# G-SLIDE: A GPU-Based Sub-Linear Deep Learning Engine via LSH Sparsification

Zaifeng Pan, Feng Zhang, Hourun Li, Chenyang Zhang, Xiaoyong Du, Dong Deng

**Abstract**—Deep learning has been one of the trendiest research topics. However, as data quantities rise exponentially, training large neural networks can become prohibitively expensive with billions of parameters. Fortunately, recent research has discovered that not all of the computations in traditional network training are necessary. By selectively sparsifying the majority of the neurons during training, we can still obtain acceptable accuracy. SLIDE, a C++ OpenMP-based sub-linear deep learning engine, has been developed in this situation. SLIDE uses the algorithm of locality sensitive hashing (LSH) to query neurons with high activation in sub-linear time. It achieves a remarkable speedup in training large fully-connected networks by making use of the network sparsity as well as multi-core parallelism. However, SLIDE is limited to CPUs, ignoring the popular GPU devices with greater parallel potential and computational capability. In this paper, we propose G-SLIDE, a GPU-based sub-linear deep learning engine, which combines the benefits of SLIDE's adaptive sparsification algorithms with GPUs' high performance. The main challenges in developing G-SLIDE are efficiently using LSH to sparsify networks and training the special sparse neural networks on the GPU. To address these challenges, we propose several novel solutions, such as specific data formats and appropriate workload partitioning for threads to fully utilize the GPU resources. We evaluate G-SLIDE on two extremely sparse datasets with a 2080 Ti GPU, and the results demonstrate that for the time of one training epoch, G-SLIDE can achieve more than 16.4× speedup over SLIDE on a 32-core/64-thread CPU. Furthermore, on the same platform, G-SLIDE can earn an average of 16.2× speedup over TensorFlow-GPU and 30.8× speedup over TensorFlow-CPU.

**Index Terms**—GPU, machine learning system, adaptive sparsity, sparse neural network, LSH

## 1 Introduction

Deep learning technologies have been applied to a wide range of application scenarios in recent years. The success of neural networks relies on millions or even billions of parameters, which are driven and powered by vast amounts of data. However, deep learning models can incur high computational costs in training and inference phases as the amount of data grows. Large neural networks are therefore impractical to deploy in resource-constrained situations.

Fortunately, not all computation is necessary during neural network training. Recent studies [1], [2] show that we can achieve close accuracy by sampling only a few active neurons during every gradient update in training. However, this adaptive sparsification cannot guarantee computational savings [3], as the process of sampling can also lead to extra computation. Spring *et al.* [4] utilized a smart maximum inner product search method based on locality sensitive hashing (LSH) [5] to adaptively sparsify neurons in sub-linear time, thus showing the possibility of efficient training by adaptive sparsification.

In such a situation, Chen *et al.* [3], [6] developed a sub-linear deep learning engine (SLIDE), which extends the idea of LSH-based sparsification in [4] and first turns the computation advantage into an efficient implementation for fully connected networks. SLIDE is a C++ OpenMP-based system that utilizes multi-core parallelism to accelerate the training processes of adaptively sparse networks. Experiments [3] show that SLIDE on a 44-core CPU can outperform the TensorFlow implementation on an NVIDIA V100 GPU by a significant margin.

SLIDE provides us with a new perspective for the training of large neural networks. However, the current implementation of SLIDE is confined to multi-core CPUs, excluding GPUs, which are the most popular devices in the deep learning area. The high parallel computing capabilities and memory bandwidth of GPUs provide lots of opportunities for SLIDE acceleration. Besides, the extreme sparsity of SLIDE also makes it possible for GPUs to operate many computations directly in their on-chip cache (shared memory). Hence, we continue the work and develop an efficient GPU-based SLIDE system.

Although using GPUs to accelerate SLIDE is promising, we need to address three challenges. First, designing an efficient LSH-based sparsification algorithm on GPUs is challenging. A finer-grained parallel design of this algorithm is required to fully exploit the GPU's high parallelism. Some missing important data structures on GPUs and the memory limitations also exacerbate this problem. Second, there are no GPU-based tools that target the particular sparse computations required in SLIDE, and developing high-performance kernels on GPUs to handle this sparsity is difficult. Third, the difference between sparse and dense layers changes the features of the computing tasks, and the computing strategy needs to be re-designed accordingly to compute more efficiently.

We propose G-SLIDE, a **G**PU-based **S**ub-**LI**near **D**eep Learning **E**ngine, further improving the SLIDE system and

- *Z. Pan, F. Zhang, H. Li, C. Zhang, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and with the School of Information, Renmin University of China, Beijing 100872, China. E-mail: {panzaifeng, fengzhang,lihourun,chenyangzhang,duyong}@ruc.edu.cn.*
- *D. Deng is with the Computer Science Department, Rutgers University, The United States of America. E-mail: dong.deng@rutgers.edu (Corresponding author: Feng Zhang)*

achieving higher acceleration. G-SLIDE includes three novel characteristics. First, we analyze the memory access patterns and select appropriate data formats for various scenarios. Second, we develop an LSH sparsification module by properly partitioning the workloads and designing effective assistant data structures on GPUs. Third, we provide an efficient sparse neural network module to fully utilize the GPU resources and the benefits of the network sparsity.

We evaluate G-SLIDE on two extremely sparse real datasets, Amazon-670K and WikiLSHTC-325K, with more than 100 million parameters for training. We evaluate G-SLIDE and TensorFlow-GPU with a 2080 Ti GPU, and SLIDE and TensorFlow-CPU with a 32-core/64-thread CPU. Experiments show that for the time of one training epoch, G-SLIDE can achieve 11.9× speedup over SLIDE, 25.6× speedup over TensorFlow-GPU, and 48.9× speedup over TensorFlow-CPU on Amazon-670K. Besides, G-SLIDE can achieve 20.8× speedup over SLIDE, 6.8× speedup over TensorFlow-GPU, and 12.7× speedup over TensorFlow-CPU on WikiLSHTC-325K.

We summarize our contributions as follows.

- We analyze the SLIDE framework and find opportunities to accelerate it on the GPU to obtain better performance.
- We design efficient LSH sparsification and adaptively sparse neural network modules on the GPU with an appropriate workload partitioning strategy and well-designed data structures.
- We propose G-SLIDE, a deep learning acceleration system on the GPU. G-SLIDE accelerates deep neural networks based on SLIDE and involves many special optimizations towards the acceleration process and GPU architecture.
- We evaluate G-SLIDE on two large datasets, compare it with SLIDE and TensorFlow solutions on both CPU and GPU, and find that G-SLIDE achieves great performance benefits.

The remaining part of this paper is organized as follows. Section 2 introduces the background of the GPU, LSH, and SLIDE. Section 3 shows the motivation of G-SLIDE as well as the challenges in developing the system. We then present the detailed design of the G-SLIDE system in Section 4. The evaluation and experimental results are shown in Section 5. Sections 6 and 7 are the related work and conclusion, respectively.

## 2 Background

In this section, we introduce the background of the SLIDE deep learning system from three aspects, including graphics processing unit (GPU), locality sensitive hashing (LSH), and sub-linear deep learning engine (SLIDE).

### 2.1 Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is specialized for image processing initially. However, due to its highly-parallel computing capability, it has become an integral part of mainstream computing systems nowadays [7], [8], [9], [10].

Unlike CPUs, GPUs contain many lightweight cores, which are grouped into many streaming multiprocessors (SMs). Logically, a group of threads constitutes a *block*, and each block can only be assigned to one SM. The threads within a block share the local resources. The scheduling unit in an SM is a *warp*, which consists of 32 threads. Threads within a warp are executed in a single-instruction-multiple-threads (SIMT) fashion. Hence, the branches of threads can cause warp divergence, leading to performance degradation. Therefore, to fully utilize the parallel computing capability, appropriate partitioning of workloads is required.

The GPU has various kinds of memory resources. The *global memory* is the memory that can be accessed by all threads, equipped with L1 and L2 caches. When each thread within a warp accesses global memory, the memory transaction can be performed in the unit of 128 bytes. Hence, making the global memory accesses of threads within a warp successive can significantly reduce the overhead. Besides, an SM has its own *shared memory*, a programmable on-chip cache that can be accessed by all threads within a block. Utilization of this cache is of great importance for efficient programming on the GPU.

### 2.2 Locality Sensitive Hashing (LSH)

Locality sensitive hashing is a family of hash functions with the property that similar inputs have a higher probability of collision. A more strict condition of LSH is that the collision probability of two inputs monotonically increases with their similarity. And the property is satisfied by the majority of popular LSH functions, such as Minhash [11], Simhash [12], Winner Take All hash [13], and Densified Winner Take All hash [14]. LSH has been widely used in the nearest neighbor search [5], [15], [16].

**Winner Take All.** Winner Take All (WTA) hash was first seen in the work done by J. Yagnik *et al.* [13], which was put forward as a new measure to help solve the rank correlation measurement problem. Because the rank correlation measure problem is based on the relative ordering of elements, an efficient and useful method of presenting input features and retrieving similar ones is critical. WTA is just such a sparse embedded method that can transform the input feature space to binary codes so that Hamming distance in the resulting space can reflect the similarity of inputs.

An example of the WTA hash procedure is shown in Figure 1. Assume we have 4 input vectors $x_0, x_1, x_2$ and $x_3$ with lengths of 4, a permutation $\Theta = \{3, 0, 1, 2\}$, and a window size $K = 3$. Then, to figure out the WTA hash codes of inputs, we need to read the data in $x$ according to the permutation $\Theta$. For example, the first value of $\Theta(x_0)$ is 6, as $\Theta[0] = 3$, and $x_0[3] = 6$. After obtaining the value of $\Theta(x)$, we truncate the arrays by the value of the window size $K$, with $\Theta_K(x)$ left. The hash codes are the indices of the maximum values in $\Theta_K(x)$.

We use WTA as our LSH function, and to improve the performance and save the space, we set the window size to a constant number and then truncate the permutation $\Theta$ as we only need the first $K$ items. The number of hashing stages, therefore, shrinks to 2 accordingly.

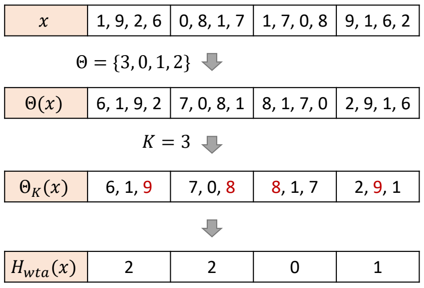


Fig. 1. An example of the Winner Take All hash procedure. The color red indicates that this element is the maximum one in the vector.

### 2.3 Sub-Linear Deep learning Engine (SLIDE)

SLIDE is a C++ OpenMP-based system proposed by Beidi Chen *et al.* [3], which employs the smart LSH algorithm [4] to sparsify the neural networks during training.

The key idea of the LSH-based sparsification algorithm in SLIDE is to select the neurons that tend to have high activations in sub-linear time. The algorithm uses two parameters, $(K, L)$, which means that a sparse layer is equipped with $L$ independent LSH tables, and each table has $K$ random hash functions. At the pre-processing phase, the LSH tables are constructed with many buckets, and each bucket contains a set of neurons. Two neurons are inserted into the same bucket only when their $K$ hash codes of the weight vector are identical. During the query phase, SLIDE computes the input's hash codes and then unions the selected $L$ buckets of these LSH tables. By this mechanism, the neurons with large inner products of weight vector and inputs (which is just the activation) are sampled, as the research [5] points out.

The workflow of SLIDE contains four stages: 1) initialization of $L$ LSH tables by neurons' weights, 2) query on the LSH tables for active neurons to feed-forward, 3) backward propagation or gradient update on the active neurons sampled in forwarding, and 4) update of the LSH tables after weight updates. Benefiting from the network sparsity, SLIDE gains lots of computation savings when training.

Besides, SLIDE utilizes OpenMP to guide the parallelism across a batch on a multi-core CPU. It also adopts the HOGWILD [17] style gradient update, as the updates are unlikely to overlap due to the extreme sparsity. Experiments show that for a huge fully-connected neural network, the training process of SLIDE on a 44-core CPU can be significantly accelerated compared to the optimized TensorFlow implementation on an NVIDIA V100 GPU with slight accuracy loss.

## 3 Motivation and Challenges

In this section, we show the motivation of G-SLIDE and the major challenges in developing the system.

### 3.1 Motivation

We introduce the motivation of G-SLIDE from three perspectives, including the importance of SLIDE, the limitations of SLIDE, and the opportunities for GPU acceleration, respectively.

**Importance of SLIDE.** Deep learning is a computing model composed of multiple processing layers, which can present and learn multiple abstract levels of input data. Various deep learning models fully empower fundamental applications such as computer vision [18], [19], [20], robotics [21], [22], data analytics [23], [24], classification problems [25], [26], natural language processing [27], and so on. Despite their wide range of applications, deep learning models have a high computational cost in training and inference. Even worse, the scale of neural networks is becoming increasingly large, with even billions of parameters to train. To handle this problem, SLIDE has been proposed, which can achieve significant computation savings and acceleration on extremely sparse datasets over traditional methods. SLIDE indicates a promising direction to address the challenges of huge networks with billions of parameters.

**Limitations of SLIDE.** The current version of SLIDE is only implemented on multi-core CPUs, without resorting to high-performance heterogeneous coprocessors like GPUs. However, although the sparsity of networks can significantly reduce the computations, the left scale of work is still too large for CPUs to handle efficiently. For example, for the dataset of Amazon-670K [28], although SLIDE only samples about 3000 neurons [3] for the last wide layer, there are still $128 \times 128 \times 3000 \times 256 = 12.6$ billion floating multiplications if the batch size is 256, even without consideration of the Softmax activation. Similarly, the number of memory accesses is also unbearable for CPUs. Besides, to update the LSH tables, SLIDE has to scan all neurons' weight vectors and figure out their buckets at the $L$ LSH tables. The complexity of this computation is about $O(nKL)$, where $n$ is the neuron number of this layer, without any saving from the sparsity.

**Opportunities for GPU acceleration.** GPUs are widely used for high-performance computing, especially in the area of deep learning. Due to the thousands of cores, GPUs can achieve a much higher degree of parallelism than CPUs. Besides, GPUs also have greater floating-point computing capability and memory access bandwidth over CPUs. There are two major opportunities to accelerate the SLIDE system on the GPU. The first one is the great potential of parallelism. SLIDE uses OpenMP to achieve coarse-grained parallelism across a batch. We can further extract the finer-grained parallelism with the cooperation of threads within a block. The second is the acceleration of both floating-point computation and memory access, which are still on a large scale even after sparsification for large networks, as previously discussed.

### 3.2 Challenges

Enabling G-SLIDE to take advantage of GPUs' high bandwidth and computing capability requires us to handle the following three challenges.

The first challenge is how to develop the LSH-based sparsification algorithm on GPUs efficiently. The algorithm can be separated into two operations: LSH table construction and active neuron query. For both the construction and query operations, we need to first compute the corresponding $K \times L$ hash codes for multiple inputs by using Winner

Take All, which contains massive parallelism opportunities. However, how to partition the process and fully utilize the computing and memory resources of GPUs is a problem that deserves consideration. Besides, in query operation, we need to gather all active neurons in the selected buckets of those $L$ hash tables after generating the hash codes. Then, we count the frequency of each occurring neuron for further sampling. SLIDE uses one linked hash table for each training sample to count the frequency, but this data structure is currently not provided by the CUDA library. Even worse, the space cost of $L$ LSH hash tables can also be a problem, as the GPU memory is limited.

The second challenge comes from the adaptive sparsity of neural networks. SLIDE queries the corresponding active neuron set from the LSH hash tables for each input, so the active neuron sets vary both inter- and intra-batch. Besides, unlike typical sparse matrix multiplication, both the input and output matrices can be sparse in SLIDE. As far as we know, no existing tool on GPUs applies to this special situation. More details of the unique features of this sparse network are discussed in Section 4.4.

The third challenge is the significant difference between sparse and dense layers, including neuron sizes and memory access patterns. A general solution to forward/backward propagation target for both from sparse layer to dense and from dense layer to sparse can introduce many issues, such as warp divergence and memory bandwidth waste, resulting in performance degradation. Hence, we should design custom kernels to adapt to the workloads and memory access patterns in different situations.

## 4 G-SLIDE System

We present the design and optimization of G-SLIDE in this section.

### 4.1 General Design

In this part, we introduce the general design of G-SLIDE, as shown in figure 2. We mainly discuss the data structure support, the LSH sparsification module, the sparse neural network module, and the workflow of G-SLIDE. We also present our solutions to the aforementioned challenges.

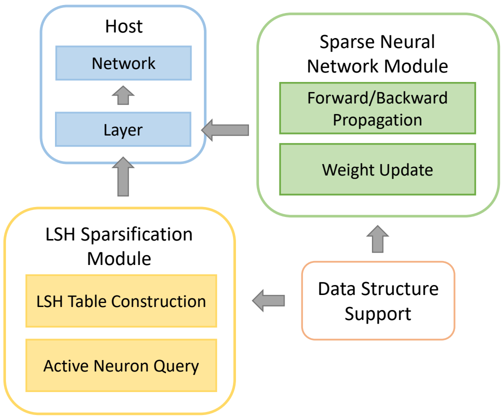


Fig. 2. An overview of the G-SLIDE system.

**Data structure support.** In developing G-SLIDE, because the neural network is extremely sparse, using ordinary data structures can significantly waste the bandwidth of the GPU, leading to performance degradation. Hence, we need to use custom data structures to address this problem. Besides, assistant data structures used in SLIDE, such as linked hash tables, are missing on the GPU, so we need to design their GPU version for usage. This individual module provides data structures, such as compressed matrices and linked hash tables, to the other modules.

**LSH sparsification module.** The LSH sparsification module performs the same functionality as in SLIDE, which contains the LSH table construction and the active neuron query operations. For the construction operation, we construct the LSH tables by the weights of the layer. For the query operation, we query the tables and retrieve the active neurons according to the input activations.

**Sparse neural network module.** Although the sparsity reduces the computation of neural network training, there are no existing tools applicable to the special structure of neural networks in G-SLIDE. We develop efficient customized kernels for forward and backward propagation and weight update in this module.

**Workflow.** An example of the G-SLIDE workflow is shown in Figure 3, which contains the following four stages.

- Initialization: We build the network with random weights and construct the LSH tables for sparse layers at this stage.
- Forward propagation: In the forward propagation, if the next layer is sparse, we query its corresponding LSH tables to obtain the active neurons, and we only compute these neurons' activations rather than all neurons in this layer.
- Backward propagation: After the forward propagation, we compare the outputs with the labels and start error backward propagation and gradient update. These operations only occur on the active neurons sampled at the forward stage.
- Weight update and LSH table reconstruction: After the backward propagation stage, we update the weights according to the gradients. The LSH table update is also required due to the weight change.

**Solutions to challenges.** In Section 3.2, we discuss the major challenges in developing an efficient GPU-based SLIDE system. In G-SLIDE, we address these challenges through our careful design. To address the first challenge of implementing an efficient LSH-based sparsification algorithm, we analyze the scale of each input dimension and distribute the memory accesses and computation workloads properly to each thread in a block. We also develop effective data structures on GPUs to support the entire process (Section 4.3). To address the second challenge of the adaptive sparsity of neural networks, we design efficient custom kernels for the training of this particular network. These kernels take advantage of the computation savings brought by the network sparsity and fully utilize the GPU resources (Section 4.4). To address the third challenge of the difference between sparse and dense layers, we discuss the memory

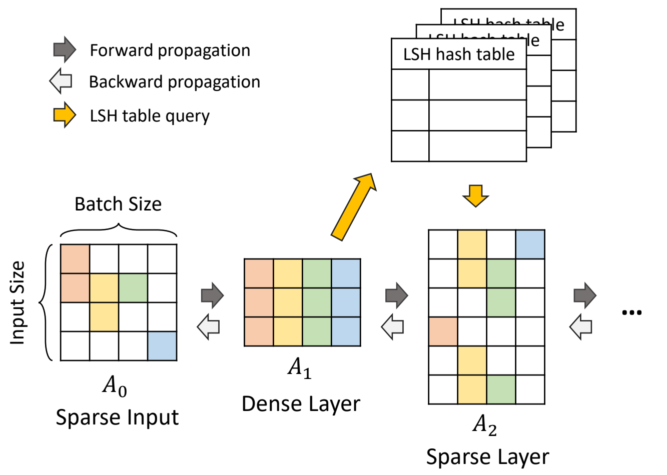


Fig. 3. The workflow of G-SLIDE. The matrices represent the activations of a batch, with the row size of the neuron number or input size and the column size of the batch size. Different colors represent active neurons of different sample inputs in the batch.

access pattern for different situations and adaptively select their proper data formats (Section 4.2).

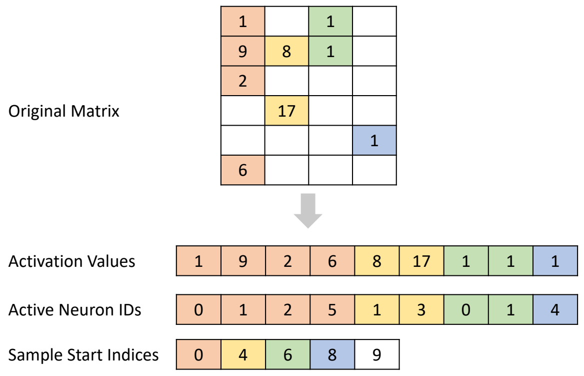
## 4.2 Data Format

In this part, we discuss the data formats of both the active neurons in sparse layers and the weight matrices.

**Compressed sparse column format.** There are only a few neurons active for each sample in the batch for sparse layers, so allocating a $\#neuron \times batch\_size$ matrix is wasteful. Compressed sparse column (CSC) is a common storage format used for sparse matrices. As illustrated in Figure 4, we represent the active neurons in a batch by three arrays, which contain the active neuron IDs, the corresponding activation values, and the sample start indices, respectively. This format not only saves the memory space, but also benefits the memory accesses. *Memory coalescing* [29] is an important concept for GPUs, which means combining multiple global memory accesses into a single transaction. Memory coalescing occurs when threads in single warp access successive 128 bytes of global memory from the address aligned to 32. Hence, the CSC format can significantly increase the chances of memory coalescing, as all data in global memory is stored successively, while the uncompressed format can introduce random memory accesses with more memory transactions, thus achieving a low utilization of global memory bandwidth. The reason why we use CSC rather than CSR (compressed sparse row) is that we tend to process the active neurons from the same sample with the same block of threads.

**Order selection of the weight matrices.** There are two typical methods for storing matrices or multidimensional arrays, which are row-major order and column-major order. A matrix is called in row-major/column-major order if the consecutive elements in a row/column are stored next to each other.

Figure 5 (a) shows an example of a dense layer next to a sparse layer, where active neurons are painted orange. As we only consider the active neurons, not all of the weights



| | | | |
|---|---|---|---|
| 1 | | 1 | |
| 9 | 8 | 1 | |
| 2 | | | |
| | 17 | | |
| | | | 1 |
| 6 | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Activation Values | 1 | 9 | 2 | 6 | 8 | 17 | 1 | 1 | 1 |
| Active Neuron IDs | 0 | 1 | 2 | 5 | 1 | 3 | 0 | 1 | 4 |
| Sample Start Indices | 0 | 4 | 6 | 8 | 9 | | | | |

Fig. 4. Illustration for the CSC format. The different colors represent active neurons from different samples in a batch.

in the weight matrix are really needed in both forward and backward propagation. The useful weights are also painted in the matrix in the figure, and accordingly, we can observe that the weights we need to read are successive in the column. Hence, a weight matrix in column-major order is preferred in this situation, as it increases the possibility for us to do memory coalescing to improve throughput. Similarly, according to figure 5 (b), row-major order is preferred for the weight matrix of a sparse layer next to a dense layer. In our test, if we do not store the weight matrix in proper order, the kernel execution time can be 20× longer.

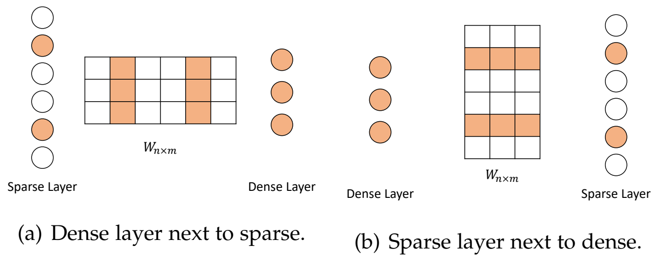


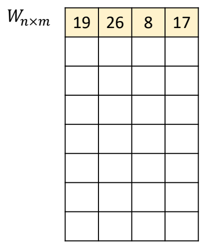
Fig. 5. The active neurons and their corresponding weights (painted orange) in different situations.

G-SLIDE maintains the order of the weight matrices during training, since whether a layer is sparse or not is determined by users before training. Only when the model needs to be saved for further training in another framework, such as TensorFlow, will the format conversions occur.

## 4.3 LSH Sparsification

In this part, we show the implementation of the LSH sparsification module with the LSH table construction and active neuron query operations. We take one layer with $L$ hash tables and $K$ hash functions of each hash table as an example in this part. In addition, we present the design of the GPU-based assistant data structures used in this module.

**LSH table construction.** For the construction operation, we need to first compute the $K \times L$ hash codes for weight vectors of each neuron by Winner Take All. Then, we insert the neuron IDs into the corresponding buckets of all these $L$ hash tables. Figure 6 shows the bucket computing process of the first row of the weight matrix (i.e., the weight vector of neuron 0 in this layer), with 2 hash tables and 2 hash functions for each table. For each random permutation $\Theta$ of the hash function, we should first read the weights in the vector according to the indices in $\Theta$ to get $\Theta(w)$. Then, we find the position of the maximum element in $\Theta(w)$, which is $H_{wta}(w)$. Finally, we concatenate the $K$ hash codes in a hash table to obtain the bucket index. The concatenation of hash codes is done by the shifting operation, and in this example, we compute the first bucket index by $(1 << 2) + 2 = 6$, as the length of each permutation is 3 and $\lceil log_2(3) \rceil = 2$.



| $\Theta$ | 0, 2, 1 | 3, 1, 2 | 1, 2, 0 | 0, 2, 3 |
|---|---|---|---|---|
| $\Theta(w)$ | 19, 8, 26 | 17, 26, 8 | 26, 8, 19 | 19, 8, 17 |
| $H_{wta}(w)$ | 2 | 1 | 0 | 0 |
| Bucket | 6 | | 0 | |

Fig. 6. An example of bucket computing in LSH table construction.

In our practice, the number of neurons in sparse layers is always huge, while the number of neurons in their previous layers tends to be small. The total length of all permutations can be large but still much less than this layer's neuron number. This observation hints that we can extract the parallelism by partitioning the computation from the dimension of neurons in this layer or the $K \times L$ hash functions.

As shown in Figure 7, we use a CUDA block responsible for the bucket index computing of a certain number of neurons with all $L$ hash tables. The tables will be divided into many tiles, and the block of threads will sequentially process these tiles (the reason for this division is discussed in the next paragraph). When the block is processing a tile of tables, each thread in the block is responsible for a pair of one neuron and several tables in the tile. The threads will scan the corresponding permutations of their responsible tables, compute the hash codes, figure out the bucket index, and finally add the neuron to the tables. Finer-grained parallelism, such as dividing the computation of one table or one permutation into multiple tiles for multiple threads, is not recommended because both the number of hash functions (or permutations) in a table and the length of a permutation are always small. The further division can cause performance degradation due to the cost of inter-thread communications.

For the bucket computing of each neuron, we need to read the indices in all permutations, so the permutation array is read multiple times. Hence, we can consider using *shared memory* to store the array. Shared memory is an on-chip memory shared by all threads within a block, and it can be regarded as a programmable cache on the GPU. By using shared memory, we can load the permutation data from global memory to shared memory only once for each block, and then read from the shared memory with low latency. The problem is that the size of shared memory on the GPU is limited (e.g., 96KB for V100). Therefore, instead of loading the entire array once, we partition the permutation array into many tiles, with each tile containing the permutations

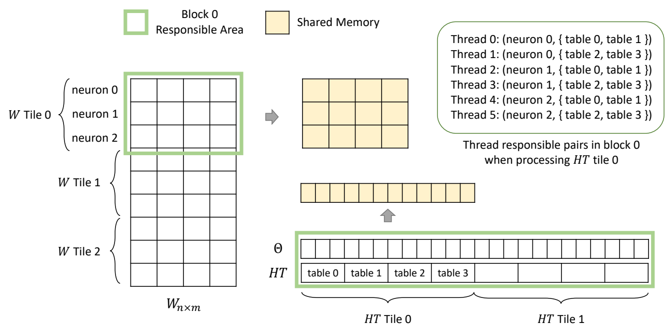

Block 0 Responsible Area
Shared Memory
Thread 0: (neuron 0, { table 0, table 1 })
Thread 1: (neuron 0, { table 2, table 3 })
Thread 2: (neuron 1, { table 0, table 1 })
Thread 3: (neuron 1, { table 2, table 3 })
Thread 4: (neuron 2, { table 0, table 1 })
Thread 5: (neuron 2, { table 2, table 3 })
Thread responsible pairs in block 0 when processing $HT$ tile 0
$W$ Tile 0
neuron 0
neuron 1
neuron 2
$W$ Tile 1
$W$ Tile 2
$W_{n \times m}$
$\Theta$
$HT$
table 0
table 1
table 2
table 3
$HT$ Tile 0
$HT$ Tile 1

Fig. 7. An example of the workload partition of LSH table construction. We divide both neurons (weights) and tables into many tiles, and block 0 is responsible for tile 0 of neurons and all tiles of tables. Block 0 processes the table tiles sequentially, and the threads' responsible pairs of the neuron and tables when processing tile 0 of tables are shown.

of several tables. Then, a block can process the tiles sequentially, and reload the tile data into the shared memory before processing the next tile, which reduces the shared memory usage to the tile size. Besides, if the number of neurons of the previous layer is small, we can also fit the weight tile of responsible neurons into the shared memory, which can not only reduce the global memory access times, but also avoid the random memory access due to the random permutation, so that the kernel can fully utilize the memory bandwidth.

Algorithm 1 describes the detailed task of the LSH table construction for one thread. $\_\_syncthreads()$ is a barrier that synchronizes all threads in a block, and we use it to ensure that threads can access the shared memory correctly. The barrier at Line 9 avoids the case that some threads access the shared memory before other threads load it from global memory. The barrier at Line 18 ensures that the shared memory is updated after all threads finish the computation of the iteration.

**Algorithm 1** LSH table construction

```
    Input: weights $W$, permutations $\Theta$
    Output: LSH tables $HT$
1:  procedure CONSTRUCTIONKERNEL
2:      $W_t \leftarrow$ getBlockTileW($W, blockIdx$)
3:      $W_{s,t} \leftarrow$ loadToSharedMem($W_t$)
4:      $n \leftarrow$ getResponsibleNeuron($threadIdx$)
5:      $W_{s,n} \leftarrow$ getNeuronW($W_{s,t}, n$)
6:      for each tile $HT_t$ in $HT$ do
7:          $\Theta_t \leftarrow$ getTblTheta($\Theta, HT_t$)
8:          $\Theta_{s,t} \leftarrow$ loadToSharedMem($\Theta_t$)
9:          __syncthreads()
10:         $HT_r \leftarrow$ getResponsibleTbl($HT_t, threadIdx$)
11:         for each $tbl$ in $HT_r$ do
12:             $\Theta_{s,tbl} \leftarrow$ getTblTheta($\Theta_{s,t}, tbl$)
13:             $bucketIdx \leftarrow 0$
14:             for each $\theta$ in $\Theta_{s,tbl}$ do
15:                 $h \leftarrow H_{wta}(W_{s,n}, \theta)$
16:                 concat($bucketIdx, h$)
17:             addToTblBucket($tbl$, $bucketIdx, n$)
18:         __syncthreads()
```

**Active neuron query.** The active neuron query procedure is illustrated in Figure 8, which takes one row in a batch as an example. The left matrix in the figure is the transpose of the activation matrix. The query operation takes the neurons' activation values of the previous layer as input, and similar to the construction operation, it computes the corresponding bucket indices in all hash tables. After getting the selected table buckets, we union all neurons in these buckets and count the frequency of those neurons for further sampling by threshold.

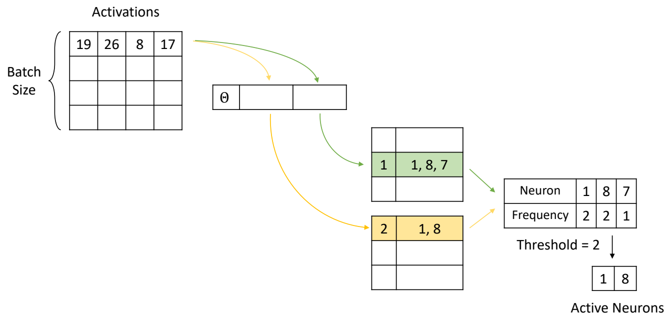


Fig. 8. An example of active neuron query.

For the bucket computing stage, we can use a similar method to that deployed in the table construction operation, while turning the weight matrix into the batch of activation values. However, the scale of batch size is always much less than the neuron number of the sparse layer. Hence, using multiple blocks to divide the batch seems to be improper. Instead, a block is utilized to handle only a tile of hash tables and compute the bucket indices for the entire batch in these tables.

For the union and sub-sampling stage, SLIDE uses the data structure of a linked hash table to count the frequency of neurons. However, this data structure is missing in Nvidia's Thrust library [30]. Note that this hash table is unlike the LSH tables, as we need to cooperate with the dependence when adding an item to the table. To deal with this problem, we develop an intra-block mutex to protect the critical section, shown in Algorithm 2, by using atomic operations [29] such as $atomicCAS$ (compare and swap) at Line 6 in CUDA. Instead of spinning to acquire the mutex, we use a barrier at Line 11 to ensure that all threads in the block are in the same iteration, and a thread failing to hold the mutex will give up to wait for the next iteration. This design avoids the possibility of deadlock, and we show an example of the incorrect spinlock in Algorithm 3. This example will lead to a deadlock because the threads spinning can occupy the resources so that threads holding the mutex cannot be scheduled.

With this intra-block mutex, we can then develop the block-level thread-safe linked hash table on the GPU. The linked hash table on the CPU needs to malloc a new list node when inserted into a bucket, but this operation is very inefficient on the GPU. Instead, we maintain a memory pool so that each time a new list node needs to be allocated, we atomically acquire it from the pool. Figure 9 shows the GPU simulation of the data structure of the linked hash table, where the green and yellow cells represent the linked lists of two buckets, respectively. The lists are linked by the

**Algorithm 2** A simple intra-block mutex

```
1: do
2:     if $threadIdx$ is 0 then
3:         $blockStop \leftarrow$ true
4:     __syncthreads()
5:     if isJobFinished($threadIdx$) then
6:         if atomicCAS($mutex, 0, 1$) is 0 then
7:             doCriticalJob($threadIdx$)
8:             atomicExch($mutex, 0$)    ▷ release the mutex
9:         else ▷ Other thread hold the mutex, wait for next iteration
10:            $blockStop \leftarrow$ false
11:    __syncthreads()
12: while not $blockStop$
```

**Algorithm 3** An **incorrect** mutex

```
1: while atomicCAS($mutex, 0, 1$) is not 0 do    ▷ This code will cause dead lock!
2:     pass
3: doCriticalJob($threadIdx$)
4: atomicExch($mutex, 0$)
```

corresponding value of each entry in the $Next$ array, which represents the position of the next entry in the memory pool. Each bucket is protected by an individual mutex, and the mutex is only acquired when we need to insert a new item into the list. To aggregate the neurons in those $L$ buckets, each thread in the block adds multiple neurons to the hash table. The thread first probes the corresponding bucket, where the entries are linked by the *Next* array. If the neuron to be added already has an entry in the table, then the thread only needs to update the entry value atomically. Otherwise, the mutex of the bucket is required to ensure that a new item is safely inserted at the end of the bucket list. To avoid a race condition, the thread needs to examine the end of the list again after gaining the mutex. If the thread is unable to acquire the mutex, the stop flag of the block should be unset and the thread needs to wait for the next iteration. Note that a thread that fails to acquire the mutex also saves its current position in the list, allowing it to begin the probing process from this position rather than the head of the list in the next iteration. This fine-grained lock mechanism enables more parallelism than a coarse-grained lock mechanism for the whole table.

Then, utilizing this block-level hash table, we can launch a kernel whose block number is equal to the batch size, so that we can process the whole batch in parallel. By choosing the proper table size and hash function, we can make the conflicts rare and achieve high performance.

**Storage of the tables.** The space cost of the $L$ LSH tables can be expensive. Suppose each table has $B$ buckets, the size of each bucket is $b$, and the size of each permutation bin is $d$. Then, we have to allocate $L(Bb + Kd)$ memory space on the GPU, while the memory size on the GPU side is much smaller than that on the CPU side. Even worse, we also need to prepare enough memory space for linked hash tables with memory pools. To deal with this problem, we propose using the *unified memory* model introduced in CUDA 6.0.

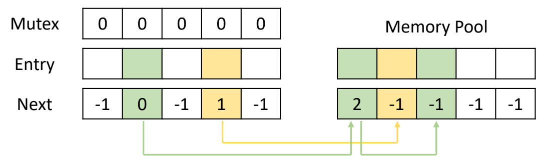
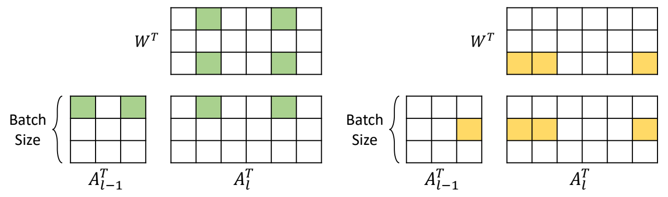


Fig. 9. The schema of the linked hash table on the GPU with mutexes and a memory pool to handle conflicts.



(a) Computation for the first row. (b) Computation for the second row.

Fig. 10. An example of the computations in forward propagation. The green and yellow cells represent the active neurons and corresponding weights for the two different rows in a batch.

Unified memory is a single address space accessible by both CPUs and GPUs in the same system [29]. GPUs with Pascal and later architectures further introduce the hardware page faulting and migration features, so the unified memory can indeed be allocated only when threads access it, rather than when we call the API of $cudaMallocManaged()$. This demand paging property makes the unified memory suitable for G-SLIDE, because we only use a small proportion of the total table memory.

### 4.4 Sparse Network Training

In this part, we show our design of the sparse neural network training on the GPU, which contains the forward/backward propagation and weight update.

**Forward propagation.** In typical neural networks, the primary operation in forward propagation is matrix multiplication, which already has efficient solutions in the cuBLAS [31] library. However, although the sparsity in SLIDE can significantly reduce the computation, it makes the traditional matrix multiplication method not applicable.

Figure 10 shows an example of the computations in forward propagation. The left matrix in Figure 10 (a) is the transpose of the previous layer's activations of a batch, denoted as $A_{l-1}^T$, while its right matrix is this layer's, denoted as $A_l^T$. The upper one is the transpose of the weight matrix, denoted as $W^T$. In dense neural networks, we will figure out the matrix of $A_l^T$ by simply multiplying $A_{l-1}^T$ and $W^T$ (with further bias addition and activation function). However, the situation in G-SLIDE is more complicated. First, in G-SLIDE, both the input matrix $A_{l-1}^T$ and the output matrix $A_l^T$ can be sparse, and the indices of the active cells in $A_l^T$ are determined by the LSH sparsification module prior to the forward computation. As a result, existing tools aimed at general matrix multiplication such as cuBLAS [31] and sparse matrix multiplication such as cuSPARSE [32] all fail to handle this unique feature. Second, the active neurons also vary for each row in $A_{l-1}^T$ and $A_l^T$, as shown in Figure 10 (a) and (b). Hence, this forward process cannot be regarded as a sub-matrix multiplication. Third, the cells in the weight matrix $W^T$ are activated according to $A_{l-1}^T$ and $A_l^T$, and it is not appropriate to compress the weight matrix $W^T$ for quick retrieval. The sparse accesses are therefore unavoidable. As far as we know, there are no efficient existing methods on the GPU designed for this special case.

For processing each row of the activation matrix, we compute the inner-product of an activation vector with a sub-matrix of $W^T$. Hence, we use one block to perform this row-vector - matrix multiplication. Similar to the design in the LSH table construction, we divide one row of the outputs into many tiles, and the responsible block will process the tiles sequentially. For each tile, we use one thread in the block to compute the activation of one active output neuron, which is the inner product of two vectors. As each thread in the block needs to read the previous activation values in the same row, we consider using shared memory to cache them.

**Softmax activation.** For Softmax activation, we need to perform maximum element search and summation, which are both reduction operations. We use one block to process the activations of a row as well, and utilize the warp shuffle technology [29] to realize efficient parallel reduction across a block. For ordinary neural networks, it is impractical to load the total row of activations into the shared memory. However, in G-SLIDE, benefiting from the extreme sparsity of the network, we only need to process a few neurons, which enables us to hold the total row in shared memory.

**Kernel fusion.** Fusing kernels properly can not only decrease the kernel launch time, but also reduce the global memory access cost significantly. As we mentioned before, in Softmax activation, we can cache an entire row thanks to the sparsity of neurons. Hence, after we multiply the previous activations and the weights, we can directly output the values to the shared memory for further Softmax activation, rather than writing back to global memory and reading again using another kernel.

The pseudo-code of Softmax forward kernel after fusion is shown in Algorithm 4. The function $blockReduce()$ in Lines 10 and 12 performs the parallel reduction mentioned before, with the parameter *op* determining the reduction operation. The function $blockTransform()$ in Lines 11 and 13 means that we use the block to perform the transform defined by the anonymous function parameter $f$ for each element in an array. During training, we even merge the error and gradient computation function into the kernel, which is not contained in the algorithm but further improves the performance.

**Backward propagation.** Similar to forward propagation, we use one block responsible for one rows' corresponding error propagation. As SLIDE points out, for backward propagation, we do not need to access non-active neurons and their corresponding weights [3]. Therefore, we can still cache both two rows of active neurons in two layers. SLIDE uses HOGWILD [17] to make gradient updates in parallel. In G-SLIDE, we use the atomic operation to maintain the consistency of gradients among the rows in a batch, which avoids the costly memory copy in SLIDE.

**Weight update.** After the backward propagation of a

**Algorithm 4** Forward propagation with Softmax activation

```
Input: weights $W^T$, biases $B^T$, previous layer's activations $A^T_{l-1}$
Output: this layer's activations $A^T_l$
1: procedure SOFTMAXFOWARDKERNEL
2:     $Row_{l-1} \leftarrow$ getRow($A^T_{l-1}, blockIdx$)
3:     $Row_{s,l-1} \leftarrow$ loadToSharedMem($Row_{l-1}$)
4:     $Row_{s,l} \leftarrow$ loadToSharedMem($B^T$)    ▷ initialize this layer's activation on shared memory with the value of biases
5:     for each tile $Row_{s,t,l}$ in $Row_{s,l}$ do
6:         $a_{ref} \leftarrow$ getResponsibleARef($Row_{s,t,l}, threadIdx$)
7:         $W_c \leftarrow$ getCorrespondW($W^T, Row_{s,l-1}$)
8:         $a_{ref} \leftarrow$ innerProduct($W_c, Row_{s,l-1}$)
9:     __syncthreads()
10:    $a_{max} \leftarrow$ blockReduce($Row_{s,l}, op = max$)
11:    blockTransform($Row_{s,l}, f$ = lambda $a$: exp($a - a_{max}$))
12:    $a_{sum} \leftarrow$ blockReduce($Row_{s,l}, op = sum$)
13:    blockTransform($Row_{s,l}, f$ = lambda $a$: $a/a_{sum}$)
14:    flushToGlobalMem($Row_{s,l}, A^T_l, blockIdx$)
```

batch, we update the total weights with the Adam optimizer [33]. The reconstruction of LSH tables after weight updates is also required, but the reconstruction operation is quite expensive. Therefore, we do not reconstruct the hash tables for every batch, but with a certain frequency instead. As SLIDE indicates [3], the frequency with exponential decay is also preferred.

**Adaptive kernel selection.** Using shared memory properly can significantly reduce the access overhead of global memory. However, the size of shared memory on the GPU is limited. The kernels using shared memory can fail to be launched if the scale of the data exceeds the threshold. To address this issue, we also develop kernels using little or no shared memory in G-SLIDE, and adaptively select the proper kernels to launch depending on the data scale as well as the available shared memory.

## 5 EVALUATION

### 5.1 Experimental Setup

In this part, we illustrate the evaluated methods, platforms, datasets, and hyper-parameters used in the experiments.

**Evaluated methods.** We use the same fully-connected neural network as in [3] for evaluation. We compare the proposed G-SLIDE system on GPUs with three solutions. The first one is SLIDE [3], denoted as "SLIDE". The second one is the TensorFlow implementation on CPUs, denoted as "TF-CPU". The third one is the TensorFlow implementation on GPUs, denoted as "TF-GPU". The source codes of these three solutions are all provided in [3].

**Platform for G-SLIDE and TF-GPU.** The experimental platform for G-SLIDE and TF-GPU is a server that consists of an 8-core/16-thread Intel(R) Core(TM) i9-9900K CPU at 3.60GHz and an NVIDIA GeForce RTX 2080 Ti graphics card. The GeForce RTX 2080 Ti GPU is powered by the Turing GPU architecture and the RTX platform, and this architecture is widely used for high-performance computing. This graphics card has 4,352 GPU cores, and its theoretical maximum floating-point performance is 28.5 TFLOPS (tera floating-point operations per second). The GPU integrates an 11GB GDDR6, and the memory bandwidth can reach 616 GB/s. The operating system we use is Ubuntu 20.04.2 LTS. The CUDA Toolkit versions for G-SLIDE and TF-GPU are 11.1 and 10.2, respectively. The cuDNN version for TF-GPU is 7.6.4.

**Platform for SLIDE and TF-CPU.** For fair comparisons, we rent a powerful server with a 32-core/64-thread Intel(R) Xeon(R) Platinum 8369B CPU at 2.70GHz on Alibaba Cloud [34] to evaluate SLIDE and TF-CPU.

**Datasets.** We evaluate G-SLIDE and other solutions on two real datasets from the Extreme Classification Repository [28]. Amazon-670K is a publicly available real-world dataset for product-to-product recommendation with 670K labels. The prediction task in Amazon-670K is to recommend products that the user might be interested in out of all 670K products, with a given interested product. WikiLSHTC-325K is a dataset extracted from Wikipedia with 325K labels. The statics of these datasets are shown in Table 1.

TABLE 1
Dataset statistics.

| Dataset | #Features | #Labels | #Train Points | #Test Points |
|---|---|---|---|---|
| Amazon-670K | 135,909 | 670,091 | 490,449 | 153,025 |
| WikiLSHTC-325K | 1,617,899 | 325,056 | 1,778,351 | 587,084 |

**Hyperparameters.** We use a fully-connected neural network model with a hidden dense layer of 128 neurons on both the Amazon-670K and WikiLSHTC-325K datasets. For Amazon-670K, we choose $K = 6$, $L = 50$, and 256 as the batch size. For WikiLSHTC-325K, we choose $K = 5$, $L = 350$, and 128 as the batch size. For both datasets, we reconstruct the LSH tables after training every 6,400 samples.

### 5.2 Results

In this part, we show the performance and accuracy results of G-SLIDE compared with the other baselines.

**Time-wise accuracy of G-SLIDE.** The time-wise accuracy plots of G-SLIDE on Amazon-670K and WikiLSHTC-325K are shown in Figure 11 (a) and (b), respectively. The training time on WikiLSHTC-325K is much longer than that of Amazon-670K due to its larger training set.

**Epoch-wise time comparison.** We measure the time consumed over epochs of G-SLIDE, SLIDE, TF-CPU, and TF-GPU, and show the results in Figure 12. We observe that G-SLIDE is significantly faster than SLIDE, TF-CPU, and TF-GPU on both datasets, which proves our idea of benefiting from both the adaptive sparsity of the neural network and the computing capability of the GPU. The detailed average one-epoch speedups over different solutions are presented in Table 2. There are three major reasons that make the speedups on WikiLSHTC-325K much smaller

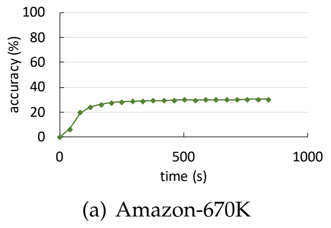


Fig. 11. Time-wise accuracy of G-SLIDE.

than the speedups on Amazon-670K. First, the sparsity of the network for Amazon-670K is greater than WikiLSHTC-325K, so more computations can be saved. Second, some fast kernels used on Amazon-670K cannot be launched on WikiLSHTC-325K due to the limited size of shared memory on the GPU. Accordingly, we need to access the slow global memory repeatedly without caching on WikiLSHTC-325K. Third, the greater number of active neurons on WikiLSHTC-325K leads to a longer active neuron query time.

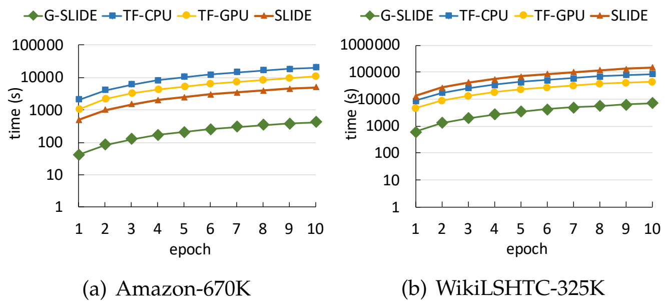


Fig. 12. Time consumed over epochs of G-SLIDE, SLIDE, TF-CPU, and TF-GPU.

TABLE 2
Average one-epoch speedups of G-SLIDE over different solutions.

| Dataset | SLIDE | TF-GPU | TF-CPU |
|---|---|---|---|
| Amazon-670K | 11.9× | 25.6× | 48.9× |
| WikiLSHTC-325K | 20.8× | 6.8× | 12.7× |

**Epoch-wise accuracy comparison.** We show the epoch-wise comparison between G-SLIDE and SLIDE, TF-CPU, and TF-GPU in Figure 13. We observe that the accuracy of G-SLIDE and SLIDE are very close to TF-CPU and TF-GPU, which is also indicated in [3]. After convergence, we can achieve about 90% accuracy of TensorFlow. The slight accuracy loss can come from the selected activation of neurons, which are figured out by querying the LSH hash tables. Besides, the parallel computation on the GPU is not the same as the sequential computing on the CPU due to the floating-point precision issue.

## 5.3 Detailed Analysis

We present a detailed analysis of the time for different parts of the G-SLIDE training process. We partition the execution process into several parts and show the time of each part within a batch in Figure 14. The whole process includes two forward propagation phases (denoted as “FW0” and

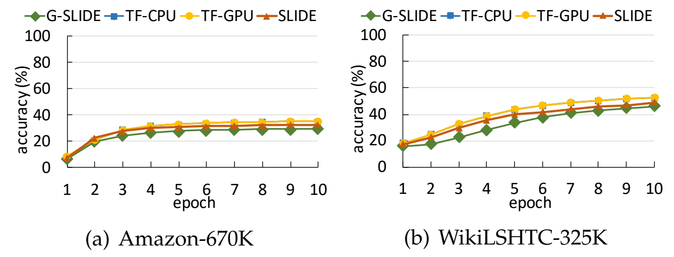


Fig. 13. Accuracy over epochs of G-SLIDE, SLIDE, TF-CPU, and TF-GPU.

“FW1”), backward propagation phases (denoted as “BP0” and “BP1”), weight update phases (denoted as “UD0” and “UD1”), active neuron query of LSH tables (denoted as “LSH QR”), and reconstructions of LSH tables (denoted as “LSH RC”). LSH table reconstructions are time-consuming. However, as discussed in Section 4.4, we do not need to reconstruct the tables for each batch, and thus its time cost can be balanced across multiple batches. Therefore, in Figure 14, we use the reconstruction time divided by the batch interval instead of the actual time.

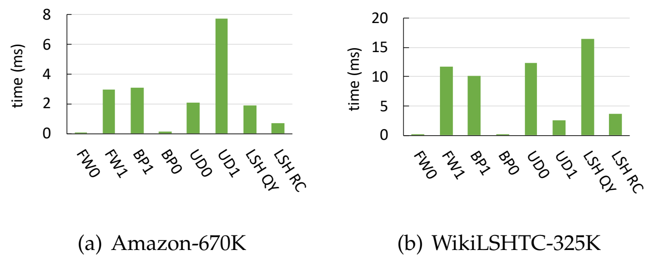


Fig. 14. Time for each part in a batch.

First, we observe that weight updates are costly on both datasets. The reason is that for weight update, we need to update the total weight matrix with Adam optimizer for each batch, while forward and backward propagation involves only the active neurons and their corresponding weights. Our analysis shows that about half of the time cost comes from the global memory accesses, while the other half is from the floating-point division. If we replace the division in the kernel with a low precision division, we can further reduce the overhead of weight update by nearly 50%.

Second, we also find that the most time-consuming part on WikiLSHTC-325K is LSH QY. This is because we set $L$ a high value of 350 and it takes a long time to aggregate the selected neurons of those $L$ LSH tables. The phases of FW1 and BP1 are also relatively expensive on WikiLSHTC-325K compared to Amazon-670K due to the lower network sparsity and the random accesses on global memory, which is also mentioned in Section 5.2.

Third, for forward and backward propagation on both datasets, we observe that the major overheads come from the last Softmax layer. This result is intuitive as the computation of this layer is more complicated than the hidden one, and the number of active neurons in this layer is also much larger than that of the sparse input features. The same phenomenon is observed in [3].

### 5.4 Discussion

In this part, we discuss the application scenarios and the limitations of G-SLIDE, and summarize our findings based on experiments.

**Application scope.** G-SLIDE sparsifies the neural networks adaptively using LSH-based randomized hashing and gains significant acceleration due to the computation savings on sparse layers. Hence, G-SLIDE is suitable for networks with wide layers and large sparse datasets. Besides, G-SLIDE is also designed for those resource-constraint scenarios. Researchers and companies without powerful hardware can train their huge network models in a very short time by using a single modest GPU with the help of G-SLIDE.

**Limitations.** There are two major limitations to current G-SLIDE. The first limitation is that G-SLIDE only targets fully connected layers currently similar to SLIDE. In the future, we will exploit the adaptive sparsity for more types of networks, such as convolutional neural networks. The second one is that G-SLIDE only focuses on a single GPU. To further improve the performance and scalability of G-SLIDE, we will extend G-SLIDE to support multi-GPU training.

**Summary of findings.** From the experiments, we have the following findings. First, combining LSH-based adaptive sparsification and GPU parallelism can result in significant performance improvements with close accuracy over SLIDE and TensorFlow implementations. Second, when the network is sparse enough, the active neurons can be cached in the shared memory on the GPU, and G-SLIDE can achieve high performance. Third, the major overheads of forward and backward propagation come from the last Softmax layer, and the weight update can be the most time-consuming part if the network is extremely sparse.

## 6 Related Work

**Machine learning systems.** Machine learning systems have become a hot topic in recent years [35], [36], [37], [38], [39], [40]. TensorFlow, PyTorch, and MXNet [41] are the most common machine learning systems. Apache Spark [42] is a popular large-scale data analytics engine, which has been widely used for the acceleration of machine learning applications [39], [40]. Researchers also proposed various machine learning systems to achieve more efficient model training and inference. Kalamkar *et al.* [37] accelerated the training process of deep learning recommendation systems on the CPU cluster. Mudigere *et al.* [43] proposed a co-designed system utilizing both software and hardware for the large-scale distributed training process of deep learning recommendation models. These works aim to train large recommendation models efficiently, whereas SLIDE [3], [6] and G-SLIDE aim to utilize the adaptive sparsity of neural networks. Besides, machine learning algorithms have been adopted to improve system performance. Tang *et al.* [35], [36] used machine learning algorithms in cloud computing systems for fair resource allocation.

**GPU acceleration of neural networks.** Due to the increasing development of neural networks and the high computing capacity of GPUs, many research studies on GPU acceleration of neural networks appear. Yao *et al.* [44] proposed a fine-grained sparsity method for efficient neural network inference on GPUs, called Balanced Sparsity. Cui *et al.* [45] developed GeePS, a GPU parameter server enabling scalable neural networks across distributed GPU systems. Chen *et al.* [46], [47] proposed ParSecureML, a secure deep learning acceleration system on GPUs.

**LSH-based applications.** Locality Sensitive Hashing (LSH) is an effective clustering method deployed in various applications. Shrivastava *et al.* [5] applied Asymmetric LSH to perform the task of approximate maximal inner product search. Ning *et al.* [48] proposed deep reuse, a method using LSH to quickly retrieve the similarities among neuron vectors to speed up the CNN inference. Pan *et al.* [49] developed an efficient GPU-based LSH technique to compute approximate k-nearest neighbor. Different from the previous work, in G-SLIDE, we use Winner Take All as our LSH function and focus on LSH-based neural network sparsification.

**Sparsity in neural networks.** Our work utilizes the sparsity in neural networks to accelerate the training and inference processes. Researchers have done related work on the discovery and development of sparsity in neural networks. Makhzani *et al.* [1], [2] observed that during every gradient update process, the neural network training can achieve high accuracy by ignoring most of the neurons according to their activation. Srivastava *et al.* [50] found that the method of selecting neurons sparsely in the neural network training process can achieve even higher accuracy due to implicit regularization. To further utilize the above sparsity feature in neural networks, Spring *et al.* [4] employed LSH to find a sparse set of neurons that needed to be activated, which is both efficient and valid. However, they did not achieve much improvement with this method compared with the method accelerated by hardware. SLIDE is proposed by Chen *et al.* [3], utilizing sparsity to perform neural network training and inference with high accuracy and efficiency.

## 7 Conclusion

There is currently no efficient GPU implementation of SLIDE. We find it a great opportunity to further improve the performance of SLIDE and propose G-SLIDE to fill this gap. G-SLIDE takes advantage of sparsity in large neural networks with smart LSH algorithms and data structures targeting GPUs. This paper expounds on how those specific designs and optimizations of GPUs are invented and implemented to better utilize the GPUs' computing capacity. Experiments show that G-SLIDE achieves 16.4× speedup over SLIDE, 30.8× over the TensorFlow solution on CPU, and 16.2× over the TensorFlow solution on GPU. This proves the huge potential of applying GPUs for sparse neural network acceleration with special designs.

## Reproducibility

We support reproducible science. G-SLIDE is available on GitHub (https://github.com/PanZaifeng/G-SLIDE), Mulan Open Source Community (https://code.mulanos.cn/MLbg3lrmuz/G-SLIDE), and Code Ocean.

## Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), National Natural Science Foundation of China (61732014, 62172419, and 61802412), and GHfund A (No. 20210701). This work is also sponsored by Alibaba Group through Alibaba Innovative Research (AIR) Program. Feng Zhang is the corresponding author of this paper.

## References

[1] A. Makhzani and B. Frey, "K-sparse autoencoders," *arXiv preprint arXiv:1312.5663*, 2013.

[2] A. Makhzani and B. J. Frey, "Winner-take-all autoencoders," *Advances in neural information processing systems*, vol. 28, pp. 2791–2799, 2015.

[3] B. Chen, T. Medini, J. Farwell *et al.*, "SLIDE: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 291–306, 2020.

[4] R. Spring and A. Shrivastava, "Scalable and sustainable deep learning via randomized hashing," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 445–454.

[5] A. Shrivastava and P. Li, "Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS)," *arXiv preprint arXiv:1405.5869*, 2014.

[6] S. Daghaghi, N. Meisburger, M. Zhao *et al.*, "Accelerating SLIDE deep learning on modern CPUs: Vectorization, quantizations, memory optimizations, and more," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[7] J. D. Owens, M. Houston, D. Luebke *et al.*, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[8] F. Zhang, J. Zhai, B. He *et al.*, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, 2016.

[9] F. Zhang, Z. Pan, Y. Zhou *et al.*, "G-TADOC: Enabling efficient GPU-based text analytics without decompression," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1679–1690.

[10] Z. Pan, F. Zhang, Y. Zhou *et al.*, "Exploring data analytics without decompression on embedded GPU systems," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[11] A. Z. Broder, M. Charikar, A. M. Frieze *et al.*, "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

[12] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, no. 6, 1999, pp. 518–529.

[13] J. Yagnik, D. Strelow, D. A. Ross *et al.*, "The power of comparative reasoning," in *2011 International Conference on Computer Vision*. IEEE, 2011, pp. 2431–2438.

[14] B. Chen and A. Shrivastava, "Densified winner take all (WTA) hashing for sparse datasets," in *Uncertainty in artificial intelligence*, 2018.

[15] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[16] Q. Lv, W. Josephson, Z. Wang *et al.*, "Multi-probe LSH: efficient indexing for high-dimensional similarity search," in *33rd International Conference on Very Large Data Bases, VLDB 2007*. Association for Computing Machinery, Inc, 2007, pp. 950–961.

[17] F. Niu, B. Recht, C. Ré *et al.*, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *arXiv preprint arXiv:1106.5730*, 2011.

[18] K. He, X. Zhang, S. Ren *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[19] S. Tang, C. Wang, J. Nie *et al.*, "EDL-COVID: Ensemble Deep Learning for COVID-19 Cases Detection from Chest X-Ray Images," *IEEE Transactions on Industrial Informatics*, 2021.

[20] Q. Cao, W. Zhang, and Y. Zhu, "Deep learning-based classification of the polar emotions of "moe"-style cartoon pictures," *Tsinghua Science and Technology*, vol. 26, no. 3, pp. 275–286, 2021.

[21] K. Zhu and T. Zhang, "Deep reinforcement learning based mobile robot navigation: A review," *Tsinghua Science and Technology*, vol. 26, no. 5, pp. 674–691, 2021.

[22] H. Huang, J. Lin, L. Wu *et al.*, "Machine learning-based multimodal information perception for soft robotic hands," *Tsinghua Science and Technology*, vol. 25, no. 02, pp. 255–269, 2020.

[23] F. Zhang, J. Zhai, X. Shen *et al.*, "TADOC: Text analytics directly on compression," *The VLDB Journal*, vol. 30, no. 2, pp. 163–188, 2021.

[24] F. Zhang, J. Zhai, X. Shen *et al.*, "POCLib: A High-Performance Framework for Enabling Near Orthogonal Processing on Compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 459–475, 2022.

[25] S. C. K. Tekouabou, S. Hartini, Z. Rustam *et al.*, "Improvement in automated diagnosis of soft tissues tumors using machine learning," *Big Data Mining and Analytics*, vol. 4, no. 1, pp. 33–46, 2021.

[26] A. Guezzaz, Y. Asimi, M. Azrour *et al.*, "Mathematical validation of proposed machine learning classifier for heterogeneous traffic and anomaly detection," *Big Data Mining and Analytics*, vol. 4, no. 1, pp. 18–24, 2021.

[27] J. Devlin, M.-W. Chang, K. Lee *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[28] K. Bhatia, K. Dahiya, H. Jain *et al.*, "The extreme classification repository: Multi-label datasets and code," 2016. [Online]. Available: http://manikvarma.org/downloads/XC/XMLRepository.html

[29] D. Guide, "CUDA C programming guide," *NVIDIA, July*, vol. 29, p. 31, 2013.

[30] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[31] Nvidia, "cuBLAS :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[32] Nvidia, "cuSPARSE :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[34] Alibaba, "Alibaba cloud." [Online]. Available: https://www.alibabacloud.com

[35] S. Tang, B.-S. Lee, and B. He, "Fair resource allocation for data-intensive computing in the cloud," *IEEE Transactions on Services Computing*, vol. 11, no. 1, pp. 20–33, 2016.

[36] S. Tang, Z. Niu, B. He *et al.*, "Long-term multi-resource fairness for pay-as-you use computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1147–1160, 2018.

[37] D. Kalamkar, E. Georganas, S. Srinivasan *et al.*, "Optimizing deep learning recommender systems training on CPU cluster architectures," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[38] J. Sun, G. Sun, S. Zhan *et al.*, "Automated performance modeling of HPC applications using machine learning," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 749–763, 2020.

[39] X. Meng, J. Bradley, B. Yavuz *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[40] S. Tang, B. He, C. Yu *et al.*, "A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[41] T. Chen, M. Li, Y. Li *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[43] D. Mudigere, Y. Hao, J. Huang *et al.*, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," 2021.

[44] Z. Yao, S. Cao, W. Xiao *et al.*, "Balanced sparsity for efficient DNN inference on GPU," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 5676–5683.

[45] H. Cui, H. Zhang, G. R. Ganger *et al.*, "Geeps: Scalable deep learning on distributed GPUs with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[46] Z. Chen, F. Zhang, A. C. Zhou *et al.*, "ParSecureML: An efficient parallel secure machine learning framework on GPUs," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

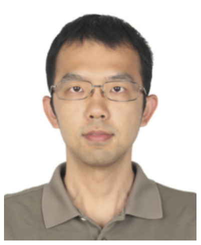
[47] F. Zhang, Z. Chen, C. Zhang *et al.*, "An efficient parallel secure machine learning framework on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2262–2276, 2021.

[48] L. Ning and X. Shen, "Deep reuse: streamline CNN inference on the fly via coarse-grained computation reuse," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 438–448.

[49] J. Pan and D. Manocha, "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation," in *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2011, pp. 211–220.

[50] N. Srivastava, G. Hinton, A. Krizhevsky *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

**Zaifeng Pan** received the bachelor degree from Shanghai Jiao Tong University in 2021, and he is pursuing the master degree in computer science at Renmin University of China, advised by prof. Feng Zhang. His current research interests include parallel computing and heterogeneous computing.



**Feng Zhang** received the bachelor degree from Xidian University in 2012, and the PhD degree in computer science from Tsinghua University in 2017. He is an associate professor with the Key Laboratory of Data Engineering and Knowledge Engineer (MOE), Renmin University of China. His major research interests include high performance computing, big data systems, and parallel and distributed systems.

**Hourun Li** is an undergraduate in School of Information, Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2021.

**Chenyang Zhang** is an undergraduate in School of Information, Renmin University of China. She joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2019. Her research interests include big data management, heterogeneous computing, and machine learning systems.

**Xiaoyong Du** obtained the B.S. degree from Hangzhou University, Zhejiang, China, in 1983, the M.E. degree from Renmin University of China, Beijing, China, in 1988, and the Ph.D. degree from Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.

**Dong Deng** received the bachelor degree from Beihang University in 2011 and PhD degree in computer science from Tsinghua University in 2016. He is an assistant professor in the Computer Science Department at Rutgers University - New Brunswick. His research interests include data management, database system, data curation, and data-centric AI.