SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU

Shui Jiang
The Chinese University of Hong Kong
Hong Kong
sjiang22@cse.cuhk.edu.hk

Tsung-Wei Huang
The University of Wisconsin at Madison
USA
tsung-wei.huang@wisc.edu

Bei Yu
The Chinese University of Hong Kong
Hong Kong
byu@cse.cuhk.edu.hk

Tsung-Yi Ho
The Chinese University of Hong Kong
Hong Kong
tyho@cse.cuhk.edu.hk

ABSTRACT

Sparse deep neural network (DNN) has become an important technique for reducing the inference cost of large DNNs. However, computing large sparse DNNs is very challenging because inference iterations can incur highly irregular patterns and unbalanced loads. To address this challenge, the recent HPEC Graph Challenge seeks novel high-performance inference methods for large sparse DNNs. Despite the rapid progress over the past four years, solutions have largely focused on static model compression or sparse multiplication kernels, while ignoring dynamic data compression at inference time which can achieve significant yet untapped performance benefits. Consequently, we propose `SNICIT`, a new GPU algorithm to accelerate large sparse DNN inference via compression at inference time. `SNICIT` leverages data clustering to transform intermediate results into a sparser representation that largely reduces computation over inference iterations. Evaluated on both HPEC Graph Challenge benchmarks and conventional DNNs (`MNIST`, `CIFAR-10`), `SNICIT` achieves $6 \sim 444\times$ and $1.36 \sim 1.95\times$ speed-ups over the previous champions, respectively.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**; **Machine learning**.

KEYWORDS

Sparse Deep Neural Networks, Data Compression, GPU

ACM Reference Format:
Shui Jiang, Tsung-Wei Huang, Bei Yu, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *52nd International Conference on Parallel Processing (ICPP 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605573.3605625

1 INTRODUCTION

Deep neural networks (DNNs) have achieved unprecedented success in many applications, such as image classification [15, 37], speech recognition [8], and drug discovery [27]. To learn more complex patterns, modern DNNs are evolving toward the use of larger and deeper layers of parameters. For example, the natural language processing (NLP) model GPT-3 [5] developed by OpenAI contains 175 billion parameters. To mitigate the high computation costs of deploying large DNNs (i.e., inference), researchers have proposed various techniques (pruning [6, 13, 14, 40], sparse training [9, 21, 43], etc.) that generate *sparsified* models with fewer parameters while retaining decent accuracy. However, computing large sparse DNNs is very challenging because it can incur highly irregular patterns and unbalanced load over inference iterations (i.e., forward propagation). To tackle this challenge, MIT and Amazon co-organized the Sparse DNN Graph Challenge (SDGC) at IEEE HPEC [22], seeking novel sparse inference solutions from the high-performance computing (HPC) community.

In the past four years, winners of SDGC have proposed various solutions to accelerate large sparse DNN inference, such as accelerator hardware [16, 20], tensor core kernels [34], task graph parallelism [17, 18, 29, 30], fine-grained optimization space exploration [38], input feature partitioning [4], and compile-time data embedding [39]. While order-of-magnitude speed-ups have been reported, solutions have largely focused on static model compression or sparse multiplication kernel algorithms. Recently, a few works [12, 25, 26, 28, 33, 40] have achieved another degree of speed-up via data compression techniques at inference time, such as exploiting activation sparsity and caching historical results. While these works mostly target dense DNN, their results have inspired us to accelerate sparse DNN inference by exploring data compression at inference time.

Figure 1 illustrates our motivation. When feeding an input batch to a sparse DNN, we discover that the intermediate results at later layers exhibit a higher degree of similarity and become more centralized. We visualize the outputs of layers 2, 4, and 8 in a batch on a 2-dimensional space using t-SNE [35] (the scatter plot in Figure 1). We can clearly see that *convergence* of intermediate results takes place at layer 8, where the cluster layout of the ten labels is highly centralized and does not change onwards. After convergence, multiplying nearly identical intermediate results with the

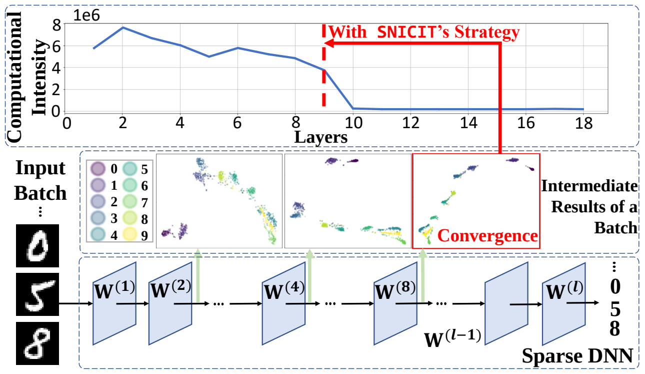


Figure 1: SNICIT explores similarity in intermediate results, and converts the representation of the intermediate results to reduce computational intensity.

weight matrix can incur many redundant computations. To mitigate this redundancy, we can turn converged intermediate results into a sparser representation. Specifically, we find the centroid of each class from intermediate results and replace every non-centroid intermediate result with deviation (i.e., residue error) to its centroid. This organization allows us to sparsify the intermediate results and thus largely reduce the computational intensity after convergence, as shown in the line chart in Figure 1.

In this paper, we propose SNICIT, an efficient GPU algorithm to accelerate sparse DNN inference using compression at inference time. We summarize our technical innovations below:

- We introduce a sparse representation for storing converged intermediate results using centroids and residue errors. This representation allows us to largely reduce redundant computations after convergence.
- We introduce an efficient GPU kernel algorithm to convert intermediate results into our sparse representation at inference time.
- We introduce a feed-forward GPU kernel to quickly update intermediate results of post-convergence layers atop our sparse representation.

We have evaluated SNICIT on the official SDGC benchmarks [22]. Compared with the 2021 champion XY-2021 [38], SNICIT is up to 6.31× faster, which translates to a 24000× speed-up over the official CPU baseline provided by SDGC. To further demonstrate the efficiency beyond SDGC, we evaluate SNICIT on four medium-scale DNNs that are representative of common deep-learning applications. Compared with the 2020 champion SNIG-2020 [30] which provides a more general solution than XY-2021, SNICIT is up to 1.83× faster with only an accuracy loss of 0.06% for sparse DNNs targeting MNIST [41], and 1.48× faster with only an accuracy loss of 0.45% for sparse DNNs targeting CIFAR-10 [2]. In this work, we focus on inference acceleration for *given* DNNs, which means that modifications on DNNs themselves are out of the scope. The source code is available at https://github.com/IDEA-CUHK/SNICIT.

2 BACKGROUND

In this section, we introduce SDGC and go through related works of DNN inference acceleration.

Table 1: Statistics of SDGC benchmarks.

Benchmarks		Bias	Density	Connections	Size
Neurons	Layers				(GB)
1024	120	−0.3	0.03	3,932,160	0.076
	480			15,728,640	0.30
	1920			62,914,560	1.22
4096	120	−0.35	0.008	15,728,640	0.328
	480			62,914,560	1.32
	1920			251,658,240	5.26
16384	120	−0.4	0.002	62,914,560	1.38
	480			251,658,240	5.54
	1920			1,006,632,960	22.17
65536	120	−0.45	0.0005	251,658,240	5.78
	480			1,006,632,960	23.12
	1920			4,026,531,840	92.48

2.1 Sparse DNN Graph Challenge

Recently, the HPC community worked with MIT and Amazon to co-organize the Sparse DNN Graph Challenge (SDGC) at IEEE HPEC [22]. The challenge established a rigorous environment to evaluate the performance of an inference method over a set of large sparse DNNs. The input matrix, $\mathbf{Y}^{(0)}$, consists of stacks of feature vectors derived from the handwritten digit dataset MNIST [41]. Each 28×28 pixel image is resized with fine granularity to 32×32 (1024 neurons), 64×64 (4096 neurons), 128×128 (16384 neurons), and 256×256 (65536 neurons), and then flattened into columns to form feature vectors purposed for different network architectures. The network architectures are generated based on Radix-Net synthetic sparse DNN generator [23]. Each neuron in all architectures has 32 edge connections with neurons in adjacent layers. The non-zero weights are set by random values, and the biases are set as constants. The detailed statistics are listed in Table 1. Participants are invited to efficiently implement the consecutive feed-forward computation of $\mathbf{Y}^{(i+1)} = \sigma(\mathbf{W}^{(i+1)} \cdot \mathbf{Y}^{(i)} + \mathbf{b}^{(i+1)})$, where $\mathbf{Y}^{(i)}$, $\mathbf{Y}^{(i+1)}$, $\mathbf{W}^{(i+1)}$ and $\mathbf{b}^{(i+1)}$ denote the input matrix, output matrix, weight matrix and bias matrix of layer i. $\sigma(\cdot)$ is ReLU [1] with an upper bound of 32, which is the activation function used in the contest.

Although SDGC only targets sparse linear layers, they are essential components of machine learning models (e.g., Transformers, graph neural networks). We shall demonstrate that our solution not only outperforms previous champions of SGDC but can also accelerate common DNN problems (MNIST and CIFAR-10) that incorporate sparse linear layers for performance improvement.

2.2 Related Works

2.2.1 Acceleration of Sparse DNN Inference. Previous SDGC winners [4, 16, 20, 30, 34, 38, 39] have made great contributions in different aspects to the acceleration of sparse DNN inference. BF-2019 [4], champion of SDGC 2019, partitions the input matrix into different sections and distributes the computation over multiple GPUs. SNIG-2020 [30], champion of SDGC 2020, further reduces synchronization and communication overheads across CPU and GPUs by utilizing GPU task graphs. XY-2021 [38], the champion of SDGC 2021, generalizes the sparse matrix multiplication kernels to a universal form and builds an optimization space. It finds the performance-optimal solution in the optimization space with the cost model they proposed. The champions [34, 39] of SDGC 2022 reorder the rows and columns of the weight matrices in a similarity-based manner offline and divide the weight matrices into dense and sparse submatrices. Sun et al. [34] compute dense and sparse submatrices with Tensor Cores and CUDA Cores respectively. Xu et al. [39] compute their self-defined cost of each submatrix and decide whether the traditional computing mode (same as `XY-2021` [38]) or data-embedding computing mode (compile-time optimization) is to be used. In addition, Huang et al. [16], SDGC 2019 honorable mention, and Jain et al. [20], SDGC 2021 innovation awards, develop FPGA accelerators, providing hardware architectural support for fast sparse DNN inference. There are also works [11, 13, 32] for sparse DNN inference acceleration outside SDGC. Guo et al. [13] and Gale et al. (`Sputnik`) [11] apply tiling-based approach in sparse matrix multiplication. Mishra et al. [32] exploit 2:4 sparsity pattern, utilizing the innovations in NVIDIA Ampere GPU architecture. Most of these works focus on static model compression or sparse matrix multiplication kernel design. Much less attention was given to dynamic data compression at inference time, which can achieve significant yet untapped performance benefits.

2.2.2 Accelerating DNNs via Data Compression at Inference Time. Some works [12, 26, 33, 40] explore *sparsities* in intermediate results at inference time. Concretely, they induce sparsity in intermediate results (or activation maps), and rely on different compression strategies to speed up the inference. `DASNet` [40] proposes a dynamic winners-take-all dropout technique to prune the none top-ranking intermediate results. Kurtz et al. [26] boost activation sparsity by applying Hoyer regularization and thresholding. They apply compressed sparse row (CSR) to compress the intermediate results and boost inference speed. Georgiadis [12] leverages a three-stage compression and acceleration pipeline that sparsifies, quantizes and entropy encodes the activation maps for faster inference. Oh et al. [33] propose to use dense representation with sparse access convolution, instead of CSR convolution. Other works [25, 28] exploit the *similarity patterns* within a DNN's intermediate results. They store the historical output of hidden layers to construct caches. A hit is reported if confidence for the similarity between a query's intermediate result and a cached result is rather high. Upon hit, the query can then obtain the pre-stored label and experience an early exit, which can yield a shorter inference latency. These works introduce nontrivial overhead proportional to the number of layers. Moreover, they have not applied data compression techniques at inference time to sparse DNNs yet.

3 ALGORITHM

Our algorithm, as shown in Figure 2, consists of four key components: pre-convergence sparse matrix multiplication, cluster-based conversion, post-convergence update, and final results recovery. We assume intermediate results *converge* at threshold layer t, where the intermediate results are highly centralized and the cluster layout does not change onwards. Term *pre-convergence* indicates the computation taking place before layer t, while *post-convergence* indicates the computation occurring after layer t.

Pre-convergence sparse matrix multiplication is simply a feed-forward process that continuously calculates the results of each pre-convergence layer. Cluster-based conversion transforms the intermediate results into a much sparser representation at inference time, which can compress the computational workload after convergence. In post-convergence update, we develop a feed-forward GPU kernel that quickly computes the intermediate results of post-convergence layers in the sparse representation. In the end, we retrieve the results from the sparse representation in the final results recovery stage. Table 2 gives essential notations along with their descriptions throughout this paper.

Table 2: Notations used in this paper.

Not.	Description of the notation
N	Number of neurons for each layer
l	Layers of the sparse DNN
B	Batch size of the input
$\mathbf{W}^{(i)}$	Weight matrix of layer i
$\mathbf{b}^{(i)}$	Bias matrix of layer i
$\mathbf{Y}^{(i)}$	Output of layer i, of size $N \times B$. Each column is an intermediate result vector.
$\mathbf{Y}^{(0)}$	Input of the sparse neural network, of size $N \times B$
σ	Activation function, $\sigma(x) = \min(\max(x, 0), 32)$
t	Index of threshold layer where convergence occurs
s	Sample size
n	Dimension of each sample
$\hat{\mathbf{Y}}^{(i)}$	Converted output of layer i, of size $N \times B$
$\mathbf{F}$	Sample matrix, of size $n \times s$, sampled from $\mathbf{Y}^{(t)}$
$\mathbf{M}$	Centroid mapper that maps a residue error column index to a centroid column index
$\mathbf{y}^*$	Set of centroid column indices
$\mathbf{A}_{:,i}$	Column i of matrix $\mathbf{A}$

3.1 Pre-convergence Sparse Matrix Multiplication

During the pre-convergence phase, we have

$$\mathbf{Y}^{(i+1)} = \sigma(\mathbf{W}^{(i+1)} \cdot \mathbf{Y}^{(i)} + \mathbf{b}^{(i+1)}), \quad (1)$$

where $i = 0, 1, ..., t-1$. The acceleration of these series of sparse matrix multiplications has already been intensively studied [4, 30, 38]. `SNICIT` does not impose any constraint on the kernel used in the pre-convergence phase. The implementation of any previous SDGC champion can be easily incorporated here.

3.2 Cluster-based Conversion

In cluster-based conversion, we convert intermediate results at layer t, $\mathbf{Y}^{(t)}$, to a much sparser form, $\hat{\mathbf{Y}}^{(t)}$. We resort to a *coarse-grained clustering algorithm* to find the *centroids* (certain columns of $\mathbf{Y}^{(t)}$) in $\mathbf{Y}^{(t)}$. We then assign the closest centroid to each non-centroid column, and replace the column with the *residue error* to the centroid, forming a new matrix, $\hat{\mathbf{Y}}^{(t)}$. Since the activation function in SDGC has both an upper bound and a lower bound, many elements of the non-centroid column and the corresponding centroid share the same value and can cancel out, making the residue error column very sparse. The computational workload of matrix multiplication in future post-convergence layers is hence reduced. Section 3.2.1 will cover the selection of centroids, while

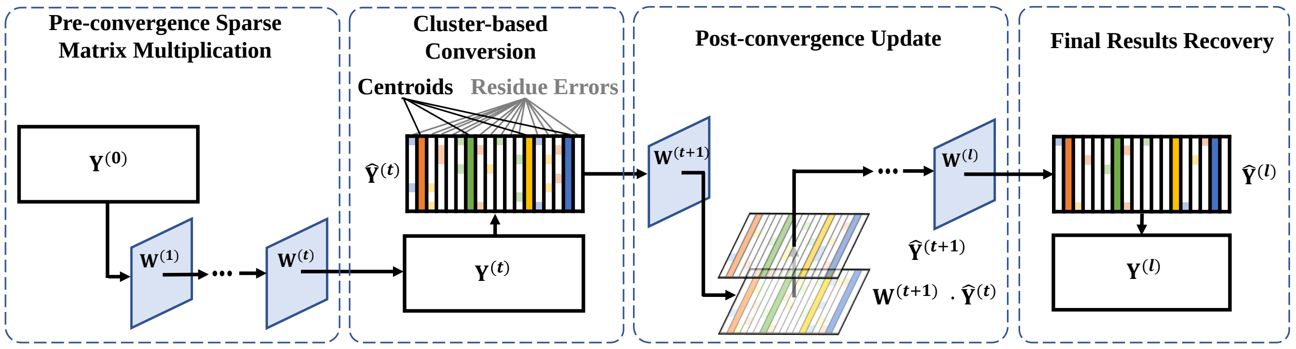


Figure 2: The overview of the entire inference computation flow. Pre-convergence sparse matrix multiplication simply calculates the outputs of pre-convergence layers. Cluster-based conversion transforms intermediate results at layer t, into a sparser representation. Different colors represent different classes in $\hat{Y}^{(i)}$. Post-convergence update computes the outputs of post-convergence layers while maintaining them in sparse representation. Final results recovery translates the sparse representation output of the last layer back to the original dense representation.

Section 3.2.2 will further explain how to obtain residue errors and the construction of $\hat{Y}^{(t)}$.

3.2.1 Centroids Selection. Fine-grained clustering algorithms, such as `K-Means` [31], `Mean Shift` [10], or even their GPU-accelerated versions [3, 42] will result in tremendous overhead in our case. These algorithms require reading all the elements in $Y^{(t)}$ and have to undergo multiple iterations before finally arriving at stable centroids. In `SNICIT`, on the other hand, we select the centroids by a coarse-grained *sampling* approach. To reduce the search space, we first apply *column sampling* to take out only a small proportion of columns from $Y^{(t)}$. Then, we use *sum downsampling* to further reduce the complexity of the samples. Finally, we narrow down the centroids by *pruning* the downsampled results.

Column Sampling. In a large-scale clustering problem, if a clustering C satisfies *strict threshold separation* (there exists a constant T such that $\|\mathbf{p}-\mathbf{q}\| \leq T, \forall \mathbf{p}, \mathbf{q} \in C_i$, and $\|\mathbf{p}-\mathbf{q}\| > T, \forall \mathbf{p} \in C_i, \forall \mathbf{q} \in C_{j\neq i}$), then it suffices for an algorithm to take $s = O(k \log \frac{k}{\delta})$ samples uniformly and randomly so that the samples contain at least one point from each cluster with a probability at least $1-\delta$, where k is the number of clusters [36].

Inspired by this property, and noticing that the columns of $Y^{(t)}$ are highly centralized, we randomly sample s columns from $Y^{(t)}$. Since the datasets (i.e., `MNIST` or `CIFAR-10`) already shuffled the inputs from different classes, we can simply take the first s columns.

Dimension Reduction with Sum Downsampling. Now, we have s columns, each containing up to 65536 elements. These elements will be constantly accessed for multiple rounds in future steps, causing a large overhead. Therefore, a dimension reduction method is needed to portray the original columns in a lightweight representation, while preserving their features. We adopt a *sum downsampling* approach. Concretely, *sum downsampling* divides the original column into n segments, each containing N/n elements, where N is the number of neurons per layer. For each segment, we run a simple sum reduction algorithm in parallel on GPU. Figure 3a depicts the combined procedure of *column sampling* and *sum downsampling*.

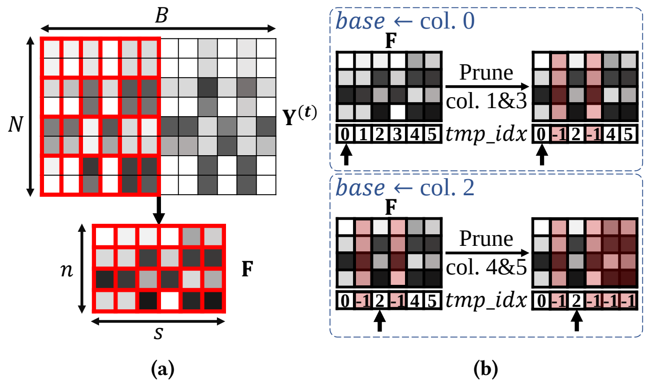


Figure 3: (a) Column sampling and sum downsampling. We take the first s columns from the original matrix of intermediate results, $Y^{(t)}$, and downsample them to matrix F. The grayscale level of each cell represents the value of the corresponding element. A red block in $\hat{Y}^{(t)}$ represents a segment to be summed up. (b) Illustration of sample pruning. (Algorithm 1)

The resulting sample matrix $\mathbf{F}$ is then generated from the original matrix $Y^{(t)}$.

Sample Pruning. To ensure that our s dimension-reduced samples in $\mathbf{F}$ cover all the k classes, as suggested in [36], s satisfies $s >> k$. This indicates that there are multiple samples in a single class. To reduce the complexity of clustering, it is necessary to *prune* the redundant samples in $\mathbf{F}$. Algorithm 1 demonstrates the GPU implementation of *sample pruning*. We launch the kernel by $<<< 1, (n, s) >>>$. Since n and s are relatively small, we only launch one block. We go through the columns of $\mathbf{F}$ iteratively. In each iteration, we set the current column as the base, and compare the remaining columns with it in parallel. Some of the remaining columns will be pruned and discarded for further comparisons if they exhibit strong similarity to the base.

At the beginning of the kernel, we construct *base* (an array for a column under comparison at each iteration), *diff* (an array to record the differences between a remaining column and *base*), and an array that records the remaining column indices after pruning, *tmp_idx*, which is initialized as the original column indices, in shared memory (lines 2-5). Then, we traverse each column (*cmp*), and as long as this column is not discarded, we set *base* to be this column (lines 6-9) and initialize *diff* with zeros (lines 10-12). The difference between *base* and another remaining column i is defined as the number of elements that have a difference larger than η, a pre-defined parameter.

$$diff[i] = \Sigma_{j=1}^{n} \operatorname{sgn}(|\mathbf{F}[i][j] - base[j]| - \eta). \tag{2}$$

All valid columns' elements are accessed in a parallel manner, and the differences between *base* and the accessed columns are summed up using atomic operations (lines 13-15). If the difference between a column and *base* is smaller than $n \cdot \epsilon$, where parameter $\epsilon < 1$, we consider the column and *base* to be in the same class and set the column's index to -1, discarding the column (lines 16-18). Finally, after all the columns are traversed, we update *col_idx* with *tmp_idx* (lines 19-20). The active ($\neq -1$) elements in the resulting *col_idx* are considered as indices of centroid columns, chosen from $\mathbf{Y}^{(t)}$. We discard all the -1s from *col_idx* and sort the remaining valid centroid indices. We then store them in $\mathbf{y}^*$, the set of centroid column indices. The iteration order (line 6) can impact selected centroids. However, at least one centroid will represent a class, and thus which column becomes which class centroid does not matter.

We illustrate Algorithm 1 with an example in Figure 3b. First, we choose column 0 as *base*, compare the rest of the columns with *base* in parallel, and discard columns with high similarities with column 0. In this case, we discard column 1 and column 3 by setting the corresponding *tmp_idx* as -1. Then, we choose the next remaining column (column 2) as *base* and conduct the comparison and pruning with the remaining columns once again. Eventually, the surviving columns (0 and 2) are the centroids chosen from $\mathbf{Y}^{(t)}$.

Algorithm 1 Kernel to prune samples

Input: *col_idx*: column indices of **F**, **F**: sample matrix, *n*: number of rows in **F**, *s*: number of columns in **F**, η: parameter, ϵ: parameter
Output: *col_idx*: updated column indices of **F**

```
1:  $tid \leftarrow$ thread.$x$ + thread.$y$ * blockDim.$x$
2:  shared $base[n]$, $diff[s]$, $tmp\_idx[s]$
3:  if thread.$x == 0$ then
4:      $tmp\_idx$[thread.$y$] $\leftarrow$ $col\_idx$[thread.$y$]
5:  __syncthreads()
6:  for $cmp \leftarrow 0$; $cmp < s$; $cmp++$ do
7:      if $tmp\_idx[cmp] \neq -1$ then
8:          if $tid < n$ then
9:              $base[tid] \leftarrow \mathbf{F}[tid][tmp\_idx[cmp]]$
10:         if $tid < s$ then
11:             $diff[tid] \leftarrow 0$
12:         __syncthreads()
13:         if $tmp\_idx$[thread.$y$] $\neq -1$ and $|\mathbf{F}$[thread.$x$][thread.$y$] $-$ $base$[thread.$x$]$| < \eta$ then
14:             atomicAdd($diff$[thread.$y$],1)
15:         __syncthreads()
16:         if thread.$x == 0$ and thread.$y \neq cmp$ and $diff$[thread.$y$] $< n\epsilon$ then
17:             $tmp\_idx$[thread.$y$] $\leftarrow -1$
18:         __syncthreads()
19: if $tid < s$ then
20:     $col\_idx[tid] \leftarrow tmp\_idx[tid]$
```

3.2.2 Residue Errors and $\hat{\mathbf{Y}}^{(t)}$. Here, we explain how to arrive at residue error columns and the construction of $\hat{\mathbf{Y}}^{(t)}$. Every non-centroid column in $\mathbf{Y}^{(t)}$ will first select the closest centroid from $\mathbf{y}^*$ (the set of centroid column indices) in terms of $L0$ norm, and store that very centroid's index in a mapper $\mathbf{M}$. For centroid columns, the corresponding values in $\mathbf{M}$ are -1. This process can be expressed as the following equation.

$$\mathbf{M}(i) = \begin{cases} \arg\min_{j \in \mathbf{y}^*} \|\mathbf{Y}^{(t)}_{:,i} - \mathbf{Y}^{(t)}_{:,j}\|_{L0}, & i \notin \mathbf{y}^*; \\ -1, & i \in \mathbf{y}^*, \end{cases} \tag{3}$$

$\mathbf{M}$ is a fixed array. It will not be modified in future computations. For non-centroid column i, $i \notin \mathbf{y}^*$ is equivalent to $\mathbf{M}(i) \neq -1$. By assigning each non-centroid column i to a certain centroid $\mathbf{M}(i)$, the non-centroid column i finds a *class* represented by the centroid $\mathbf{M}(i)$. We obtain the residue error for column i by operating vector subtraction on non-centroid column i and centroid column $\mathbf{M}(i)$. Then, we store the residue error column in $\hat{\mathbf{Y}}^{(t)}$ at ith column. The centroid columns, on the other hand, remain unchanged and are also stored in $\hat{\mathbf{Y}}^{(t)}$ at the same position as in $\mathbf{Y}^{(t)}$, as shown in Equation (4).

$$\hat{\mathbf{Y}}^{(t)}_{:,i} = \begin{cases} \mathbf{Y}^{(t)}_{:,i} - \mathbf{Y}^{(t)}_{:,\mathbf{M}(i)}, & i \notin \mathbf{y}^*; \\ \mathbf{Y}^{(t)}_{:,i}, & i \in \mathbf{y}^*. \end{cases} \tag{4}$$

Algorithm 2 describes the GPU kernel that constructs $\hat{\mathbf{Y}}^{(t)}$ and centroid mapper $\mathbf{M}$. For centroid i, the mapper value $\mathbf{M}(i)$ is initialized to be -1 before calling the kernel. The number of centroids, *cent_cnt*, is also counted beforehand. The kernel is called by $<<< (B + 1024 - 1)/1024, 1024 >>>$. 1024 consecutive columns of $\mathbf{Y}^{(t)}$ are grouped and accessed by threads in one single block. We use a shared array *cent* to store 1024 consecutive elements of a centroid in $\mathbf{y}^*$ (line 2). Then, we iterate all the centroids (line 4). In each centroid, we divide $\hat{\mathbf{Y}}^{(t)}$ columns into segments, each having 1024 elements. We iteratively inspect all the segments and record the $L0$ norm of the difference between each non-centroid column with the current centroid (lines 5-13). In the iterations, we use variable *cluster* to mark the closest centroid to each non-centroid column (lines 14-16). Next, we apply Equation (4) to **Y1**, and obtain $\hat{\mathbf{Y}}^{(t)}$ (lines 17-22). Finally, we update mapper $\mathbf{M}$ with the non-centroid column's closest centroid, and *ne_rec* with whether the residue error column is non-empty or not (lines 23-29). This algorithm is illustrated in Figure 4 from a high-level perspective.

Centroids are chosen in a one-shot fashion and are not recomputed. There is no guarantee that the centroid indices remain the same throughout post-convergence layers. However, as the main purpose of finding centroids is to convert the original representation to a sparse representation, there is no need to adjust them

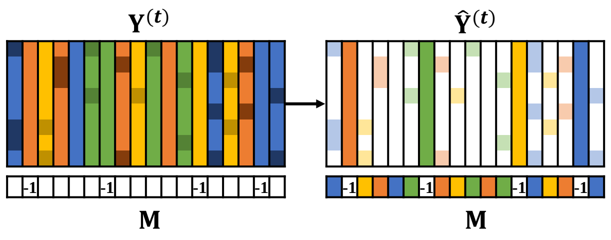


Figure 4: The construction of $\hat{\mathrm{Y}}^{(t)}$ and a complete M. Initially, we have $\mathrm{Y}^{(t)}$ and an incomplete M with only centroid indices marked as -1. Then, each non-centroid column finds the closest centroid, and updates the corresponding M entry with the centroid's index and the corresponding $\hat{\mathrm{Y}}^{(t)}$ column with residue error. Different colors represent different classes in $\hat{\mathrm{Y}}^{(t)}$.

Algorithm 2 Kernel to construct $\mathrm{Y}^{(t)}$ and **M**

Input: **Y0**: $\mathrm{Y}^{(t)}$, *cent_col*: $\mathbf{y}^*$, *cent_cnt*: number of centroids, **M**: index mapper, N: number of neurons, B: batch size

Output: **Y1**: $\hat{\mathrm{Y}}^{(t)}$, **M**: index mapper, *ne_rec*: records whether columns are non-empty in $\hat{\mathrm{Y}}^{(t)}$

```
1:  $tid \leftarrow$ thread.$x$ + thread.$y$ * blockDim.$x$
2:  shared $cent[1024]$
3:  $dist \leftarrow N + 1$
4:  for $i \leftarrow 0$; $i < cent\_cnt$; $i{+}{+}$ do
5:      $this\_dist \leftarrow 0$
6:      for $r \leftarrow 0$; $r < N/1024$; $r{+}{+}$ do
7:          $cent[$thread.$x] \leftarrow$ Y0$[1024r +$ thread.$x][cent\_col[i]]$
8:          __syncthreads()
9:          if $tid < B$ then
10:             for $k \leftarrow 0$; $k < 1024$; $k{+}{+}$ do
11:                 if $cent[k] \neq$ Y0$[1024r + k][tid]$ then
12:                     $this\_dist{+}{+}$
13:         __syncthreads()
14:     if $this\_dist < dist$ then
15:         $dist \leftarrow this\_dist$
16:         $cluster \leftarrow i$
17: for $r \leftarrow 0$; $r < N$; $r{+}{+}$ do
18:     if $tid < B$ then
19:         if M$[tid] \neq -1$ then
20:             Y1$[r][tid] \leftarrow$ Y0$[r][tid]$ − Y0$[r][cent\_col[cluster]]$
21:         else
22:             Y1$[r][tid] \leftarrow$ Y0$[r][tid]$
23: if $tid < B$ then
24:     if M$[tid] \neq -1$ then
25:         M$[tid] \leftarrow cent\_col[cluster]$
26:         if $dist == 0$ then
27:             $ne\_rec[tid] \leftarrow$ false
28:     else
29:         $ne\_rec[tid] \leftarrow$ true
```

to yet another sparse form based on new centroids. Frequently updating them can incur significant runtime overhead.

3.3 Post-convergence Update

The goal of post-convergence update is to derive $\hat{\mathrm{Y}}^{(i+1)}$ from $\hat{\mathrm{Y}}^{(i)}$ in a systematic manner with low latency, where $i = t, t+1, ..., l-1$. We summarize the general principle for post-convergence update in Equation (5). If a column in $\hat{\mathrm{Y}}^{(i)}$ is a centroid, then we do not have to consider any errors when updating to the upcoming layer. We can simply apply the standard feed-forward formula in Equation (1). If a column in $\hat{\mathrm{Y}}^{(i)}$ is a residue error column, then we have to first find the proper output of layer i, then update the column with the new residue error of layer $i+1$.

$$\hat{\mathrm{Y}}^{(i+1)}_{:,j} = \begin{cases} \sigma(\underline{\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}_{:,j}} + \mathbf{b}^{(i+1)}), & j \in \mathbf{y}^*; \\ \sigma(\underline{\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}_{:,j}} + \underline{\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}_{:,\mathbf{M}(j)}} + \mathbf{b}^{(i+1)}) \\ \quad -\sigma(\underline{\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}_{:,\mathbf{M}(j)}} + \mathbf{b}^{(i+1)}), & j \notin \mathbf{y}^*. \end{cases} \tag{5}$$

Illustrated in Figure 5, we assign two kernels in series for Equation (5) calculations. We use the first kernel to calculate the matrix multiplication of $\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}$, which corresponds to the underlined terms in Equation (5). We call this *load-reduced sparse matrix multiplication (spMM)* because $\hat{\mathrm{Y}}^{(i)}$ is a sparser matrix compared to $\mathrm{Y}^{(i)}$ after conversion, and the workload of $\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}$ is, therefore, smaller than $\mathbf{W}^{(i+1)} \cdot \mathrm{Y}^{(i)}$. A detailed description of load-reduced spMM is provided in Section 3.3.1. Then, we apply another kernel to update the centroids and residue errors in $\hat{\mathrm{Y}}^{(i+1)}$ (detailed in Section 3.3.2).

3.3.1 Load-reduced spMM. In this section, we determine a suitable spMM strategy for an efficient $\mathbf{W}^{(i+1)} \cdot \hat{\mathrm{Y}}^{(i)}$ computation. $\mathbf{W}^{(i+1)}$ is a highly sparse matrix, stored in a compressed form. $\hat{\mathrm{Y}}^{(i)}$ is also a sparse matrix, but stored in a dense format. We can adopt *sparse-sparse matrix multiplication* (spGEMM), nevertheless, there are two major issues. Firstly, $\hat{\mathrm{Y}}^{(i)}$ should be transformed into a compressed format in each layer, introducing nontrivial overhead proportional to the number of post-convergence layers. Secondly, spGEMM can be highly irregular and may exhibit low arithmetic intensity [7]. Considering that relatively dense centroid columns and relatively sparse residue error columns exist in the same matrix $\hat{\mathrm{Y}}^{(i)}$, the workload for spGEMM will be greatly imbalanced. Therefore, we resort to *sparse-dense matrix multiplication.*

We record and sort the indices of non-empty columns of $\hat{\mathrm{Y}}^{(i)}$ (True elements in *ne_rec*), and store the indices in *ne_idx*. Similar to Section 3.1, we leverage off-the-shelf kernels [4, 38] from SDGC champions for our spMM problem. However, there are two adjustments made. (1) We neglect the entirely empty columns of $\hat{\mathrm{Y}}^{(i)}$, so that memory access overhead and computational workload can be greatly reduced. As shown in Figure 5 (load-reduced spMM), we only apply the spMM kernel on non-empty columns (marked by *ne_idx*) in $\hat{\mathrm{Y}}^{(i)}$. In order to induce more empty columns in $\hat{\mathrm{Y}}^{(i)}$ and boost computation speed, we relax the condition in the construction and update of $\hat{\mathrm{Y}}^{(i)}$. After obtaining $\hat{\mathrm{Y}}^{(i)}$ columns from Equation (4) and Equation (5), we prune elements that are close to zero. (2) We only perform multiplication in the kernel, and we leave bias addition and the activation function to the next kernel.

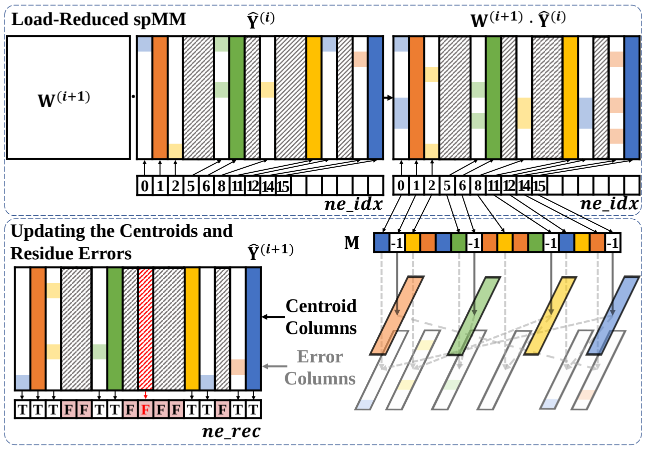

Load-Reduced spMM
$\hat{\mathbf{Y}}^{(i)}$
$\mathbf{W}^{(i+1)} \cdot \hat{\mathbf{Y}}^{(i)}$
$\mathbf{W}^{(i+1)}$
0 1 2 5 6 8 11 12 14 15
ne_idx
Updating the Centroids and Residue Errors
$\hat{\mathbf{Y}}^{(i+1)}$
M
-1
Centroid Columns
Error Columns
T T T F F T T F F F F T T F T T
ne_rec

Figure 5: We apply two kernels in post-convergence update: (1) load-reduced spMM and (2) centroids and residue errors update.

3.3.2 Centroids and Residue Errors Update. After retrieving the results of spMM, we update the centroids and residue errors for $\hat{\mathbf{Y}}^{(i+1)}$ using Equation (5). We describe the kernel in Algorithm 3, and it is called by <<< **size**(*ne_idx*), 1024 >>>. We launch **size**(*ne_idx*) blocks, each of which is associated with a non-empty column in $\mathbf{W}^{(i+1)} \cdot \hat{\mathbf{Y}}^{(i)}$. Each block contains 1024 threads. We divide the columns into $N/1024$ segments and access one segment in parallel at a time. First, we obtain the corresponding non-empty column index of a certain block (line 1). If the column is a centroid column, then we just simply update the corresponding column in $\hat{\mathbf{Y}}^{(i+1)}$, with the first case of Equation (5), and set its non-empty record *ne_rec* to `True` (lines 2-6). If the column is not a centroid column, we then update the corresponding column using the second case of Equation (5), while counting the non-zero elements in the resulting column (lines 7-11). Finally, for this non-centroid column, we update *ne_rec* with whether the column is non-empty or not (lines 12-13).

We demonstrate the idea of Algorithm 3 in Figure 5 (centroids and residue errors update). We find the indices of non-empty columns in $\mathbf{W}^{(i+1)} \cdot \hat{\mathbf{Y}}^{(i)}$ from *ne_idx*. Within these non-empty columns, we then find centroid columns and error columns by checking their $\mathbf{M}$ values. We apply different actions to the two cases based on Equation (5), and we obtain the resulting $\hat{\mathbf{Y}}^{(i+1)}$. Finally, we update *ne_rec* with whether the non-empty column is still non-empty. For example, in Figure 5, column 8 in $\hat{\mathbf{Y}}^{(i+1)}$ turns into an empty column and the corresponding *ne_rec* value is updated as `False`.

ne_rec is updated in Algorithm 3 for every layer. *ne_idx*, nevertheless, is updated outside the kernel by running serial iterations on *ne_rec*. To avoid too much serial iteration overhead, *ne_idx* can be updated less frequently than *ne_rec*. In practice, we update *ne_idx* by an interval of 200 layers for the SDGC benchmarks.

Algorithm 3 Kernel to update centroids and residue errors

Input: **Y0**: $\mathbf{W}^{(i+1)} \cdot \hat{\mathbf{Y}}^{(i)}$, **M**: centroid index mapper, *ne_idx*: non-empty columns' indices in $\hat{\mathbf{Y}}^{(i)}$, N: neuron size, b: bias (constant for each SDGC benchmark), B: batch size
Output: **Y1**: $\hat{\mathbf{Y}}^{(i+1)}$, *ne_rec*: records whether columns are non-empty in $\hat{\mathbf{Y}}^{(i+1)}$

```
1: $r \leftarrow ne\_idx[\textbf{block}.x]$
2: if $\mathbf{M}[r] == -1$ then
3:     for $j \leftarrow \textbf{thread}.x$; $j < N$; $j += \textbf{blockDim}.x$ do
4:         $\mathbf{Y1}[j][r] \leftarrow \sigma(\mathbf{Y0}[j][r] + b)$
5:     $ne\_rec[r] \leftarrow$ true
6:     return
7: $count \leftarrow 0$
8: for $j \leftarrow \textbf{thread}.x$; $j < N$; $j += \textbf{blockDim}.x$ do
9:     $v \leftarrow \sigma(\mathbf{Y0}[j][\mathbf{M}[r]] + \mathbf{Y0}[j][r] + b) - \sigma(\mathbf{Y0}[j][\mathbf{M}[r]] + b)$
10:    $count += \texttt{\_\_syncthreads\_count}(v \neq 0)$
11:    $\mathbf{Y1}[j][r] \leftarrow v$
12: if $\textbf{thread}.x == 0$ then
13:    $ne\_rec[r] \leftarrow (count \neq 0)$
```

3.4 Final Results Recovery

Eventually, in the final layer, we recover the final result $\mathbf{Y}^{(l)}$ from $\hat{\mathbf{Y}}^{(l)}$. We simply reverse Equation (4).

$$\mathbf{Y}^{(l)}_{:,i} = \begin{cases} \hat{\mathbf{Y}}^{(l)}_{:,i} + \hat{\mathbf{Y}}^{(l)}_{:,\mathbf{M}(i)}, & i \notin \mathbf{y}^*; \\ \hat{\mathbf{Y}}^{(l)}_{:,i}, & i \in \mathbf{y}^*. \end{cases} \tag{6}$$

Following this guideline, we easily restore the final result $\mathbf{Y}^{(l)}$ by reversing the generation of centroids and residue errors in a parallel manner.

4 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of `SNICIT` on two fronts: (1) The official SDGC benchmarks at a large scale and (2) Four medium-scale sparse DNNs targeting `MNIST` [41] and `CIFAR-10` [2] workloads. The purpose of the second experiment is to demonstrate that `SNICIT` can scale beyond HPEC SDGC benchmarks and accelerate medium-scale sparse DNNs that are representative of common deep-learning applications. We ran our experiments on a CentOS 8 x86 64-bit machine, with 8 Intel i7-11700 CPU cores at 2.5 GHz, one RTX A6000 48 GB GPU, and 128 GB RAM. The programs are compiled with Nvidia CUDA nvcc with optimization flag `-O3` enabled.

4.1 SDGC Benchmarks

We first evaluate `SNICIT` by comparing our performance with the latest three open-source champions of SDGC. Next, we study the runtime breakdown. Notice that pre-convergence sparse matrix multiplication has identical performance as `XY-2021`, and cluster-based conversion and final results recovery introduce a certain overhead. Therefore, the overall acceleration in `SNICIT` comes from *post-convergence update*. We will study this efficiency in depth by presenting `SNICIT`'s reduction of latency in post-convergence layers. Finally, we will analyze the impact of threshold layer t and batch size B on runtime. In the following experiments, we set $s = 32$,

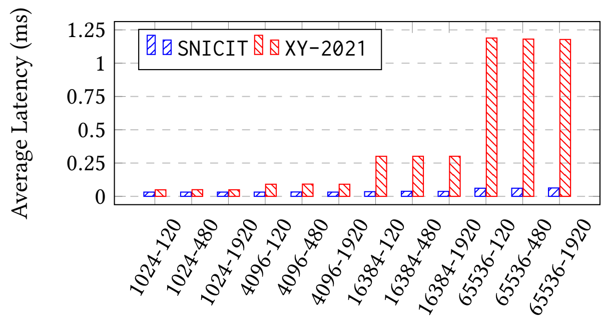


Figure 6: Comparing SNICIT's average latency per post-convergence layer with that of XY-2021 [38]. The x-axis indicates the SDGC benchmarks.

$t = 30$, $n = 16$, $\epsilon = \eta = 0.03$. We use the spMM kernel from XY-2021 for spMM tasks in pre-convergence and post-convergence stages. We use the notation, N-l, to represent the DNN benchmark of l layers each of N neurons.

4.1.1 Runtime Comparison With Previous Years' Champions. We consider the champions of SDGC 2019 (BF-2019 [4]), 2020 (SNIG-2020 [30]), and 2021 (XY-2021 [38]) as our baselines. We do not consider the 2022 champions [34, 39] because their code is not open-source. However, when we look at the runtime results of [34, 39], their speed-ups over XY-2021 [38] are not as good as SNICIT (e.g., 2.46× by [34], 1.61× by [39], and 6.31× by SNICIT for the largest SDGC benchmark). We configure the batch size B as 60000 for benchmarks with neuron size smaller than 65536, and 30000 for benchmarks with neuron size equal to 65536, to ensure that no overflow occurs in GPU memory.

Table 3 shows the overall performance comparison. SNICIT outperforms all the baselines across all the benchmarks. Our speed-up increases as the size of each benchmark (neuron size and layer depth) becomes larger. For example, at the largest benchmark of 1920 layers each of 65536 neurons, SNICIT achieves a 6.31× speed-up over XY-2021 [38]. This is because we reinterpreted post-convergence layer i's result, $Y^{(i)}$, in a sparser form, $\hat{Y}^{(i)}$, leading to a large computation reduction when propagating results over layers. Since deeper DNN has more post-convergence layers, the reduction becomes more remarkable.

4.1.2 Latency Reduction in Post-convergence Layers. Figure 6 compares the average latency per post-convergence layer (layer $t \sim l$) between SNICIT and XY-2021 [38] on the 12 official SDGC benchmarks. We can see the average latency of SNICIT is much faster than XY-2021, and the difference becomes larger when the DNN size increases. For example, for benchmark 65536-1920, the latency reduction in post-convergence layers is up to 18.69×.

4.1.3 Runtime Breakdown. Figure 7 breaks down the runtime of the entire inference workload for four DNNs (1024-120, 4096-120, 16384-120, and 65536-120). When the number of neurons enlarges, pre-convergence latency will dominate to a larger extent. In terms of overhead, the final results recovery latency is negligible (0.25% ∼ 0.43%) on all benchmarks, while the cluster-based conversion latency grows with neuron size (9.65% ∼ 16.92%).

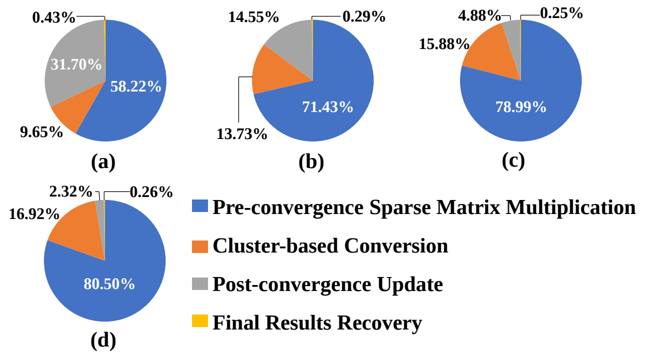


Figure 7: Runtime breakdown of four DNNs: (a) 1024-120, (b) 4096-120, (c) 16384-120, and (d) 65536-120.

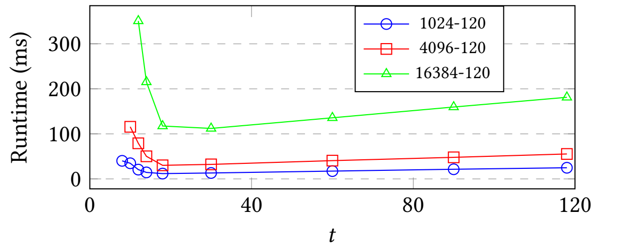


Figure 8: Runtime vs threshold layer t on three DNNs: 1024-120, 4096-120, and 16384-120.

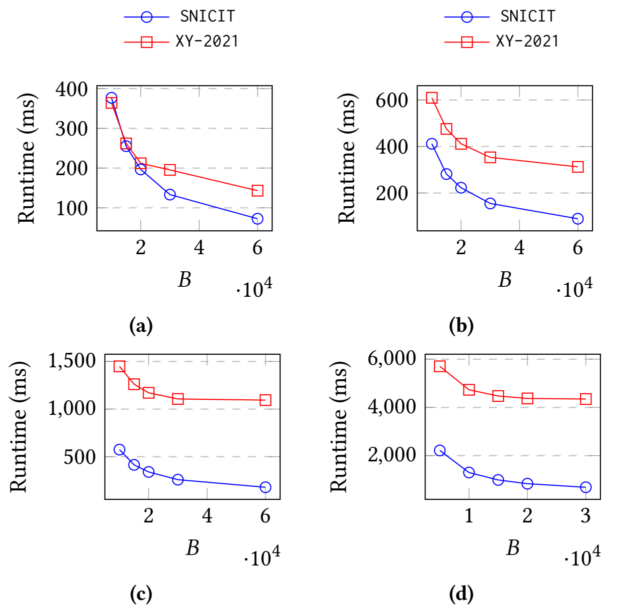


Figure 9: Runtime vs batch size B on four DNNs: (a) 1024-1920, (b) 4096-1920, (c) 16384-1920, and (d) 65536-1920.

4.1.4 Impact of Threshold Layer t and Batch Size B. We study the runtime of SNICIT at different threshold layers (i.e., t) on three DNNs (1024-120, 4096-120, and 16384-120). As shown in Figure 8,

Table 3: Overall runtime comparison of SNICIT and the previous years' champions (XY-2021 [38], SNIG-2020 [30] and BF-2019 [4]) on HPEC Sparse DNN Graph Challenge (SDGC) benchmarks [22]. All the results match the golden reference provided by the SDGC evaluation platform.

Benchmarks		SNICIT (Ours)	XY-2021 [38]		SNIG-2020 [30]		BF-2019 [4]	
Neuron	Layer	Runtime (ms)	Runtime (ms)	Speed-up	Runtime (ms)	Speed-up	Runtime (ms)	Speed-up
1024	120	13.40	14.93	1.11×	242	18.06×	498	37.16×
	480	25.10	40.93	1.63×	835	33.27×	1496	59.60×
	1920	72.52	143.15	1.97×	3203	44.17×	5464	75.34×
4096	120	32.39	39.06	1.20×	731	22.57×	1792	55.32×
	480	44.17	93.89	2.12×	2464	55.78×	5387	121.96×
	1920	89.30	312.32	3.51×	9407	105.34×	19750	221.16×
16384	120	112.09	142.91	1.27×	2523	22.51×	6687	59.66×
	480	125.82	332.93	2.65×	8374	66.56×	20314	161.45×
	1920	179.37	1094.11	6.10×	31656	176.48×	73528	409.92×
65536	120	462.81	559.61	1.21×	9950	21.50×	29142	62.97×
	480	506.96	1316.97	2.60×	28824	56.86×	84455	166.59×
	1920	689.18	4346.82	6.31×	104205	151.20×	305655	443.50×

SNICIT performs the best when t is between 20 and 40. When t is beyond this range, the advantage of SNICIT becomes smaller. We observe that when t is small, i.e., the intermediate output entries are not fully converged, SNICIT will cluster more centroids, which will prolong post-convergence update. When t is too large, the time spent on pre-convergence sparse matrix multiplication becomes dominant as this step is the most time-consuming (see the runtime breakdown in Figure 7).

Next, we study the impact of batch size B on runtime using the four deepest DNNs (1024-1920, 4096-1920, 16384-1920, and 65536-1920). Figure 9 plots the runtime of SNICIT and XY-2021 [38] at different batch sizes. When B increases, the speed-up of SNICIT becomes larger. This is because when the batch size becomes larger, the workload of XY-2021 increases. However, the number of centroids remains basically unchanged within the batches, and more intermediate results are represented in the sparse form. The workload of SNICIT becomes relatively smaller than XY-2021, thus the speed-up will be larger.

4.2 Beyond SDGC

We have demonstrated that SNICIT outperforms previous SDGC champions on SDGC benchmarks at a large scale. In this section, we show that SNICIT can also handle medium-scale sparse DNNs that are representative of common deep-learning applications. We apply PyTorch 1.12.1 and SparseLinear [19] toolkit to train three sparse DNNs (A, B, and C in Table 4) on the MNIST [41] dataset and one sparse DNN (D in Table 4) on the CIFAR-10 [2] dataset. Networks A, B, and C consist of a fully connected layer of size $784 \times N$ at the first layer, l sparsely connected layers each of size $N \times N$, and a fully connected output layer of size $N \times 10$. Network D consists of three consecutive series of two convolution layers + one max pooling layer combination, a fully connected layer for dimension calibration, l sparsely connected layers each of size $N \times N$, and finally a fully connected output layer of size $N \times 10$. The densities for all networks are between 50% $\sim$ 60%. We use cross-entropy as our loss function and Adam [24] as our optimization algorithm. We train the neural network for 150 epochs using a learning rate of 6×10^{-5}. The activation function $\sigma(\cdot)$ is ReLU [1] with an upper-bound of 1. In this experiment, we focus on the l sparsely connected hidden layers for the four networks and compare SNICIT with the baselines on these sparse layers, and we count our threshold layer t from the first sparsely connected layer.

Table 4: Statistics of medium-scale sparse DNNs (DS: dataset, MN: MNIST, and CF: CIFAR-10), and performance comparison between SNICIT and the previous champions (SNIG-2020 [30] and BF-2019 [4]).

ID	$N-l$	DS	DNN acc.	Acc. loss	Speed-up w.r.t.	
					SNIG-2020 [30]	BF-2019 [4]
A	128-18	MN	94.94%	0.24%	1.38×	1.58×
B	256-18	MN	96.88%	1.43%	1.83×	1.95×
C	256-12	MN	95.61%	0.06%	1.36×	1.40×
D	256-12	CF	75.86%	0.45%	1.48×	1.53×

4.2.1 Baselines and SNICIT's Implementation. We consider BF-2019 [4] and SNIG-2020 [30] as our baselines. We do not include XY-2021 [38] because its open-source implementation is hard-coded for SDGC benchmarks. For SNICIT's implementation, we follow the guideline of Section 3 with some modifications: We adopt spMM kernels from BF-2019 [4] for spMM tasks in pre-convergence and post-convergence stages. In cluster-based conversion, we do not perform sum downsampling, given the relatively small neuron size. Here, we update *ne_idx* for every layer, since l for medium-scale benchmarks is relatively small. We set t as the largest even integer no greater than $l/2$, $s = 128$, $\epsilon = \eta = 0.03$, and run SNICIT and the baselines on the testing set of MNIST or CIFAR-10, which consists of a batch of $B = 10000$ inputs.

4.2.2 Runtime and Accuracy Comparison With Baselines. Table 4 shows SNICIT's (1) inference accuracy loss (caused by pruning, elaborated in Section 3.3.1) from the initial DNN and (2) speed-ups over two baselines. The baselines do not have inference accuracy loss.

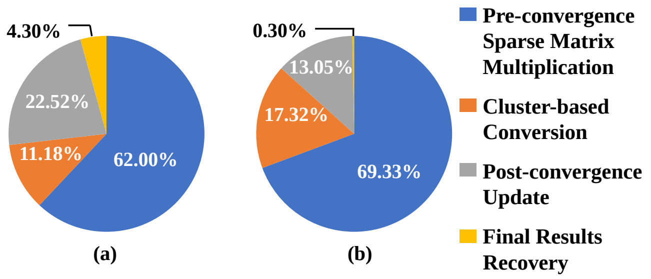


Figure 10: Runtime breakdown for medium-scale sparse DNNs: (a) DNN A, (b) DNN D.

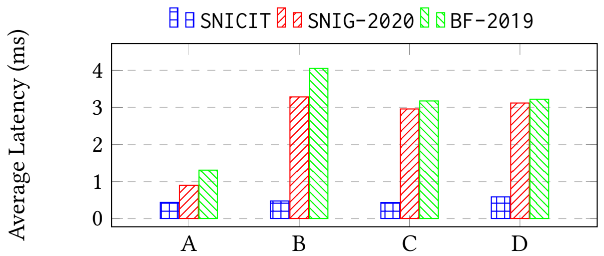


Figure 11: Comparing SNICIT's average latency per post-convergence layer with that of SNIG-2020 [30] and BF-2019 [4]. The x-axis indicates the medium-scale sparse DNNs.

SNICIT outperforms both SNIG-2020 and BF-2019 on all medium-scale sparse DNNs by up to 1.95× speed-up with negligible accuracy losses. Furthermore, we find SNICIT exhibits better performance when the medium-scale sparse DNN is deeper and larger, which is consistent with SDGC results. On the other hand, if the given DNNs are small, dense, and have many labels, the overhead of our cluster-based conversion may outweigh the advantage of post-convergence update. In this case, SNICIT may not perform well compared with other approaches.

Figure 10 shows the runtime breakdown of SNICIT on medium-scale DNNs A and D. Similar to the SDGC benchmark, pre-convergence sparse matrix multiplication takes the majority of the runtime here (62.00% and 66.70%), and recovery of final results has very little overhead (4.30% and 0.30%). Figure 11 compares the average latency across post-convergence layers between SNICIT and the baselines. SNICIT has lower average post-convergence latency than both BF-2019 and SNIG-2020 across all medium-scale sparse DNNs. Moreover, the variation of post-convergence latency for SNICIT is much smaller than that of BF-2019 and SNIG-2020 (similar to SDGC benchmarks in Figure 6). This result shows that post-convergence update exhibits good scalability across different DNNs.

4.2.3 Impact of Threshold Layer t and Batch Size B. We now study the performance of SNICIT and baselines at different combinations of threshold layer t and batch size B on the medium-scale sparse DNNs. We conduct a grid search for $t \in [0, l)$ with a step size of 2, and for $B \in \{1000, 2000, 2500, 5000, 10000\}$ on all the medium-scale sparse DNNs. We report the results in Figure 12, which includes SNICIT's speed-ups over SNIG-2020 [30] and SNICIT's accuracy loss at different (t, B) points.

From Figures 12a, 12c, 12e, and 12g, we conclude that a larger B leads to larger speed-ups. For example, when $B = 1000$, SNICIT is only faster than SNIG-2020 on DNN B. However, as B increases to 10000, SNICIT is faster than the SNIG-2020 regardless of t values. Next, when we fix B and increase t from 0 to $l-2$, it is apparent that SNICIT's speed-up over SNIG increases first, reaches the maximum at a t that is slightly smaller than $l/2$, then decreases. For example, when $B = 10000$ on benchmark B, the speed-up increases from 2.25× ($t = 0$) to 2.75× ($t = 2$), then decreases to 1.06× ($t = 16$). This feature aligns well with what we discussed in Section 4.1.4.

From Figures 12b, 12d, 12f, and 12h, we discover that B does not impact much SNICIT's inference accuracy. When t increases, SNICIT's inference accuracy loss generally decreases. However, this trend is not monotonic: (1) A larger t indicates fewer near-zero residue error prunings in $\hat{\mathbf{Y}}^{(i)}$ update, resulting in lower accuracy loss. (2) On the other hand, a smaller t yields more centroids during cluster-based conversion. More centroids can better represent a batch of intermediate results, which can also lead to fewer near-zero residue error prunings and lower accuracy loss. For example, in Figure 12d, $t = 2$ has a smaller accuracy loss compared to $t = 0$ and $t = 4$ for all B.

The best selection strategy of t and B depends on the application (e.g., neural network architectures) and GPU memory capacity. Since there is no universal optimal value, we parameterize it for applications. However, according to our experiments, using $l/2$ for t can generally achieve a good balance between accuracy loss and speed-up. Applications can further fine-tune the result by decreasing t to gain more speed-up or increasing t to gain more inference accuracy.

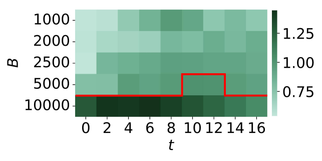
5 CONCLUSION

In this paper, we have presented SNICIT, a novel GPU algorithm to accelerate sparse DNN inference via compression at inference time. We leverage data clustering to convert intermediate results to a sparse representation that largely reduces computation over inference iterations. We have evaluated SNICIT on both the official SDGC benchmarks and medium-scale sparse DNNs. SNICIT is up to 6.31× faster than the champion of SDGC 2021 (XY-2021 [38]) on SDGC benchmarks, up to 1.83× faster than the champion of SDGC 2020 (SNIG-2020 [30]) for sparse DNNs targeting MNIST [41], and 1.48× faster than the champion of SDGC 2020 (SNIG-2020 [30]) for sparse DNNs targeting CIFAR-10 [2].

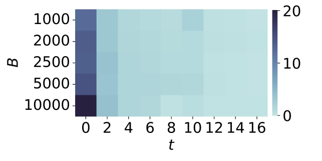
In the future, we will explore the application of SNICIT in sparse DNN training. Furthermore, we plan to develop a dynamic data-driven approach for determining threshold t.

ACKNOWLEDGMENTS

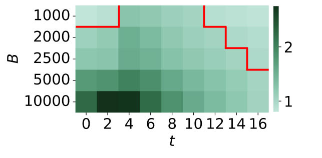
This research was partially supported by ACCESS - AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR. We are also grateful for the support of National Science Foundation of US (CCF-2126672, CCF-2144523 (CAREER), OAC-2209957, and TI-2229304) for this project.



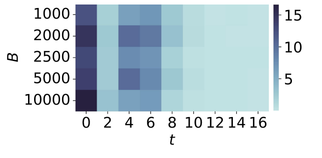
(a) SNICIT's speedup over SNIG on DNN A for different (t, B).



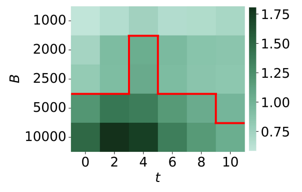
(b) SNICIT's accuracy loss on DNN A for different (t, B).



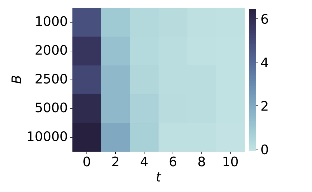
(c) SNICIT's speedup over SNIG on DNN B for different (t, B).



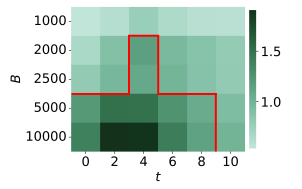
(d) SNICIT's accuracy loss on DNN B for different (t, B).



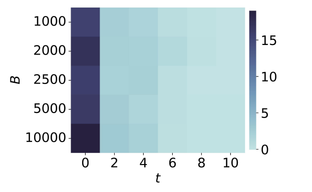
(e) SNICIT's speedup over SNIG on DNN C for different (t, B).



(f) SNICIT's accuracy loss on DNN C for different (t, B).



(g) SNICIT's speedup over SNIG on DNN D for different (t, B).



(h) SNICIT's accuracy loss on DNN D for different (t, B).

Figure 12: Impact of different (threshold t, batch size B) combinations on the performance of SNICIT. Darker pixel colors signify larger values (i.e., speed-up or accuracy loss). For (a), (c), (e), and (g), pixels under the red rectilinear lines represent points with an actual speed-up (i.e., speed-up > 1).

REFERENCES

[1] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).
[2] Vinod Nair Alex Krizhevsky and Geoffrey Hinton. [n. d.]. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html
[3] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 2015. Accelerating K-Means clustering with parallel implementations and GPU computing. In *IEEE HPEC*. 1–6.
[4] Mauro Bisson and Massimiliano Fatica. 2019. A GPU Implementation of the Sparse Deep Neural Network Graph Challenge. In *IEEE HPEC*. 1–8.
[5] Tom Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language models are few-shot learners. *NeurIPS* 33 (2020), 1877–1901.
[6] Kyusik Choi and Hoeseok Yang. 2021. A GPU Architecture Aware Fine-Grain Pruning Technique for Deep Neural Networks. In *Euro-Par*. Springer, 217–231.
[7] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing Sparse Matrix—Matrix Multiplication for the GPU. *ACM TOMS* 41, 4, Article 25 (oct 2015).
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, et al. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[9] Utku Evci, Trevor Gale, Jacob Menick, et al. 2020. Rigging the lottery: Making all tickets winners. In *ICML*. PMLR, 2943–2952.
[10] K. Fukunaga and L. Hostetler. 1975. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE TIT* 21, 1 (1975), 32–40.
[11] Trevor Gale, Matei Zaharia, Cliff Young, et al. 2020. Sparse GPU Kernels for Deep Learning. In *SC20*. 1–14.
[12] Georgios Georgiadis. 2019. Accelerating convolutional neural networks via activation map compression. In *CVPR*. 7085–7095.
[13] Cong Guo, Bo Yang Hsueh, Jingwen Leng, et al. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20*. IEEE, 1–15.
[14] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, et al. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.
[16] Sitao Huang, Carl Pearson, Rakesh Nagi, et al. 2019. Accelerating Sparse Deep Neural Networks on FPGAs. In *IEEE HPEC*. 1–7.
[17] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, et al. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 974–983.
[18] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, et al. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 6 (2022), 1303–1320.
[19] hyeon95y. 2020. SparseLinear. https://github.com/hyeon95y/SparseLinear
[20] Abhishek Kumar Jain, Sharan Kumar, Aashish Tripathi, et al. 2021. Sparse Deep Neural Network Acceleration on HBM-Enabled FPGA Platform. In *IEEE HPEC*. 1–7.
[21] Siddhant Jayakumar, Razvan Pascanu, Jack Rae, et al. 2020. Top-kast: Top-k always sparse training. *NeurIPS* 33 (2020), 20744–20754.
[22] Jeremy Kepner, Simon Alford, Vijay Gadepally, et al. 2019. Sparse Deep Neural Network Graph Challenge. In *IEEE HPEC*. 1–7.
[23] Jeremy Kepner and Ryan Robinett. 2019. Radix-net: Structured sparse matrices for deep neural networks. In *IEEE IPDPSW*. 268–274.
[24] D. P. Kingma and J. Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[25] Adarsh Kumar, Arjun Balasubramanian, Shivaram Venkataraman, et al. 2019. Accelerating deep learning inference via freezing. In *USENIX HotCloud*.
[26] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, et al. 2020. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *ICML*. PMLR, 5533–5543.
[27] Yuquan Li, Chang-Yu Hsieh, Ruiqiang Lu, et al. 2022. An adaptive graph learning method for automated molecular interactions and properties predictions. *Nature Machine Intelligence* 4, 7 (2022), 645–651.
[28] Yun Li, Chen Zhang, Shihao Han, et al. 2021. Boosting Mobile CNN Inference through Semantic Memory. In *ACM MM*. 2362–2371.
[29] D. Lin and T. Huang. 2022. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE TPDS* 33, 11 (nov 2022), 3041–3052.
[30] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE HPEC*. 1–7.
[31] J MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*. 281–297.
[32] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, et al. 2021. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021).
[33] Chanyoung Oh, Junhyuk So, Sumin Kim, et al. 2021. Exploiting Activation Sparsity for Fast CNN Inference on Mobile GPUs. *ACM TECS* 20, 5s (2021), 1–25.
[34] Yufei Sun, Long Zheng, Qinggang Wang, et al. 2022. Accelerating Sparse Deep Neural Network Inference Using GPU Tensor Cores. In *IEEE HPEC*. 1–7.
[35] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *JMLR* 9, 11 (2008).
[36] Konstantin Voevodski. 2021. Large Scale K-Median Clustering for Stable Clustering Instances. In *AISTATS*, Arindam Banerjee and Kenji Fukumizu (Eds.), Vol. 130. PMLR, 2890–2898.
[37] Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, et al. 2022. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *ICML*. PMLR, 23965–23998.
[38] Jie Xin, Xianqi Ye, Long Zheng, et al. 2021. Fast Sparse Deep Neural Network Inference with Flexible SpMM Optimization Space Exploration. In *IEEE HPEC*. 1–7.
[39] Shaoxian Xu, Minkang Wu, Long Zheng, et al. 2022. Towards Fast GPU-based Sparse DNN Inference: A Hybrid Compute Model. In *IEEE HPEC*. 1–7.
[40] Qing Yang, Jiachen Mao, Zuoguan Wang, et al. 2019. Dasnet: Dynamic activation sparsity for neural network efficiency improvement. In *IEEE ICTAI*. 1401–1405.
[41] Corinna Cortes Yann LeCun and Christopher J.C. Burges. [n. d.]. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/
[42] Le You, Han Jiang, Jinyong Hu, et al. 2022. GPU-accelerated Faster Mean Shift with euclidean distance metrics. In *IEEE COMPSAC*. 211–216.
[43] Aojun Zhou, Yukun Ma, Junnan Zhu, et al. 2021. Learning n: m fine-grained structured sparse neural networks from scratch. In *ICLR*.