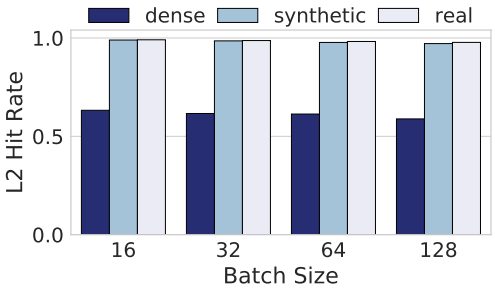


(c)

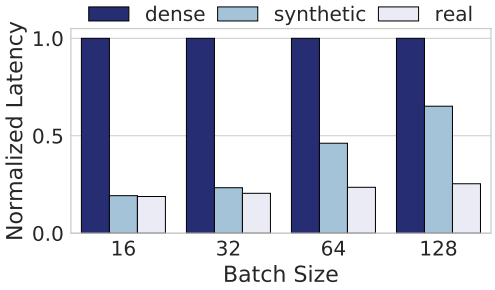
Fig. 19: Performance metric of Layer C: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.



(a)



(b)



(c)

Fig. 20: Performance metric of Conv4 in AlexNet: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.

	AlexNet	**VGGNet-16**	**ResNet-18**
P	87.40%	88.58%	82.20%
Speedup	2.29	1.87	1.55

TABLE 8: Measured P values and estimated speedup for training of AlexNet, VGGNet-16, and ResNet-18, with the ImageNet dataset, across 5 epochs.

As a major component of Spartan, the sparsity monitor is implemented in software, so no hardware cost is incurred. The only component that incurs hardware cost is the compaction engine. As the storage required by compaction engine (see Table 4) shares the space with the L2 cache, no additional hardware cost is needed for storage (though there is some minor performance overhead due to the logic for data look-up). For the processing part of the compaction engine, we need one 1-bit shifter and address calculator (one multiplier and an adder) for the SIB. We need 16 comparators, 16 FP16 converters, a 16-bit register, a selector and a multiplexer for each compaction processing unit.

8 Related Work

Exploiting sparsity (e.g., in weights and activations) to accelerate Deep Neural Networks has been widely discussed in previous research. This prior work can be divided into two main categories: 1) accelerating training and 2) accelerating inference.

Training. Jain et al. [10] proposed *Gist*, a strategy for exploiting sparsity of selected activation maps to achieve efficient data compression for saving memory space. However, this approach selects the activation maps based on the assumption that some data may contain a high degree of sparsity. The authors provide few insights as to how sparsity characteristics evolve with training. Their focus is on their optimizing memory usage and incur a 4% performance overhead. Dey et al. [47] and Cao et al. [48] present reconfigurable hardware architectures for accelerating training, which use pre-determined and structured sparsity to lower memory and computational requirements. The main problem with the proposed designs is that they cannot capture dynamic sparsity patterns. Rhu et al. [11] present a general

purpose compression DMA engine that can be used for high-performance DNN virtualization. The virtualization strategy uses the CPU's memory to increase the available memory space and improve the overall training performance. Qin et al. [4] proposed *SIGMA*, a flexible and scalable accelerator architecture, adopting a novel reduction tree microarchitecture to accelerate irregular sparse matrix operations. However, SIGMA targets custom accelerator architectures rather than GPUs.

Inference. Turner et al. [49] and Yu et al. [50] show that generating sparsity in the network weights can actually hurt inference performance, which is one of the motivations for designing specialized accelerators. For example, Han et al. [8] took advantage of pruned and quantized weights to design an energy-efficient inference engine for embedded systems, which was evaluated on nine DNN benchmarks. This work targets fully-connected layers, though only accounts for roughly 10% of the overall computation in modern DNNs. *Cnvlutin* [51] is an accelerator that follows a value-based approach to dynamically eliminate ineffectual multiplications (those that include an operand that is zero). However, the sparsity levels observed by the authors (44% on average) is lower than the sparsity observed in our study. *Minerva* [52] exploits observed network sparsity to minimize data accesses and MAC operations, and enables low-power acceleration for highly-accurate DNN prediction. While impressive work, this accelerator targets power-constrained mobile environments. *Eyeriss v2* [53] proposes an accelerator architecture designed for running compact and sparse (in the weights and activations) DNNs. However, this accelerator is mainly targeted for compact DNNs such as MobileNet. Han et al. [9] and Ren et al. [54] present efficient speech recognition engines that can work directly on a compressed sparse LSTM model and are implemented using an FPGA. However these accelerators are specific for LSTMs, not CNNs. Gray et al. [55] take advantage of the structured sparsity patterns of computation maps to accelerate inference with block-sparse matrix operations. However, this approach is evaluated only with one network model and may incur accuracy loss.

Each of the previous approaches has its own merits and limitations when exploiting sparsity. Our framework provides the following advantages over these works: i) The design of our sparse monitor is based on insights from an extensive characterization study on sparsity behavior in the training process. Therefore, it is able to efficiently and accurately measure sparsity; ii) The monitoring is done adaptively during training; iii) The sparsity is measured with a focus on activation maps instead of weights, meaning no potential model accuracy loss; iv) Finally, the additional cost for monitoring the sparsity is negligible. In particular, our novel ELLPACK-DIB format, combined with our efficient Compaction Engine hardware, enables us to accelerate sparse matrix operations.

9 Conclusions and Future Work

In this paper, we present Spartan, a framework leveraging activation sparsity to accelerate DNN training on a GPU. Our work characterizes the sparsity patterns present in activation maps during training. We highlight major challenges associated with using state-of-the-art sparse libraries: 1) the large overhead of profiling sparsity during training, and 2) the long latency of dynamic dense-to-sparse conversion.

To address these challenges, we design a novel sparsity monitor, exploiting the activation sparsity trends during training. The sparsity monitor can significantly reduce sparsity profiling overhead 52.5×, on average, leading to negligible overhead during sparsity profiling. In addition, we propose ELLPACK-DIB, a novel sparse format that addresses conversion overhead, and delivers a customized GEMM library that works with this new sparse format. To further hide the conversion overhead, we propose a novel hardware component, the compaction engine, to dynamically compact sparse data into ELLPACK-DIB format during run time. We integrate the compaction engine in an AMD Instinct MI6 GPU, simulated using MGPUSim. The results show that, for convolutional layers, we can achieve an average speedup of 3.4× for AlexNet, 2.14× for VGGNet-16, and 2.02× for ResNet-18.

In the future, we plan to perform a broader design space exploration for Spartan. We also plan to study the impact of the sparse pattern on performance.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang, "Sda: Software-defined accelerator for large-scale dnn systems," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, pp. 1–23, 2014.

[3] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40, 2015.

[4] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, 2020.

[5] 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.

[6] NVIDIA, "Nvidia dgx-2 the world most powerful deep learning system for the most complex ai challenges," 2019.

[7] AMD, "Radeon Instinct(TM) MI60 Accelerator," 2019.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, June 2016.

[9] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 75–84, ACM, 2017.

[10] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 776–789, June 2018.

[11] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 78–91, Feb 2018.

[12] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 15–28, Oct 2018.

[13] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), pp. 236–249, ACM, 2019.

[14] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[15] L. Cavigelli and L. Benini, "Extended bit-plane compression for convolutional neural network accelerators," *CoRR*, vol. abs/1810.03979, 2018.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Computer Vision – ECCV 2016*, (Cham), pp. 630–645, Springer International Publishing, 2016.

[17] NVIDIA, "cublas," 2019.

[18] AMD, "rocblas," 2019.

[19] NVIDIA, "cusparse," 2019.

[20] AMD, "rocsparse," 2019.

[21] S. Dong, X. Gong, Y. Sun, T. Baruah, and D. Kaeli, "Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, (New York, NY, USA), pp. 96–106, ACM, 2018.

[22] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón, "Improving the performance of the sparse matrix vector product with gpus," in *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 1146–1151, June 2010.

[23] J. Gao, W. Ji, Z. Tan, and Y. Zhao, "A systematic survey of general sparse matrix-matrix multiplication," 2020.

[24] GPUOpen, "clsparse," 2016.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[27] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015.

[28] 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)*, pp. 770–778, June 2016.

[29] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[30] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 209–218, April 2018.

[31] J. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[32] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, "Miopen: An open source library for deep learning primitives," 2019.

[33] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018.

[34] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2 ed., 2013.

[35] J. H. Ahn, M. Erez, and W. J. Dally, "The design space of data-parallel memory systems," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pp. 2–2, 2006.

[36] AMD, "Amd gcn3 isa architecture manual," 2016.

[37] AMD, "Radeon(TM) RX Vega56 Graphics," 2017.

[38] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[39] NVIDIA, "Nvidia tesla v100 gpu architecture," 2017.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[41] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015.

[42] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), pp. 197–209, ACM, 2019.

[43] AMD, "Amd radeon instinct mi6 accelerator," 2018.

[44] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *ArXiv*, vol. abs/1603.07285, 2016.

[45] AMD, "Amd app sdk 3.0 getting started," 2017.

[46] S. Dong and D. Kaeli, "Dnnmark: A deep neural network benchmark suite for gpus," in *Proceedings of the General Purpose GPUs*, GPGPU-10, (New York, NY, USA), p. 63–72, Association for Computing Machinery, 2017.

[47] S. Dey, K. Huang, P. A. Beerel, and K. M. Chugg, "Pre-defined sparse neural networks with hardware acceleration," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, pp. 332–345, June 2019.

[48] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), pp. 63–72, ACM, 2019.

[49] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey, "Characterising across-stack optimisations for deep convolutional neural networks," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 101–110, September 2018.

[50] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 548–560, June 2017.

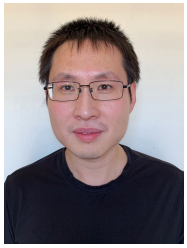
[51] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, June 2016.

[52] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, June 2016.

[53] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, pp. 292–308, June 2019.

[54] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pp. 63–72, 2019.

[55] M. Ren, A. Pokrovsky, B. Yang, and R. Urtasun, "Sbnet: Sparse blocks network for fast inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8711–8720, June 2018.



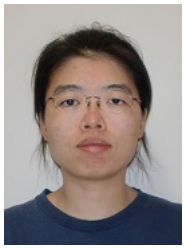
Shi Dong received his Ph.D. degree in 2020 from the Department of Electrical and Computer Engineering at Northeastern University. His research interest lies in compute support of Deep Neural Networks on GPUs, parallel computing, and high-performance computing. He received his Master of Science (M.S.) degree and Bachelor of Engineering (B.E.) degree from Beijing University of Posts and Telecommunications in 2010 and 2007. Prior to studying at Northeastern, He spent 2 years at Intel Inc. as a software engineer and 2 years at Mediatek Inc. as a DSP engineer.

Yifan Sun received his Ph.D. degree in 2020 from the Department of Electrical and Computer Engineering at Northeastern University. His research interest lies in performance modeling for CPU and GPU systems, GPU micro-architecture design, and high-performance computing. He received his Master of Science (M.S.) degree from the Department of Electrical Engineering of University at Buffalo in 2013. He received his Bachelor of Science (B.S.) degree from the Department of Electronic Science and Technology of Huazhong University of Science and Technology in 2007.

Nicolas Bohm Agostini is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Northeastern University. In 2015, he completed his Bachelor's degree in Electrical Engineering at the Universidade Federal do Rio Grande do Sul (UFRGS, Brazil). His primary research interests include Computer Architecture and High-Performance Computing. He enjoys teaching and mentoring, which he exemplified by instructing courses such as Compilers, GPU Programming, and Embedded Robotics during his graduate career. Recently, he has been working on projects that focus on accelerating machine learning and linear algebra applications by proposing new algorithmic optimizations or computer architecture features.

Elmira Karimi received her M.S. and B.S. in Electrical Engineering from Sharif University of Technology in 2013 and 2011. Currently she is a Ph.D. candidate in Computer Engineering at Northeastern university. She is a member of the Northeastern University Computer Architecture Research Laboratory (NUCAR). Her research main focuses are in high performance computing, GPU architecture, memory system and data structures for parallel processing.

Daniel Lowell is the technical lead for MIOpen, AMDs open source deep learning GPU kernels library. He has over ten years of industry experience in the GPU and HPC space. Interests include machine learning, neuroscience and brain-computer interfaces, HPC, and growing vegetables.



Jing Zhou received the Ph.D. degree in electrical engineering in 2014 from Clemson University. She held the Postdoctoral Scholar position at Stanford University. Currently she is working in industry. Her research interests include machine intelligence, signal processing and high performance computing.

José Cano received the MS and PhD degrees in computer science from the Polytechnic University of Valencia (UPV), Valencia, Spain, in 2004 and 2012, respectively. He is currently an assistant professor with the School of Computing Science, University of Glasgow, Glasgow, U.K, where he leads the Intelligent Computing Lab. He was a postdoctoral researcher with the Polytechnic University of Catalonia (UPC) between 2012 and 2013, and with the University of Edinburgh between 2014 and 2018. He has authored over 30 refereed publications. His research interest focuses on the intersection between computing systems and machine learning. He has served as a co-organizer, as a chair (publicity and session), and as a TPC in numerous conferences and workshops. He is a member of the IEEE TPDS Review Board, and the ACM TACO distinguished reviewers Board. He is member of ACM.

José L. Abellán received the BS, MS, and PhD degrees in computer science and engineering from the University of Murcia, Murcia, Spain, in 2007, 2008, and 2012, respectively. He is currently an associate professor with the Polytechnic School, Catholic University of Murcia (UCAM), Murcia, Spain. Prior to joining UCAM in 2014, in the summer of 2011, he was a predoctoral researcher at the University of Ferrara, Ferrara, Italy. From 2012 to 2014, he was a postdoctoral researcher with the ICSG research laboratory, Boston University, Boston, MA, USA. He has authored over 50 refereed publications and 1 book. He has served as a TPC, as a Chair (program, publicity, session, and track chair), and as a co-organizer in numerous conferences and workshops. He is a member of the HiPEAC, IEEE, ACM, and CAPA-H networks, and a member of the ACM TACO distinguished reviewers Board. He was the recipient of a HiPEAC 2011 collaboration grant, the best paper award at IPDPS 2011, and the HiPEAC 2019 and 2020 paper awards. His research interests include HW/SW co-design for energy-efficient HPC and edge computing.

David Kaeli received a BS and PhD in Electrical Engineering from Rutgers University, and an MS in Computer Engineering from Syracuse University. He is presently a COE Distinguished Processor on the ECE faculty at Northeastern University, Boston, MA where he directs the Northeastern University Computer Architecture Research Laboratory (NUCAR). Prior to joining Northeastern in 1993, Kaeli spent 12 years at IBM, the last 7 at T.J. Watson Research Center, Yorktown Heights, NY. Dr. Kaeli is a Fellow of the IEEE and a Distinguished Scientist of the ACM. In 1996, he received the NSF CAREER Award.