GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction

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

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

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

***Abstract*—Sparse deep neural networks (DNNs) leverage sparse representations to achieve faster inference and lower memory footprint. However, deploying sparse DNNs comes with challenges, such as irregular memory access patterns, workload imbalance, etc. To address these challenges, IEEE HPEC has organized the Sparse DNN Graph Challenge (SDGC), seeking new methods from the high-performance computing community. For many years, SDGC has yielded innovative works on accelerating sparse DNN inference. However, none of them have identified redundant global memory access that contributes to significant runtime overhead. To overcome this challenge, we propose `GLARE`, a framework that can assist existing sparse inference kernels in effectively reducing redundant global memory access. We have applied `GLARE` to previous SDGC champions and a recent sparse inference engine `SNICIT`. Evaluated on SDGC benchmarks, we demonstrate the promising performance of `GLARE` and its generalizability in accelerating existing sparse inference kernels, for instance, up to 31.56× speed-up over one of the previous SDGC champions.**

***Index Terms*—Sparse Neural Network, Memory, GPU**

I. Introduction

In recent years, deep neural networks (DNNs) have witnessed remarkable accomplishments across various domains, including computer vision [1], and natural language processing [2], on diverse hardware platforms [3]–[9]. To enhance accuracy, DNNs have experienced substantial growth in both depth and width. For example, the computer vision model EfficientNet [10] has 800 layers and 66 million parameters, and the current trending ChatGPT [11] contains 96 transformer layers and 175 billion parameters. However, deploying these large-scale models poses challenges in terms of runtime and memory requirements. Thus, they are oftentimes pruned to sparse DNNs for faster inference speed and lower memory footprint.

Nonetheless, achieving efficient computation of sparse DNNs, especially when developing parallel inference algorithms on GPUs, presents many challenges. Sparse DNNs' parameters are often stored in a compressed format (e.g., CSR [12]) for space efficiency at the cost of irregular and uncoalesced memory access. Furthermore, the parameter matrices may exhibit significant variations in the number of non-zero values across different rows or columns, leading to unbalanced workloads among parallel processors. To tackle this challenge, IEEE HPEC has organized Sparse Deep Neural Network Graph Challenge (SDGC) [13], seeking innovative solutions from the high-performance computing (HPC) community. Since then, academia and industry have worked together to propose many methods [4]–[9] for SDGC. These methods focus on optimizing sparse DNN inference kernels using various techniques, such as task-graph parallelism [5], [14], [15], intra-batch similarity-based transformation [9], and so on. Despite significant performance improvement achieved by these methods, we have identified a potential to further enhance the inference speed by reducing the number of global memory access (GMA) of their kernels.

TABLE I: A memory latency example of CUDA GPU.

Memory level	**Accessibility**	**Access time**
Registers	Private to thread	1 clock cycle
Shared memory	Private to block, shared within a block	4 clock cycles
Global memory	Accessible to all threads	> 100 clock cycles

Table I outlines different memory latency values for a typical CUDA GPU. Each CUDA thread has access to its own set of registers that normally take one clock cycle per read/write access [16]. Shared memory is a fast, on-chip memory shared by multiple threads within a block. Its access latency is about 4 clock cycles [17]. Global memory is accessible by all threads in a CUDA kernel and provides large off-chip memory space. However, it has the slowest access time at a scale of hundreds of clock cycles [17]–[19]. To maximize kernel efficiency, it is important to reduce the number of read/write operations on global memory whenever possible.

In this paper, we identify the place that involves high GMA. We focus on the outputs of the intermediate layers (i.e., intermediate results) and target activation functions with upper bounds, i.e., $ReLUk(x) = \min(\max(x, 0), k)$. This type of activation functions is common: SDGC uses $ReLU32(\cdot)$, while off-the-shelf DNNs like MobileNetV2 [20] use $ReLU6(\cdot)$. Consequently, numerous blocks in the intermediate results contain consecutive elements that are all-upper-bound (AUB). We propose to compress the GMA for AUB segments. Figure 1 shows the prevalence of AUB segments in both

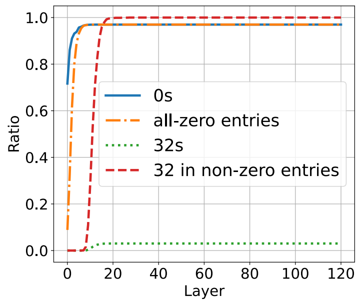
SDGC and real-world DNN applications. In Figure 1a, we present the ratios of zeros, all-zero entries (individual input vector, column or row, depending on the data matrix's shape), 32s (upper bound), and 32s in non-zero entries, of different layers on SDGC benchmark 1024-120 (i.e., with 1024 neurons per layer and 120 layers). After layer 27, the ratio of zeros and all-zero entries meet at 96.98%, while the ratio of 32s reaches 3.02% and the ratio of 32s in non-zero entries reaches 100%. In other words, all the remaining non-zero entries are *filled with 32s* (i.e., AUB segments). AUB segments widely exist in real-world DNNs as well. To demonstrate this, we train a convolutional neural network (CNN) with $ReLU1$ that achieves 98.57% test accuracy on `MNIST` dataset. Figure 1b shows a gray-scale feature map across 16 channels, with the white textures enclosed in red boxes representing numerous AUB segments within the feature map.

Given the AUB segments are prevalent in DNN computation, we aim to effectively compress their GMA. Our approach involves utilizing a single *boolean variable* to indicate whether an entire segment is classified as an AUB segment. If the variable is `true`, we save the effort of *reading* every element in the AUB segment from global memory. Furthermore, if the feed-forward result also contains AUB segments, we can omit the *writing* operation for every element within the AUB segment to global memory and just set the variable of the corresponding output AUB segment as `true`. With this important observation, we propose `GLARE`, a general framework that can accelerate sparse DNN inference kernels with significant GLobal memory Access REduction. We summarize our technical contributions below:

- We introduce a mathematical model of GMA for sparse DNN inference and propose acceleration methods by reducing GMA.
- We study the GMA patterns of existing sparse DNN inference kernels and propose an efficient matrix partitioning strategy for AUB segments of each kernel.
- We develop an algorithm to reduce redundant GMA within AUB segments. The algorithm can be applied to enhance the performance of different sparse DNN inference kernels.

We apply `GLARE` to accelerate state-of-the-art sparse inference kernels [4]–[6], [9] (incl. recent SDGC champions). We do not consider the 2022 champions [7], [8] because their code is not open-source, and their speed-ups are not as good as [9]. Evaluated on the official SDGC benchmarks, we demonstrate the promising performance of `GLARE` and its generalizability in accelerating existing sparse inference kernels, for instance, up to 31.56× speed-up over the 2019 SDGC champion. We believe `GLARE` stands out as a unique approach by accelerating sparse DNN inference from a different angle than existing methods. We have made `GLARE` open-source to facilitate high-performance machine learning research [1].

[1]GLARE source code: https://github.com/IDEA-CUHK/GLARE



(a)

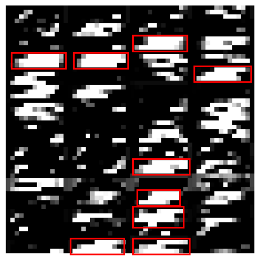

(b)

Fig. 1: Prevalence of AUB segments in DNNs with upper-bounded activation functions. (a) Ratios of 0s, all-zero entries, 32s, and 32 in non-zero entries, in intermediate results of SDGC benchmark 1024-120. (b) A reshaped feature map showing AUB segments (boxed in red) containing consecutive 1s in a CNN.

II. Background

The recent IEEE HPEC has organized SDGC to facilitate research on high-performance sparse DNN inference. SDGC establishes a rigorous environment for performing sparse DNN inference tasks. Contestants are invited to execute feed-forward computation $\mathbf{Y}_{i+1} = ReLU32(\mathbf{W}_i \cdot \mathbf{Y}_i + b)$ across l DNN layers. Here, $\mathbf{Y}_{i+1}$ and $\mathbf{Y}_i$ are the input and output *data matrices* of layer i, $\mathbf{W}_i$ is the *weight matrix* for layer i, with $0 \le i \le l-1$, and b is the bias (constant in SDGC). Since each inference iteration is carried out by a kernel, which only focuses on the input and output data matrices for one particular layer, we use $\mathbf{Y_I}$ and $\mathbf{Y_O}$ to denote the input and output data matrices of that layer. Note that in implementation, some kernels (`BF` [4] and `SNIG` [5]) adopt $\mathbf{Y}_i^T \cdot \mathbf{W}_i^T$ instead of $\mathbf{W}_i \cdot \mathbf{Y}_i$ for better memory coalescing. In these cases, we omit the transpose sign for brevity. There are 12 DNNs in total, with N neurons per layer and l layers. They are denoted as $N-l$, with $N = 1024, 4096, 16384, 65536$ and $l = 120, 480, 1920$.

III. `GLARE` Framework

In this section, we begin by overviewing the idea of `GLARE`. Subsequently, we provide formal definitions related to `GLARE`. Next, we analyze the GMA patterns of the four sparse DNN inference kernels, including SDGC champions (`BF` [4], `SNIG` [5], `XY` [6]) and `SNICIT` [9]. For each kernel, we study and select efficient data matrix partitioning strategies for AUB segments. Lastly, we present a general algorithm to reduce redundant global memory read and write for AUB segments.

A. Overview

Figure 2 provides an overview of our `GLARE` framework. During the GPU-based feed-forward computation of a layer, every CUDA block (B_{ij} in Figure 2) is assigned a segment V_{ij_I} to read from the input data matrix $\mathbf{Y_I}$, and a segment V_{ij_O} to write to the output data matrix $\mathbf{Y_O}$. As depicted in Figure 1, AUB segments can be very prevalent throughout

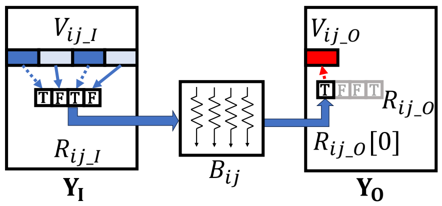


Fig. 2: An overview of GLARE. B_{ij} is a CUDA block, which reads V_{ij_I} from $\mathbf{Y_I}$ and writes V_{ij_O} to $\mathbf{Y_O}$. Blue segments represent input while red segments represent output. Dark blue/red segments in $\mathbf{Y_I}/\mathbf{Y_O}$ represent AUB while light blue segments represent non-AUB. R_{ij_I}/R_{ij_O} is used to record whether a segment in $\mathbf{Y_I}/\mathbf{Y_O}$ is AUB.

inference. In Figure 2, the first and third segments of V_{ij_I} (in dark blue) are AUB, and the other two segments of V_{ij_I} (in light blue) are not AUB. We use a four-element boolean array R_{ij_I} to track AUB segments in V_{ij_I}. $R_{ij\_I}[0] = R_{ij\_I}[2] =$ true, indicating that the first and third segments of V_{ij_I} are all AUB. Consequently, there is no need to perform global memory reads for retrieving them. Conversely, since $R_{ij\_I}[1] = R_{ij\_I}[3] =$ false, implying that the second and fourth segments of V_{ij_I} are not AUB, and thus block B_{ij} must read the corresponding segments from $\mathbf{Y_I}$ in global memory. After computation, B_{ij} finds that all the results are upper bounds (i.e., V_{ij_O} is an AUB, dark red in Figure 2). Likewise, we employ a four-element boolean array R_{ij_O} to record whether the corresponding segments are AUB. Since block B_{ij} can only output V_{ij_O}, which constitutes the leftmost quarter of an entire row in $\mathbf{Y_O}$, we use $R_{ij\_O}[0]$ to match the output (V_{ij_O}) for block B_{ij}. The remaining elements in R_{ij_O} will match the remaining segments of the row, which will be determined by other CUDA blocks. Instead of writing the entire AUB segment to global memory, we only set $R_{ij\_O}[0] =$ true.

Without writing back to global memory for synchronization, it is possible that inconsistency occurs between V_{ij_O} and $\mathbf{Y_O}$ when we claim a segment of the global memory to be AUB in R_{ij_O} without actually writing the AUB data back to $\mathbf{Y_O}$ in global memory. However, this inconsistency does not impact the result of inference. Because we apply a two-buffer scheme to R_{ij_I}/R_{ij_O} and $\mathbf{Y_I}/\mathbf{Y_O}$, in the subsequent round, the kernel will access R_{ij_I} (the previous R_{ij_O}) first. After learning that the segment in $\mathbf{Y_I}$ (the previous $\mathbf{Y_O}$) is AUB from R_{ij_I}, the kernel will not touch global memory and retrieve the inconsistent data in $\mathbf{Y_I}$. Therefore, the inconsistency does not affect reading the data at all. $\mathbf{Y_O}$ and R_{ij_O} become consistent when a round is reached where the kernel takes in an AUB segment but outputs a segment that is not AUB. In this case, $\mathbf{Y_O}$ in global memory is synchronized with R_{ij_O} to claim back consistent results. This process is similar to the write-back mechanism of caches [21].

B. Definitions

Definition 3.1. *Partition P*: A segment of the **indices** in the data matrix.

Definition 3.2. *Upper-bound universal quantifier $U(\cdot)$*: This determines whether *any* element v_i in the data matrix corresponding to index $p_i \in P$ is upper bound, i.e.,

$$U(P) := \begin{cases} 1, & \forall p_i \in P, v_i = 32; \\ 0, & \exists p_i \in P, v_i \neq 32, \end{cases} \tag{1}$$

This is used to determine whether partition P is AUB. We refer to the $U(\cdot)$ of input data matrix $\mathbf{Y_I}$ as $U_I(\cdot)$ and of output data matrix $\mathbf{Y_O}$ as $U_O(\cdot)$, respectively.

Theorem 3.1. The upper-bound universal quantifier for a *union* of multiple partitions equals the *and* of upper-bound universal quantifiers applied to those partitions, i.e.,

$$U(\bigcup_k P_k) = \bigwedge_k U(P_k) \tag{2}$$

Definition 3.3. For CUDA block B_{ij}, the *input partition* P_{ij_I} is defined as a partition that contains all the elements in the input data matrix $\mathbf{Y_I}$ that B_{ij} reads. Likewise, the *output partition* P_{ij_O} for B_{ij} is defined as a partition that contains all the elements in the output data matrix $\mathbf{Y_O}$ that B_{ij} writes. The sub-matrix that comprises elements in $\mathbf{Y_I}/\mathbf{Y_O}$ corresponding to P_{ij_I}/P_{ij_O}, is referred to as V_{ij_I}/V_{ij_O}.

Definition 3.4. *Basic partition unit $S_{i'j'}$*: The minimum unit segment of the data matrices adopted by the kernel for AUB segment identification. We use the notation $i'j'$ instead of ij, because block B_{ij} does not hold a one-to-one mapping with the basic partition unit. Among the many partition schemes for $S_{i'j'}$, we choose efficient partition schemes for different kernels (further elaborated in Section III-C). We refer to $S_{i'j'\_I}/S_{i'j'\_O}$ specifically as the basic partition unit of $\mathbf{Y_I}/\mathbf{Y_O}$.

Definition 3.5. *Recorder matrix R: R* records $U(S_{i'j'})$ for all $S_{i'j'}$ in the data matrices. R_{ij} refers to a sub-matrix of R, containing $U(S_{i'j'})$, for all $S_{i'j'}$ that B_{ij} encounters. R_{ij_I} contains $U_I(S_{i'j'\_I})$ for all $S_{i'j'\_I}$ that overlaps with P_{ij_I}, while R_{ij_O} contains $U_O(S_{i'j'\_O})$ for all $S_{i'j'\_O}$ that overlaps with P_{ij_O}.

C. Analysis of Different Kernels

In table II, we study different kernels (BF, SNIG, XY, and SNICIT) for sparse DNN inference by analyzing the global memory read and write patterns of a CUDA block B_{ij}. Then, we discuss a suitable $S_{i'j'}$ partitioning strategy for data matrices and present the GMA count expectation per layer for each kernel.

1) BF: For each round of inference iteration, B_{ij} reads a row from $\mathbf{Y_I}$, the non-zero elements from weight matrix $\mathbf{W}$, and eventually writes a row which has the same index as the input row to $\mathbf{Y_O}$. Due to limited GPU memory capacity, BF divides $\mathbf{W}$ into multiple independent vertical slabs and reads one slab from $\mathbf{W}$ at a time, hence dividing the output row into multiple segments, when the number of neurons per layer (n in Table II) is too large (e.g., 65536).

Next, to determine the appropriate $S_{i'j'}$, we study the non-zero pattern of intermediate results. We found that after only about 30 layers (varying across different DNNs), the indices of the non-zero entries stabilize. As shown in Figure 1a, the likelihood of a random entry becoming a non-zero entry is extremely low (at most 3.02%). Since BF and SNIG do not prune the all-zero entries, non-zero entries are uniformly and sparsely scattered in $\mathbf{Y_I}/\mathbf{Y_O}$. Thus, $S_{i'j'}$ must not span across multiple rows for BF and SNIG, as this can greatly increase the likelihood for $U(S_{i'j'}) = 0$. Additionally, $S_{i'j'}$ should have the minimum length of slab size (N) or n, whichever the smaller, otherwise, the highly optimized loop ordering and CUDA block partitioning schemes will undergo substantial changes, bringing potential overhead. Lastly, $S_{i'j'}$ needs to be 2^k long, being able to evenly divide n. Thus, $S_{i'j'} = P_{ij\_O}[mk : m(k+1)], k \in \{0, ..., n/m-1\}$.

Considering the constraints mentioned above, there are still multiple options for the length (m) of $S_{i'j'}$. We choose the best m by minimizing the expectation of GMA counts (E_{ij}). It is inevitable to conduct reads and writes on R_{ij_I} and R_{ij_O}, resulting in $2n/m$ counts combined. Next, we examine the counts caused by reads from $\mathbf{Y_I}$ and writes to $\mathbf{Y_O}$. Assuming every element in V_{ij_I} or V_{ij_O} has a probability of p_I or p_O to be upper bound (same in SNIG), $S_{i'j'\_I}$ or $S_{i'j'\_O}$ with m elements, has the chance of $1-p_I^m$ or $1-p_O^m$ not to be AUB. Within these n/m segments in V_{ij_I} or V_{ij_O}, the probability for k segments being AUB and the remaining segments not being AUB is $C_{n/m}^k k(1-p_I^m)^k p_I^{m(n/m-k)}$ or $C_{n/m}^k k(1-p_O^m)^k p_O^{m(n/m-k)}$. Since k ranges from 0 to n/m, the total expectation for access counts regarding $\mathbf{Y_I}$ and $\mathbf{Y_O}$ is $m\sum_{k=0}^{n/m}(C_{n/m}^k(1-p_I^m)^k \cdot p_I^{m(n/m-k)}k + C_{n/m}^k(1-p_O^m)^k \cdot p_O^{m(n/m-k)}k)$. Considering the aforementioned factors, we minimize the sum of E_{ij} (rightmost column in Table II) on all layers, s.t. $m \geq \min\{N, n\}$ and $m = 2^k$. Computation reveals that m has different values on different DNNs.

2) SNIG: B_{ij} reads a row from $\mathbf{Y_I}$, one vertical slab (of size N) from $\mathbf{W}$, and only outputs a segment of a row, whose length being $\min\{N, n\}$, to $\mathbf{Y_O}$. $S_{i'j'}$'s length m shares the same constraints as BF. We select the optimal m by minimizing the GMA count expectation.

There are n/m inputs from R_{ij_I}, and one output for R_{ij_O}. The expectation regarding P_{ij_I} is the same as BF, and the expectation regarding P_{ij_O} is the same as the writing count expectation for a non-AUB, which is $m(1-p_O^m)$. We minimize the sum of $E_{ij} = \frac{n}{m}+1+m\sum_{k=0}^{n/m}(C_{n/m}^k(1-p_I^m)^k \cdot p_I^{m(n/m-k)}k + (1-p_O^m))$ on all layers, s.t. $m \geq \min\{N, n\}$ and $m = 2^k$. After computation, we find that the best m is $\min\{N, n\}$ for all SDGC DNNs. Since P_{ij_O} has the length of $\min\{N, n\}$, we have $S_{i'j'} = P_{ij\_O}$.

3) XY: In XY and SNICIT, entries of the data matrices are columns instead of rows. XY applies multiple kernels at different layers of SDGC inference. In this work, we study the kernel XY uses after layer 22 exclusively [6]. B_{ij} takes in s columns from $\mathbf{Y_I}$ (V_{ij_I}), d rows ($d = 32$ in XY) from $\mathbf{W}$, and outputs a $d \times s$ segment (V_{ij_O}) to $\mathbf{Y_O}$. We set a $m \times d$ segment in $\mathbf{Y_I}/\mathbf{Y_O}$ to be $S_{i'j'}$. Thus, $S_{i'j'} = \bigcup_{mk/d<j<m(k+1)/d} P_{ij\_O}, k = \{0, ..., n/m-1\}$. Apparently, $m > d$, otherwise, B_{ij} will have to read n/d elements (at most 2048) from R_{ij_I}, leading to excessive global memory reads.

B_{ij} reads from R_{ij_I} n/m times. We assume that an element in $\mathbf{Y_I}$ or $\mathbf{Y_O}$ has a probability of p_I or p_O to be an upper bound. Since after a certain number of layers, the ratio of 32 in non-zero columns tends to converge to 1 as shown in Figure 1a, we consider p_I or p_O also to be the probability of a column being AUB in $\mathbf{Y_I}$ or $\mathbf{Y_O}$. As XY and SNICIT pruned the all-zero entries, p_I and p_O are very close to 1. Thus, the probability for $S_{i'j'\_I}$ or $S_{i'j'\_O}$ to be AUB is $1-p_I^s$ or $1-p_O^s$, and the expectation for memory access count regarding $\mathbf{Y_I}$ and $\mathbf{Y_O}$ is $ns(1-p_I^s) + ms(1-p_O^s)$. Also, if P_{ij_O} is not AUB (probability = $1-p_O^s$), B_{ij} writes false to R_{ij_O} (Section III-D). Ideally, according to E_{ij} (rightmost column in Table II), m should be as large as possible ($1-p_O^s \approx 0$). However, the runtime is not solely dependent on E_{ij}. Other factors like memory coalescing and memory contention affect the runtime as well. In practice, we empirically fine-tune the best m based on our GPU.

4) SNICIT: SNICIT divides the entire inference task into multiple stages and kernels. We focus on load-reduced spMM kernel in the post-convergence update stage, which involves the most layers. Compared with XY, $\mathbf{Y_I}/\mathbf{Y_O}$ in SNICIT contains fewer columns. Thus, B_{ij} reads d rows from $\mathbf{W}$, everything from $\mathbf{Y_I}$ (V_{ij_I}), and outputs only d rows to $\mathbf{Y_O}$. Similar to XY, we set a $m \times d$ segment in $\mathbf{Y_I}/\mathbf{Y_O}$ to be $S_{i'j'}$. Thus, we have $S_{i'j'} = \bigcup_{mk/d<j<m(k+1)/d} P_{ij\_O}, k = \{0, ..., n/m-1\}$. Post-convergence update takes place after the layer reaches 30, and by that time, every element in the non-zero columns is upper bound. Thus, there will not be any GMA regarding $\mathbf{Y_I}/\mathbf{Y_O}$ and R_{ij_O}. For each layer, B_{ij} will only have n/m reads from R_{ij_I}. Once again, we empirically decide the best m through experiments.

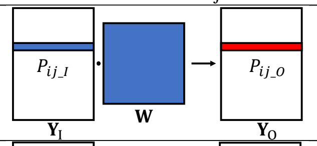
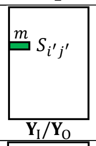
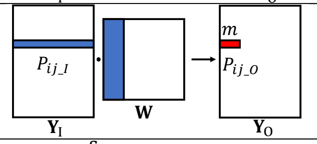
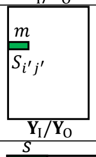
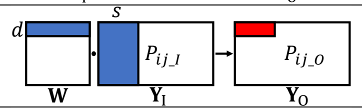
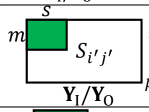
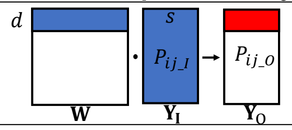
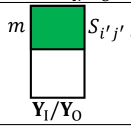
D. Global Memory Access Reduction Algorithm

Here, we provide a general algorithm to enable efficient GMA access in our GLARE framework. While applying GLARE to different kernels results in different implementations, we focus in this paper on existing SDGC kernels. Similar ideas are applicable to other kernels.

Algorithm 1 shows how a thread (x, y) in B_{ij} reads value v_{xy_I} from $\mathbf{Y_I}$. idx is the index of R_{ij_I} corresponding to v_{xy_I}. Depending on $R_{ij\_I}[idx]$'s value, we either assign 32 directly to v_{xy_I}, or load the value from $\mathbf{Y_I}$ to v_{xy_I}. R_0 is initialized as all false, but undergoes dynamic modifications during subsequent inference iterations to record the correct AUB segment information of each layer.

Algorithm 2 shows how a thread (x, y) in B_{ij} writes a value v_{xy_O} to $\mathbf{Y_O}$. If v_{xy_O} is not 32, then we have to update the corresponding element in $\mathbf{Y_O}$ (lines 1-3). idx is the index of R_{ij_O} corresponding to v_{xy_O}, while $S_{i'j'\_O}$ is the basic partition unit containing v_{xy_O}. Then, if $S_{i'j'\_O}$ is a subset of P_{ij_O} (like in BF or SNIG), we update the corresponding R_{ij_O} value with $U_1(S_{i'j'\_O})$, which is decided by each thread within $S_{i'j'\_O}$ (lines 4-5). Otherwise (like in XY or SNICIT), (x, y) set the corresponding R_{ij_O} to be

TABLE II: Different SDGC kernels with different input and output GMA. For each kernel, we show our partitioning strategy (in basic partition unit $S_{i'j'}$) for data matrices and present the GMA count expectation. Blue color represents the input segments and red represents the output segments of B_{ij}. The green-colored segment shows a basic partition unit.

Methods	Input and output GMA of B_{ij}	Basic partition unit $S_{i'j'}$	GMA count expectation E_{ij}
BF [4]	 P_{ij_I} $\mathbf{Y_I}$ $\mathbf{W}$ P_{ij_O} $\mathbf{Y_O}$	 $S_{i'j'} = P_{ij\_O}[mk{:}m(k+1)],\ k \in \{0,\ldots,\frac{n}{m}-1\}$	$2\frac{n}{m} + m\sum_{k=0}^{n/m}(C_{n/m}^{k}(1-p_I^m)^k \cdot p_I^{m(n/m-k)}k + C_{n/m}^{k}(1-p_O^m)^k \cdot p_O^{m(n/m-k)}k)$
SNIG [5]	 P_{ij_I} $\mathbf{Y_I}$ $\mathbf{W}$ m P_{ij_O} $\mathbf{Y_O}$	 $S_{i'j'} = P_{ij\_O}$	$\frac{n}{m} + 1 + m\sum_{k=0}^{n/m}(C_{n/m}^{k}(1-p_I^m)^k \cdot p_I^{m(n/m-k)}k + (1-p_O^m))$
XY [6]	 d s $\mathbf{W}$ P_{ij_I} $\mathbf{Y_I}$ P_{ij_O} $\mathbf{Y_O}$	 $S_{i'j'} = \bigcup_{\frac{m}{d}k<j<\frac{m}{d}(k+1)} P_{ij\_O},\ k = \{0,\ldots,\frac{n}{m}-1\}$	$\frac{n}{m} + ns(1-p_I^s) + (1-p_O^s)\cdot(ms+1)$
SNICIT [9]	 d s $\mathbf{W}$ P_{ij_I} $\mathbf{Y_I}$ P_{ij_O} $\mathbf{Y_O}$	 $S_{i'j'} = \bigcup_{\frac{m}{d}k<j<\frac{m}{d}(k+1)} P_{ij\_O},\ k = \{0,\ldots,\frac{n}{m}-1\}$	$\frac{n}{m}$

Algorithm 1 Reading v_{xy_I} from $\mathbf{Y_I}$ in B_{ij}

Input: R_{ij_I}: recorder sub-matrix of $\mathbf{Y_I}$, $\mathbf{Y_I}$, idx: index of R_{ij_I} corresponding to v_{xy_I}
Output: v_{xy_I}: the value fetched

```
1: if $R_{ij\_I}[idx]$ ==true then
2:     $v_{xy\_I} \leftarrow 32$
3: else
4:     $v_{xy\_I} \leftarrow \mathbf{Y_I}[i \cdot \mathbf{BlockDim.x} + x][j \cdot \mathbf{BlockDim.y} + y]$
5: end if
```

Algorithm 2 Writing v_{xy_O} to $\mathbf{Y_O}$ in B_{ij}

Input: v_{xy_O}: the value to write, idx: index of R_{ij_O}corresponding to v_{xy_O}, $S_{i'j'\_O}$: basic partition unit containing v_{xy_O}, P_{ij_O}: output partition of B_{ij}
Output: $\mathbf{Y_O}$, R_{ij_O}: recorder sub-matrix of $\mathbf{Y_O}$

```
1: if $v_{xy\_O} \neq 32$ then
2:     $\mathbf{Y_O}[i \cdot \mathbf{BlockDim.x} + x][j \cdot \mathbf{BlockDim.y} + y] \leftarrow v_{xy\_O}$
3: end if
4: if $S_{i'j'\_O} \subseteq P_{ij\_O}$ then
5:     $R_{ij\_O}[idx] \leftarrow U_O(S_{i'j'\_O})$
6: else
7:     if $v_{xy\_O} \neq 32$ then
8:         $R_{ij\_O}[idx] \leftarrow$false
9:     end if
10: end if
```

false (lines 6-10). All elements in R_{ij_O} are set to be true before Algorithm 2. According to **Theorem 3.1**, any $v_{xy\_O} \neq 32$ can cause $U_O(S_{i'j'\_O})$ =false. Therefore, even though $S_{i'j'\_O}$ spans across the output of multiple CUDA blocks, any non-upper-bound element can veto the entire $S_{i'j'\_O}$ being AUB.

IV. Experimental Results

In this section, we demonstrate the effectiveness of GLARE on four kernels, BF [4], SNIG [5], XY [6], and SNICIT [9]. BF, SNIG, and XY are the previous SGDC champions. We first measure the runtime for the four original kernels on the official SDGC benchmarks [13]. Then, we apply GLARE to each kernel and demonstrate the performance gain achieved by GLARE. All the experiments are conducted on a Ubuntu 22.04.2 LTS machine with 64 Intel Xeon Gold 6226R CPUs at 2.9 GHz and 256 GB memory capacity. The machine is equipped with an NVIDIA GeForce RTX 3090 GPU with 24 GB memory capacity. All the programs are compiled with nvcc v12.1 with -O3 flag enabled.

A. Runtime Comparison

Table III shows the runtime of the four inference kernels with and without GLARE. On average, GLARE can accelerate the four kernels by 141.73%. The performance of GLARE-accelerated BF (BF+GLARE) is faster than BF alone across all DNNs, with up to 31.56×. The reason for BF+GLARE to perform exceptionally well on DNNs of 65536 neurons is that the data for BF is too large to fit in caches. Global memory includes L1/L2 cache and DRAM [22], and GLARE saves the constant and costly communications between caches and DRAMs. For SNIG+GLARE, XY+GLARE and SNICIT+GLARE, the speed-up is up to 1.43×. However, GLARE in some DNNs is slower than the original kernels, because GLARE introduces conditional branching cost, which may outweigh its benefit.

TABLE III: Comparisons of inference time (ms) among BF, SNIG, and XY (previous SDGC champions), and SNICIT, and their enhancement after applying GLARE. Results are evaluated on SDGC benchmarks.

Benchmarks		Methods							
Neurons	Layers	BF	BF+ GLARE	SNIG	SNIG+ GLARE	XY	XY+ GLARE	SNICIT	SNICIT+ GLARE
1024	120	451.508	**431.894 (1.04×)**	209.038	**198.753 (1.05×)**	**18.2043**	20.5027 (0.89×)	**12.8492**	12.9843 (0.99×)
	480	1262.99	**1142.5 (1.10×)**	653.692	**607.657 (1.08×)**	**57.1754**	62.9377 (0.91×)	**22.468**	22.5781 (1.00×)
	1920	4564.58	**4049.81 (1.13×)**	2376.64	**2267.66 (1.05×)**	**208.593**	235.835 (0.88×)	**58.9009**	60.9954 (0.96×)
4096	120	1442.55	**1339.59 (1.08×)**	599.074	**590.782 (1.01×)**	39.3821	**37.9181 (1.04×)**	**29.2297**	29.3868 (0.99×)
	480	4229.13	**3719.68 (1.14×)**	1909.64	**1889.06 (1.01×)**	104.353	**95.0824 (1.10×)**	**38.8117**	39.3935 (0.98×)
	1920	15413.3	**13323.2 (1.16×)**	**7101.33**	7123.93 (1.00×)	348.886	**318.481 (1.10×)**	81.7058	**78.4672 (1.04×)**
16384	120	5305.51	**4965.01 (1.07×)**	2265.96	**2208.76 (1.02×)**	124.922	**114.332 (1.09×)**	97.3652	**97.2451 (1.00×)**
	480	15227.2	**13436.8 (1.13×)**	**6477.07**	6746.5 (0.96×)	300.869	**230.877 (1.30×)**	108.174	**105.142 (1.03×)**
	1920	54983.7	**48796.3 (1.13×)**	**23617.6**	25001.8 (0.94×)	994.232	**694.165 (1.43×)**	152.281	**147.089 (1.04×)**
65536	120	417769	**33912 (12.32×)**	76224.7	**74404.2 (1.02×)**	**554.5584**	571.6059 (0.97×)	738.4524	**727.829 (1.01×)**
	480	1647670	**66882.3 (24.64×)**	91935.7	**88676.8 (1.04×)**	**1427.9744**	1453.8338 (0.98×)	957.0351	**887.2877 (1.07×)**
	1920	6555300	**207741 (31.56×)**	152753	**138634 (1.10×)**	**4758.151**	4845.427 (0.98×)	1827.796	**1515.235 (1.20×)**

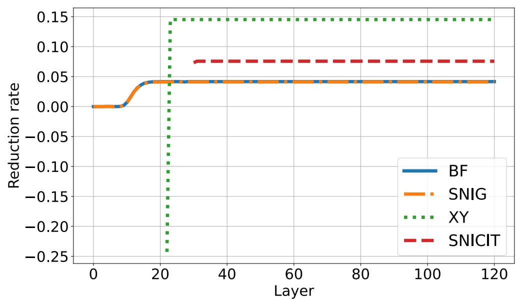


Fig. 3: GMA reduction rate with GLARE on DNN 1024-120.

B. GMA Reduction

Figure 3 illustrates the effectiveness of GLARE on reducing GMA for the four kernels on DNN 1024-120, represented by the ratio of reduced GMA access count divided by the original GMA access count. The reduction rate is below 15% because each block needs to access numerous elements of the weight matrices from the global memory as well. The curves for BF and SNIG are nearly identical, resembling the occupancy trend of AUB segments in data matrices. To be more specific, the curves remain at zero until layer 7, steadily increase, and stabilize after layer 30. Curves of XY and SNICIT begin from layers 22 and 30 respectively, as GLARE is only applied to one specific kernel, each spanning across a specific range of layers. The initial points of XY and SNICIT are lower, because R_0 is initialized to false (Section III-D), requiring more GMA. Apart from the initial points, the reduction rate for XY and SNICIT remains constant throughout the inference layers, as the number of AUB segments has little change.

V. Conclusion

In this paper, we have presented GLARE, a framework that effectively reduces redundant GMA in existing large sparse DNN inference kernels. We have introduced a mathematical model of GMA, studied the GMA patterns of existing sparse DNN inference kernels, and proposed an efficient matrix partitioning strategy for AUB segments of each kernel. Additionally, we have developed a general algorithm that can be applied to accelerate the performance of different sparse DNN inference kernels by reducing redundant GMA within AUB segments. Evaluated on the official SDGC benchmarks, GLARE has successfully accelerated all the kernels we have studied, with up to 31.56× speed-up. Future work will consider the use of new CUDA Graph [23], [24] and other task graph libraries [14], [15], [25]–[30] to gain further acceleration, as inspired by [31]–[46].

Acknowledgment

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, TI-2229304, and DMR-2235276). Special thanks go to reviewers for their time and effort on improving this paper.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] S. Jiang, S. Potluri, and T.-Y. Ho, "Scalable Scan-Chain-Based Extraction of Neural Network Models," in *IEEE/ACM DATE*, 2023, pp. 1–6.

[4] M. Bisson and M. Fatica, "A GPU Implementation of the Sparse Deep Neural Network Graph Challenge," in *IEEE HPEC*, 2019, pp. 1–8.

[5] D.-L. Lin and T.-W. Huang, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," in *IEEE HPEC*, 2020, pp. 1–7.

[6] J. Xin, X. Ye, L. Zheng, Q. Wang, Y. Huang, P. Yao, L. Yu, X. Liao, and H. Jin, "Fast Sparse Deep Neural Network Inference with Flexible SpMM Optimization Space Exploration," in *IEEE HPEC*, 2021, pp. 1–7.

[7] Y. Sun, L. Zheng, Q. Wang, X. Ye, Y. Huang, P. Yao, X. Liao, and H. Jin, "Accelerating Sparse Deep Neural Network Inference Using GPU Tensor Cores," in *IEEE HPEC*, 2022, pp. 1–7.

[8] S. Xu, M. Wu, L. Zheng, Z. Shao, X. Ye, X. Liao, and H. Jin, "Towards Fast GPU-based Sparse DNN Inference: A Hybrid Compute Model," in *IEEE HPEC*, 2022, pp. 1–7.

[9] S. Jiang, T.-W. Huang, B. Yu, and T.-Y. Ho, "SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 51–61.

[10] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[11] OpenAI, "ChatGPT," 2022. [Online]. Available: https://chat.openai.com/chat

[12] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive Sparse Tiling for Sparse Matrix Multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 300–314.

[13] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse Deep Neural Network Graph Challenge," in *IEEE HPEC*, 2019, pp. 1–7.

[14] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale," *IEEE TCAD*, vol. 40, no. 8, pp. 1687–1700, 2021.

[15] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, 2022, pp. 1303–1320.

[16] S. Mittal, "A Survey of Techniques for Architecting and Managing GPU Register File," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 16–28, 2016.

[17] Z. Yang, Y. Zhu, and Y. Pu, "Parallel Image Processing Based on CUDA," in *2008 International Conference on Computer Science and Software Engineering*, vol. 3, 2008, pp. 198–201.

[18] Y. Kim and A. Shrivastava, "CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 128–133.

[19] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 73–82.

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[22] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the Memory Hierarchy of Modern GPUs," in *Network and Parallel Computing: 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings 11*. Springer, 2014, pp. 144–156.

[23] D.-L. Lin and T.-W. Huang, "Efficient GPU Computation Using Task Graph Parallelism," in *Euro-Par*, 2021.

[24] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads," in *Euro-Par Workshop*, 2022.

[25] T.-W. Huang and L. Hwang, "Task-Parallel Programming with Constrained Parallelism," in *IEEE HPEC*, 2022, pp. 1–7.

[26] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE HPEC*, 2022, pp. 1–2.

[27] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System," *IEEE TCAD*, vol. 41, no. 5, pp. 1448–1452, 2022.

[28] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism Using Control Taskflow Graph," in *ACM HPDC*, 2022, p. 283–284.

[29] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM MM*, 2019, p. 2284–2287.

[30] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *IEEE ICPADS*, 2020, pp. 64–71.

[31] T.-W. Huang and M. D. F. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM ICCAD*, 2015, p. 895–902.

[32] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," in *IEEE TCAD*, 2021, pp. 776–789.

[33] E. Dzaka, D.-L. Lin, and T.-W. Huang, "Parallel And-Inverter Graph Simulation Using a Task-graph Computing System," in *IEEE IPDPSw*, 2023, pp. 923–929.

[34] T.-W. Huang, "qTask: Task-parallel Quantum Circuit Simulation with Incrementality," in *IEEE IPDPS*, 2023, pp. 746–756.

[35] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–12.

[36] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," in *ACM/IEEE DAC*, 2023.

[37] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[38] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[39] ——, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *ACM/IEEE DAC*, 2021, pp. 715–720.

[40] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Path-based Timing Analysis," in *IEEE/ACM DAC*, 2021, pp. 721–726.

[41] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism Using Control Taskflow Graph," in *ACM HPDC*, 2022, p. 283–284.

[42] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results," in *ACM/IEEE DAC*, 2022, p. 1388–1389.

[43] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE DAC*, 2020, pp. 1–6.

[44] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-Accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020.

[45] G. Guo, T.-W. Huang, and M. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM DATE*, 2023, pp. 1–6.

[46] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.