Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism

Dian-Lun Lin and Tsung-Wei Huang

Abstract—The ever-increasing size of modern deep neural network (DNN) architectures has put increasing strain on the hardware needed to implement them. Sparsified DNNs can greatly reduce memory costs and increase throughput over standard DNNs, if the loss of accuracy can be adequately controlled. However, sparse DNNs present unique computational challenges. Efficient model or data parallelism algorithms are extremely hard to design and implement. The recent effort MIT/IEEE/Amazon HPEC Graph Challenge has drawn attention to high-performance inference methods for large sparse DNNs. In this article, we introduce SNIG, an efficient inference engine for large sparse DNNs. SNIG develops highly optimized inference kernels and leverages the power of CUDA Graphs to enable efficient decomposition of model and data parallelisms. Our decomposition strategy is flexible and scalable to different partitions of data volumes, model sizes, and GPU numbers. We have evaluated SNIG on the official benchmarks of HPEC Sparse DNN Challenge and demonstrated its promising performance scalable from a single GPU to multiple GPUs. Compared to the champion of the 2019 HPEC Sparse DNN Challenge, SNIG can finish all inference workloads using only a single GPU. At the largest DNN, which has more than 4 billion parameters across 1920 layers each of 65536 neurons, SNIG is up to 2.3× faster than a state-of-the-art baseline under a machine of 4 GPUs. SNIG receives the Champion Award in 2020 HPEC Sparse DNN Challenge.

Index Terms—Task graph parallelism

1 Introduction

Large deep neural network (DNN) models have brought significant quality improvement to several fields, including natural language processing, speech recognition, and image classification [6], [23], [27]. To relieve the increasing strain on the hardware needed to deploy them, much research over the past decades has focused on the *sparsification* of DNNs in the interest of reduced storage and runtime costs [10], [14], [17]. Computing large sparse DNNs presents unique computational challenges and scaling difficulties. Sparseness can make the application of the DNN on current processors extremely inefficient. This inefficiency limits the size of data to what can be held in GPU memory, or it requires a high-end, expensive cluster of computers to make up for this inefficiency. Also, sparse DNN inference presents unique computational challenges from training, because the kernel efficiency largely depends on nonzero entries that vary from layer to layer. To address these problems for advancing emerging sparse machine learning (ML) systems, the 2019 MIT/IEEE/Amazon HPEC Graph Challenge has developed Sparse DNN Challenge to encourage new solutions for sparse DNN inference [19]. Table 1 lists the statistics of each sparse DNN. The largest network contains over 4 billion nonzero parameters across 1920 layers each of 65536 neurons, adding up to 100 GB memory storage.

The challenge of computing large sparse DNN inference is twofold, *kernel* and *decomposition algorithms*, both of which require strategic designs to benefit from parallelism. Existing kernel algorithms focus on optimizing sparse matrix-matrix multiplication kernels or carefully maintaining data sparsity during the weight propagation [9], [22], [25], [29]. However, most of these approaches require models to sit in the GPU memory, and they are difficult to operate on partitioned pieces, due to the cost of maintaining consistent sparse matrix structures between partitions along with iterations. Existing decomposition strategies divide large data or models into partitions and distribute partitions across GPUs [7], [15], [20], [28]. Partitioning data and models can both improve parallelism and alleviate the tension on hardware constraints, including memory limitations and communication bandwidths on GPUs. However, efficient decomposition algorithms are extremely hard to design and implement. We need to address complexity among GPU capacity, scaling flexibility, and inference efficiency. To simplify the design, *pipeline parallelism* has been a popular choice in existing frameworks [5], [13], [24], [26]. The idea of the pipeline is simple and easy to implement, but it suffers from many performance problems, including synchronous execution, imbalanced load, and limited pipeline depth.

As a consequence, we introduce SNIG, an efficient large sparse DNN inference engine using *task graph parallelism*. SNIG develops highly optimized inference kernels that can effectively avoid unnecessary computation incurred by zero entries during the inference iterations. We leverage the power of modern CUDA Graph [3] to enable efficient decomposition of model and data parallelisms. Our decomposition

• *The authors are with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84112 USA. E-mail: dian-lun.lin@utah.edu, twh760812@gmail.com.*

Manuscript received 1 Sept. 2021; revised 17 Dec. 2021; accepted 24 Dec. 2021. Date of publication 28 Dec. 2021; date of current version 23 May 2022.
This work was supported in part by NSF under Grants CCF-2126672 and NumFOCUS small development Grant.
(Corresponding author: Dian-Lun Lin.)
Recommended for acceptance by A. J. Peña, M. Si and J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2021.3138856

TABLE 1
DNN Benchmark Statistics in HPEC Sparse DNN Graph Challenge [19]

Neurons/Layers	120	480	1920	Bias	Size	Nonzeros
1024	3.9M	15.7M	62.9M	-0.30	1.25 GB	6,374,505
4096	15.7M	62.9M	251.7M	-0.35	5.40 GB	25,019,051
16384	62.9M	251.7M	1.0B	-0.40	22.70 GB	98,858,913
65536	251.7M	1.0B	4.0B	-0.45	94.70 GB	392,191,985

The largest benchmark consists of 1920 fully-connected layers each of 65536 neurons.

strategy transforms a partitioned inference workload into a GPU task graph that flows dependent GPU operations naturally with the graph structure, providing improved scheduling efficiency and runtime asynchrony. Atop the task graph parallelism, we have designed a new kernel algorithm that can efficiently avoid unwanted computation and incrementally update memory entries during the inference iterations. Compared with existing solutions, SNIG is more flexible and cost-efficient in fitting together partitioned data and models into different GPUs under hardware constraints.

We demonstrate the flexibility and efficiency of SNIG on the 12 large sparse DNNs provided by the 2019 HPEC Sparse DNN Challenge [19]. SNIG is able to complete all DNNs using only one RTX 2080 Ti GPU of 11 GB memory, and we solve the largest DNN 2.27× faster than the 2019 champion solution developed by Bisson and Fatica ("BF" method for brevity) [5]. Compared with a pipeline baseline inspired by GPipe [13], SNIG is faster at almost all networks (up to 2.19× speedup) and scales better on multiple GPUs. With these promising results, SNIG receives the Champion Award in 2020 HPEC Sparse DNN Challenge [1]. We believe SNIG stands out as a unique inference engine for large sparse DNNs, given the ensemble of kernel algorithm designs and parallel decomposition strategies we have made.

2 Background

2.1 HPEC Sparse DNN Graph Challenge

We target on the 2019 HPEC Sparse DNN Graph Challenge, which is based on a mathematically well-defined DNN inference computation and can be implemented in any programming environment [19]. The input data, Y_0, is derived from the MNIST handwritten letters by resizing each 28×28 pixel image to 32×32 (1024 neurons), 64×64 (4096 neurons), 128×128 (16384 neurons), and 256×256 (65536 neurons). The weight matrices of each sparse DNN, including the bias vectors, are generated by the RadiX-Net synthetic sparse DNN generator with a number of desirable properties such that participants can focus on the difficult, computational part of the problem [18]. The inference problem is to compute $Y_{l+1} = h(Y_l W_l + B_l)$ for each layer where $h(y) = max(y, 0)$ is a nonlinear function of rectified linear unit (ReLU). For the Sparse DNN Challenge, $h(y)$ has an upper limit set to 32. The surrounding I/O and verification provide the context for each sparse DNN inference that allows rigorous definition of both the input and the output. Table 1 lists the statistics of each sparse DNN and its input image set. Loading the smallest DNN can take gigabytes of memory using single-precision floating numbers. Preloading all matrices to GPUs is impractical. The Graph Challenge evaluates each solution based on two metrics, *correctness* in comparison to a golden reference and *performance* in terms of execution time to perform DNN inference.

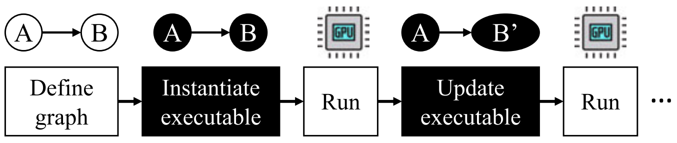


Fig. 1. Execution model of CUDA Graph consists of four major steps, graph definition, executable graph instantiation, graph execution, and executable graph parameter updates.

2.2 CUDA Graph

The new CUDA Graph programming model (since CUDA v10) allows users to express dependent GPU tasks in a task dependency graph and offload it directly to a GPU using minimal kernel call and scheduling overheads [3]. This organization can deliver significant yet largely untapped performance advantage for many large machine learning workloads. Specifically, modern GPUs are very fast and the overheads of kernel calls have become significant in many machine learning workloads that compose thousands of GPU operations and dependencies in forms of task graphs. These task graphs normally do not change once the neural network architecture is decided. There is no need to repetitively offload the same task graph using expensive host function calls and custom stream scheduling algorithms. Furthermore, CUDA runtime can perform architecture-specific and whole-graph optimizations that are almost impossible to achieve by a third-party library. For instance, the new Ampere architecture GPU A100 adds new hardware features to make the paths between grids in a task graph significantly faster [2].

Fig. 1 presents the execution model of a CUDA graph which consists of *graph definition*, *executable instantiation*, *graph execution*, and *graph update*. CUDA Graph offers two methods to define or construct a task graph, *explicit graph construction* and *implicit graph capture*. Users can explicitly construct a CUDA graph by creating nodes and edges to describe GPU operations and their dependencies. However, this method requires full details of kernel execution parameters which are often unavailable for vendor libraries, such as cuBLAS and cuSparse. To overcome this problem, users can implicitly *capture* a CUDA graph by creating streams in *capturing modes* and inserting events for cross-stream dependencies. Implicit graph capture is flexible but it takes additional steps of deciding *how* dependent GPU operations are inserted into streams and linked via events. Regardless of the explicit or implicit method, users instantiate an *executable graph* from a constructed graph and offload that executable graph to a GPU using a single host call. The overhead of graph definition and instantiation can be amortized over many executions, and graphs provide a clear advantage over streams. Between successive executions, users can update the execution parameters of a GPU operation or a node in the graph.

While CUDA Graph opens new research opportunities to accelerate machine learning workloads, there are two major challenges users need to overcome. First, CUDA Graph programming is extremely tedious. Users need to wrangle with

TABLE 2
Description of Graph Construction, Graph Execution, and Graph Update Methods in cudaFlow [12]

cudaFlow API category	Method	Description
Graph construction	`kernel(grid, block, shm, kernel_name, args)`	create a kernel task
	`copy(target, source, byte_count)`	create a memory copy task
	`precede(task)`	create a dependency to a task
Graph execution	`offload()`	offload a cudaFlow
	`offload_until(n)`	offload a cudaFlow n times
Graph update	`kernel(task, grid, block, shm, kernel_name, args)`	update the parameters of a kernel task
	`copy(task, target, source, count)`	update the parameters of a memory copy task

many error-prone and low-level details, including but not limited to parameter settings, stream/event insertions, and concurrency controls. The same code written in CUDA Graph can be 2–5× more complicated than that of streams. Second, in situations where the task graph topology is not changing, updating node parameters between successive executions requires very detailed attention to the update rules of CUDA Graph, such as context rules, memory allocation rules, child graph rules, and so on. As a result, the lack of suitable programming abstraction over CUDA Graph is imposing a high barrier on its broad adoptions due to large programming complexities.

3 SNIG

At a high level, SNIG describes the inference workload in a *GPU task graph* that comprises both data-level and model-level parallelisms. We introduce a C++ programming model called cudaFlow to abstract the programming complexity of building CUDA graphs. Our task graph can scale to arbitrary sizes of DNN and input data under different numbers of GPUs. We design a new inference kernel inside the task graph that computes only necessary entries during the inference iterations. Our kernel incorporates an efficient pruning strategy to avoid unwanted computation incurred by sparsified network and data. Everything runs in a single end-to-end task graph and there is no extra CPU-GPU or GPU-GPU synchronization to redistribute the input data among GPUs as the state of the art [5]. In the following sections, we will first introduce our cudaFlow programming model atop CUDA Graph and discuss how SNIG uses it to describe data and model parallelisms in a GPU task graph. Then, we will present our new kernel solution for the inference workload.

3.1 cudaFlow Programming Model

To enable broad adoption of CUDA Graph, we introduce a C++-based programming model called *cudaFlow* to abstract the programming complexities of CUDA Graph. cudaFlow methods consist of three major categories, *graph construction*, *graph execution*, and *graph update*. Table 2 describes a partial list of graph construction, graph exectuion, and graph update methods; complete API can be referred to [12]. Listing 1 implements the canonical saxpy (A•X plus Y) task graph that composes two host-to-device (H2D) copies, one saxpy kernel, and two device-to-host (D2H) copies using our cudaFlow programming model. Between successive executions (i.e., `offload`), we update the parameters of the task `kernel` with different sizes of grid (`GRID2`), block (`BLOCK2`), and saxpy kernel parameters. The code explains itself through a succinct graph description language. The same code that describes this workload but using the plain CUDA Graph API is shown in Listing 2 which is a lot more complicated. For instance, adding a memory copy node using the plain CUDA Graph API requires over 15 lines of code, whereas cudaFlow requires only a single line. The key advantage of cudaFlow is that we reduce a large amount of boilerplate code and parameter settings that are unnecessary for expressing common GPU operations.

Listing 1. cudaFlow Program of Constructing an Explicit Saxpy (“Single-Precision A·X + Y”) CUDA Graph

```
__global__ void saxpy(int n,int a,int *x,int *y);
// create a cudaFlow task graph
cudaFlow cudaflow;
auto h2d_x  = cudaflow.copy(dx, hx, N);
auto h2d_y  = cudaflow.copy(dy, hy, N);
auto d2h_x  = cudaflow.copy(hx dx, N);
auto d2h_y  = cudaflow.copy(hy, dy, N);
auto kernel = cudaflow.kernel(
  GRID, BLOCK, SHM, saxpy, N, 2.0f, dx, dy
);
kernel.succeed(h2d_x, h2d_y)
      .precede(d2h_x, d2h_y);
cudaflow.offload();
// update the kernel parameter
cudaflow.kernel(kernel,
  GRID2, BLOCK2, SHM, saxpy, N, 1.0f, dx, dy
);
cudaflow.offload();
```

When users *offload* a cudaFlow, an executable graph will be created in that cudaFlow which is completely hidden from users. Each graph creation method (e.g., `copy`, `kernel`) in cudaFlow comes with an overload that takes an additional argument of an existing cudaFlow task to which the rest arguments will be applied to update in its present executable graph. cudaFlow is primarily used for explicit graph construction when all the GPU operation parameters are known to programmers. For implicit graph capture, cudaFlow provides a method `capture` that allows users to capture a GPU task graph through a stream-based interface `cudaFlowCapturer::on`. Listing 3 gives an example of capturing a saxpy kernel task using a third-party library that issues the kernel through the given stream.

Unlike a cudaFlow that keeps a one-to-one mapping between a user-level graph and its native CUDA graph, a cudaFlow capturer does not have this mapping because it is impossible to know how a third-party library launches kernels. For instance, a reduction algorithm may spawn

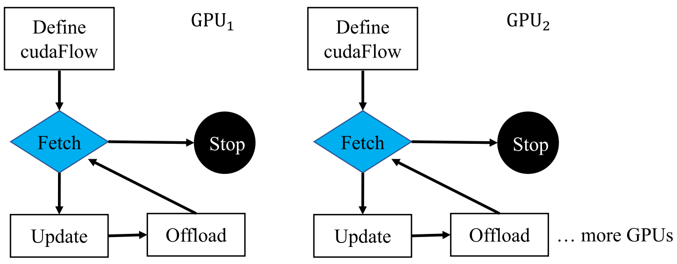


Fig. 2. Architecture of SNIG.

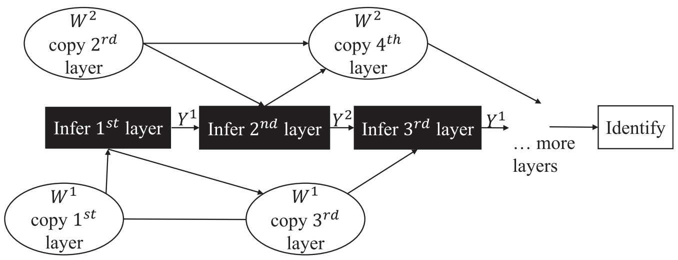

W^2 copy 2rd layer
W^2 copy 4th layer
Infer 1st layer
Y^1
Infer 2nd layer
Y^2
Infer 3rd layer
Y^1
… more layers
Identify
W^1 copy 1st layer
W^1 copy 3rd layer

Fig. 3. Partial cudaFlow diagram to perform the inference for one batch iteration.

different numbers of kernels for different input sizes along the reduction tree. This inherent limitation restricts users from updating a captured graph on a per-node basis but on a *per-graph* basis. Specifically, each cudaFlow capturer stores all the parameters of user graphs and *lazily* constructs the CUDA graph when an offload call is issued. If the executable graph exists from the previous offload call, we update it from the newly constructed CUDA graph. Otherwise, we instantiate a new executable graph from this CUDA graph.

Listing 2. Partial Code of Rewritten Listing 1 Using Plain CUDA Graph API. Adding a CUDA Graph Memory Copy Node Requires Over 15 Lines of Boilerplate Code Whereas cudaFlow Requires Only a Single Line

```
cudaGraph_t graph;
cudaGraphCreate(&graph, 0);
std::vector<cudaGraphNode_t> nodeDependencies;
cudaGraphNode_t h2d_x, h2d_y, d2h_x, d2h_y;
cudaMemcpy3DParms memcpyParams = {0};
memcpyParams.srcArray = NULL;
memcpyParams.srcPos = make_cudaPos(0, 0, 0);
memcpyParams.srcPtr = make_cudaPitchedPtr(
  hx, sizeof(float) * N, N, 1
);
memcpyParams.dstArray = NULL;
memcpyParams.dstPos = make_cudaPos(0, 0, 0);
memcpyParams.dstPtr = make_cudaPitchedPtr(
  dx, sizeof(float) * N, N, 1
);
memcpyParams.extent = make_cudaExtent(
  sizeof(float) * N, 1, 1
);
memcpyParams.kind = cudaMemcpyHostToDevice;
cudaGraphAddMemcpyNode(
  &h2d_x, graph, NULL, 0, &memcpyParams
));
//... hundreds of lines of code to follow
```

Listing 3. cudaFlow Capturer Program of Capturing a Saxpy Kernel Task Using an Existing Stream-Based Function Call

```
auto g = cudaflow.capture([](cudaFlowCapturer c){
  auto kernel = c.on([](cudaStream_t stream){
    launch_3rdparty_saxpy_kernel(stream);
  });
  // ... other captured tasks
});
```

3.2 Task Graph Architecture of SNIG

Fig. 2 shows the overview of SNIG. SNIG describes the inference workload in a task graph that includes one CPU task, *fetch*, and three GPU tasks, *define graph*, *update*, and *offload*. Our task graph defines a cudaFlow once and iteratively fetches a batch of input images to perform inference via offloading the defined cudaFlow. At each batch iteration, we update kernel inputs to the newly fetched input batch without rebuilding a CUDA graph, hence reducing CUDA Graph construction overhead. Our task graph iterates *fetch*, *update* and *offload* GPU tasks until there are no input images left. In the *fetch* task, a CPU task grabs a batch of input images. Users can tune batch size based on available GPU memory. To have multiple threads fetch data at the same time, we use an *atomic* counter to represent the remaining number of images. In the *offload* task, SNIG computes the inference on an input batch by offloading the defined cudaFlow on a GPU. SNIG is extensible due to its decentralized architecture. Users can easily extend to multiple GPUs by allocating a cudaFlow to a GPU. Each GPU infers a batch of inputs independently and shares the *atomic* counter indicating the remaining input data.

Fig. 3 illustrates details of our cudaFlow that performs the inference for one batch iteration. Each node represents one of the three GPU operations, host-to-device (H2D) copy, device-to-host (D2H) copy, and kernels. Each edge represents the dependency of two GPU operations. SNIG transposes the weight matrix of each sparse DNN layer and stores them using the Compressed Sparse Column (CSC) format. Since preloading all models to the GPU is impossible due to memory limit, we store the entire weight matrices into pinned host memory, and only keep a few weight buffers (W^1, W^2, ...) on a GPU at a time. All weight buffers have the same size equal to the maximum size of all weight matrices. More weight buffers results in a higher overlap between data communication and kernel computation. For example, we have two weight buffers, W^1 and W^2, in Fig. 3. The weight copy of the 2nd layer can overlap the inference of the 1st layer. Since the inference at one layer only depends on the results from the previous layer, we allocate for each GPU two *result buffers* Y^1 and Y^2 each of size ($batch\_size \times num\_neurons$), where $batch\_size$ denotes the input batch size and $num\_neurons$ denotes the number of neurons per input data, to perform rolling swap for storage optimization. Each result buffer can be accessed via modulo operation on 2; in each layer l, we use $Y^{l\%2+1}$ as input and $Y^{(l+1)\%2+1}$ as output. After completing the inference at the last layer, the GPU identifies the categories (predicted digits). SNIG is highly flexible with limited GPU memory. Users can configure different input batch sizes and number of weight buffers based on available GPU memory to fit arbitrary sizes of models and input data.

3.3 Inference Kernel

As we perform inference layer by layer, the number of nonzero rows in each result buffer (i.e., Y^1 and Y^2) is significantly

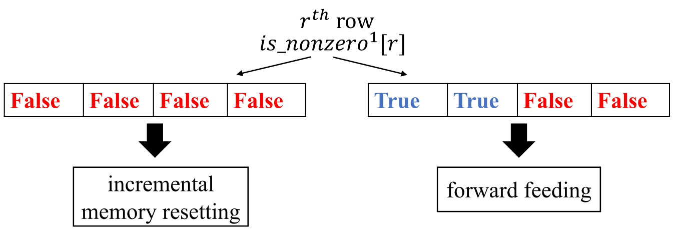


Fig. 4. Illustration of the beginning of the inference kernel.

reduced. Also, the number of zero elements in each nonzero row increases with the number of computed layers. Our inference kernel consists of two parts, *incremental memory resetting* and *forward feeding*, to efficiently prune unwanted computation incurred by empty rows and zero elements in each nonzero row. To fully utilize GPU's massively parallel architecture, we divide each row of Y^1 and Y^2 into several *sections* and assign each GPU block to compute a section. The number of sections is based on the size of GPU shared memory, and the section size is the number of neurons in a section. We allocate a boolean buffer, $is\_nonzero^1$ and $is\_nonzero^2$, for each result buffer to indicate whether a section contains at least one nonzero element. The size of each boolean buffer is ($batch\_size \times num\_secs$), where num_secs denotes the number of sections. In SNIG, each GPU keeps two result buffers and two boolean buffers for rolling swap. For simplicity, all examples in the following paper take (Y^1, $is\_nonzero^1$) as input and (Y^2, $is\_nonzero^2$) as output.

In our kernel parameter settings, the grid dimension is ($batch\_size$, num_secs, 1), and the block dimension is (2, 512, 1). We allocate (4 $\times$ sec_size) bytes of external shared memory, where sec_size denotes the section size. The kernel is launched by $<<<$ ($batch\_size$, num_secs, 1), (2, 512, 1), (4 $\times$ sec_size) $>>>$. Each GPU block $block_{r,sec}$ computes the sec^{th} section at the r^{th} row of Y^2 independently.

Algorithm 1. Beginning of the Inference Kernel

```
Input: $num\_neurons$: Number of neurons in a row of $Y^1$ and $Y^2$
Input: $num\_secs$: Number of sections in a row of $Y^1$ and $Y^2$
Input: $sec\_size$: Number of neurons in a section of $Y^1$ and $Y^2$
1  $r \leftarrow$ blockIdx.x
2  $sec \leftarrow$ blockIdx.y
3  $tid \leftarrow$ threadIdx.y * blockDim.x + threadIdx.x
4  $num\_threads \leftarrow$ blockDim.x * blockDim.y
5  $is\_all\_zero \leftarrow$ true
6  For $s \leftarrow 0$; $s < num\_secs$; ++$s$ do
7      $is\_all\_zero$ &= !$is\_nonzero^1[r][s]$
8  end
9  if $is\_all\_zero$ == $true$ then
10     /* Incremental memory resetting...  */
11 end
12 else
13     /* Forward feeding...  */
14 end
```

Fig. 4 illustrates the beginning of our kernel. We inspect each entry in the r^{th} row of $is\_nonzero^1$. If there is no true value, meaning $Y^1[r]$ only contains zero elements, we enter *incremental memory resetting*. Otherwise, we enter *forward feeding*. Algorithm 1 presents the details of the beginning of our inference kernel. We use is_all_zero to record if all entries

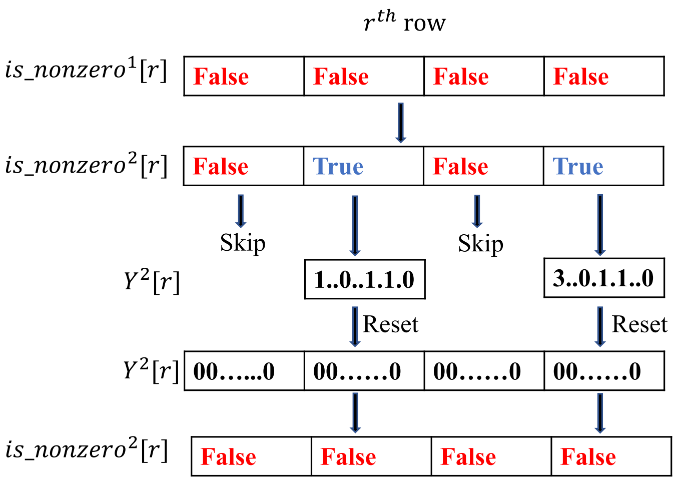


Fig. 5. Illustration of incremental memory resetting.

in $is\_nonzero^1[r]$ are false (line 5-8). Note that since SNIG determines to execute either *forward feeding* or *incremental memory resetting* per row of $is\_nonzero^1$, GPU blocks with the same r enter into the same kernel part (line 9-14).

3.3.1 Incremental Memory Resetting

The goal of incremental memory resetting is to avoid unwanted computations induced by empty rows between successive inference iterations. Fig. 5 shows the process of incremental memory resetting. Taking advantage of rolling swap, we perform incremental memory resetting to reset buffers. If all entries in $is\_nonzero^1[r]$ are false, we inspect $is\_nonzero^2[r]$ and only reset nonzero sections in $Y^2[r]$. This largely avoids the overhead to reset the entire linear buffer for the next iteration to use. Our implementation computes each section in parallel and calculates only necessary elements during inference iterations. After resetting $Y^2[r]$, we set $is\_nonzero^2[r]$ to all false.

Algorithm 2 presents the details of *incremental memory resetting*. The GPU block $block_{r,sec}$ first inspects $is\_nonzero^2[r][sec]$. If false, we directly return. Otherwise, each GPU thread resets an element at a time until all elements in the section are zero (line 2-4). After resetting, we toggle $is\_nonzero^2[r][sec]$ to false (line 6).

Algorithm 2. Incremental Memory Resetting

```
1 if $is\_nonzero^2[r][sec]$ == $true$ then
2     for $j \leftarrow tid$; $j < sec\_size$; $j$ += $num\_threads$ do
3         $Y^2[r][sec\_size * sec + j] = 0$
4     end
5     __syncthreads()
6     $is\_nonzero^2[r][sec] \leftarrow$ false
7 end
8 return
```

3.3.2 Forward Feeding

The goal of forward feeding is to perform matrix multiplication followed by ReLU and pass the results to the next layer via rolling swap, while skipping unnecessary computations induced by zeros. By inspecting $is\_nonzero^1$, our algorithm can efficiently skip a section that contains only zero elements without checking one element at a time.

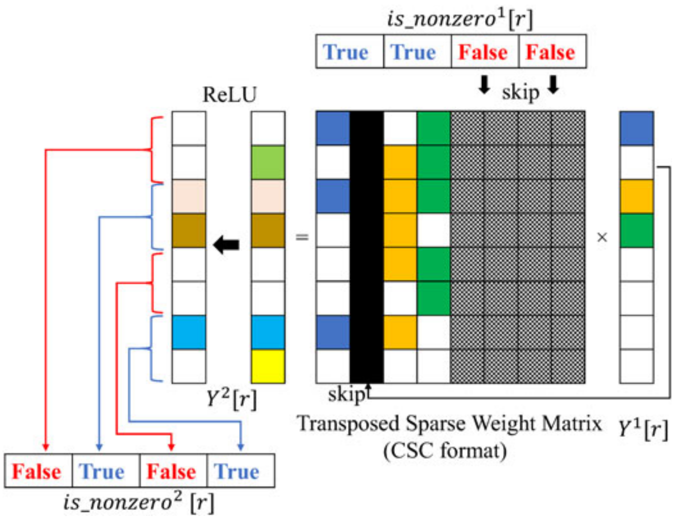


Fig. 6. Illustration of forward feeding.

Fig. 6 illustrates the process of *forward feeding* that each section contains two elements. The white color represents zeros. SNIG first inspects each entry in $is\_nonzero^1[r]$. Since the third and fourth entries in $is\_nonzero^1[r]$ are false, we can directly skip the computation on the corresponding sparse weight matrix columns shown as the grey area. During the matrix-vector multiplication, we find the second element of $Y^1[r]$ is zero, meaning we can skip the computation on the second column of the sparse weight matrix shown as the black area. We then pass $Y^2[r]$ to ReLU and set each entry in $is\_nonzero^2[r]$ based on the final results of $Y^2[r]$. For instance, since the first and second elements of $Y^2[r]$ are zero, we set the first entry in $is\_nonzero^2[r]$ to false.

Algorithm 3 presents the details of *forward feeding*. $block_{r,sec}$ computes the sec^{th} section of $Y^2[r]$. Each GPU block declares a shared memory array $results$ size of sec_size to store results (line 1) and initializes $results$ to the bias value directly (line 2-4). To avoid thread-level synchronization, we use a boolean array $rec\_nonzero$ size of two to record whether $results$ has nonzero values (line 5-6, line 31). During the matrix-vector multiplication, we iterate one section of $Y^1[r]$ at a time (line 8-26). If $is\_nonzero^1[r][s]$ is false, meaning the s^{th} section of $Y^1[r]$ contains only zero elements, we skip all elements in this section directly (line 9-11). Otherwise, all threads along y dimension loop through all elements in the s^{th} section of $Y^1[r]$ (line 13). We directly skip to the next element if the current element is zero (line 15-17). All threads along x dimension read *col_w* (line 18-19) and iterate the weight values and the weight row indices (line 20-22). To compute each section in Y^2 independently, we transform the dimension of each CSC weight matrix from $(num\_neurons, num\_neurons)$ to $(num\_neurons, num\_secs \times num\_neurons)$. All column indices are shifted by $j = j + num\_neurons \times (i/sec\_size)$, where (i, j) is the index of nonzeros in a weight matrix. In line 18-19, we read column indices of the weight matrix via adding the offset. Then, we multiply each nonzero input element with weight value and add the result to the corresponding location of *results* (line 23).

After matrix-vector multiplication, SNIG loops through the $results$ (line 28). $block_{r,sec}$ computes ReLU, writes the result to each element in the sec^{th} section of $Y^2[r]$, and sets $rec\_nonzero[1]$ to true if there exists a nonzero result (line 29-31). Finally, we toggle $is\_nonzero^2[r][sec]$ to either true or false based on $rec\_nonzero[1]$ (line 34).

Algorithm 3. Forward Feeding

```
Input: $col\_w$: array of column offsets of the weight matrix
Input: $row\_w$: array of row indices
Input: $val\_w$: array of values
1  extern_ shared_ $results[]$
2  for $k \leftarrow tid$; $k < sec\_size$; $k$ += $num\_threads$ do
3      $results[k] \leftarrow bias$
4  end
5  _shared_ $rec\_nonzero[2]$
6  $rec\_nonzero[1] \leftarrow$ false
7  __syncthreads()
8  for $s \leftarrow 0$; $s < num\_secs$; $++s$ do
9      if $!is\_nonzero^1[r][s]$ then
10         continue
11     end
12     $j \leftarrow$ threadIdx.y + $s * sec\_size$
13     for $j$; $j < (s+1) * sec\_size$; $j$ += blockDim.$y$ do
14         $y^{val} \leftarrow Y^1[r][j]$
15         if $y^{val} == 0$ then
16             continue
17         end
18         $w^- \leftarrow col\_w[sec * num\_nurons + j]$ + threadIdx.x
19         $w^+ \leftarrow col\_w[sec * num\_neurons + j + 1]$
20         for $k \leftarrow w^-$; $k < w^+$; $k$ += $blockDim.x$ do
21             $w^{row} \leftarrow row\_w[k]$
22             $w^{val} \leftarrow val\_w[k]$
23             atomicAdd(&$results[w^{row} - sec * sec\_size]$, $y^{val} * w^{val}$)
24         end
25     end
26 end
27 __syncthreads()
28 for $i \leftarrow tid$; $i < sec\_size$; $i$ += $num\_threads$ do
29     $v \leftarrow$ min(32, max($results[i]$, 0))
30     $Y^2[r][sec * sec\_size + i] \leftarrow v$
31     $rec\_nonzero[v \neq 0] \leftarrow$ true
32 end
33 __syncthreads()
34 $is\_nonzero^2[r][sec] = rec\_nonzero[1]$
```

4 Experimental Results

We evaluate SNIG's performance on the official MIT/IEEE/Amazon HPEC Sparse DNN Challenge Dataset [19]. The benchmark statistics are shown in Table 1. We implement SNIG using C++17 and CUDA nvcc 11.1 on a host compiler of GNU GCC-8.3.0 with C++17 standards `-std=c++17` and optimization flags `-O2` enabled. We undertake our experiments on two machines to demonstrate the efficiency of CUDA Graph, 1) a Ubuntu Linux 5.0.0-21-generic x86 64-bit desktop of 40 2.0 GHz Intel Xeon Gold 6138 CPU cores and four GeForce RTX 2080 Ti GPUs with 11 GB memory and 2) an Nvidia's internal Linux server of one A100 GPU with 80 GB memory. All data is an average of ten runs with `float` type. We will first present our experimental results based on our champion-award solution for the 2020 HPEC Sparse DNN Challenge (Sections 4.1, 4.2, and 4.3) [21] and then present new performance improvement by leveraging CUDA

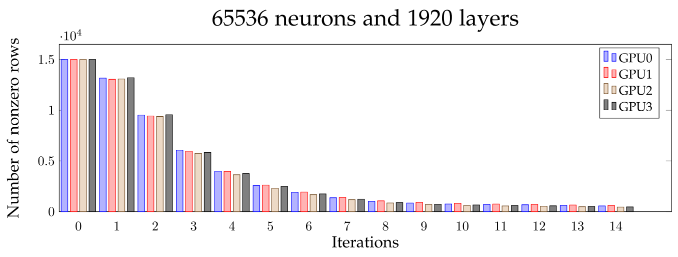


Fig. 7. Number of nonzero rows in 15 iterations on 4 GPUs using BF without NVLink.

Graph Update and A100 GPU (Sections 4.4 and 4.5). We target the benchmarks of the 2019 HPEC Graph Challenge because it defines a rigorous evaluation environment for participants to focus on the computational efficiency of large sparse neural network inference algorithms. We do not consider other models/datasets because none of them are as large as the HPEC benchmarks [19].

4.1 Baseline

We consider BF and GPipe* methods for our baseline. The BF method is the champion solution of the 2019 HPEC Sparse DNN Challenge [5]. We implemented the BF method and its kernel using CUDA streams and OpenMP. The original BF method relies on NVLink to transparently exchange data among GPUs using unified addressing. Since we do not have NVLink, such a process can be very time-consuming. We manually partition the input data in the beginning evenly across GPUs and spawn one OpenMP thread to call the inference function per GPU. This organization does not impact the load-balancing performance of BF because according to our experiment, the number of nonzero rows per iteration is very balanced at each GPU. For example, as shown in Fig. 7, the difference of the number of nonzero rows at each GPU is within 350 rows ($<0.5\%$ of the total rows) across all iterations. We implemented the GPipe* method based on GPipe [13]. GPipe is an iterative framework for training large DNNs. We extended its idea to inference by partitioning the DNN into multiple stages across GPUs and pipelining each data batch's execution over these stages using CUDA streams and OpenMP threads. We use SNIG's inference kernels inside the pipeline. The goal of comparing GPipe* with SNIG is to compare the performance difference between task graph- and pipeline-based kernel scheduling for the inference workload.

We configure the block dimension of all kernels to (2, 512, 1), the batch size of input data to 5000 for SNIG and GPipe*, and the number of weight buffers to 2 for SNIG to achieve the best performane. We will discuss the effect of different parameters in Section 4.3.

4.2 Performance Comparison

Table 3 compares the overall inference rate and runtime performance between SNIG, BF, and GPipe* using one, two,

TABLE 3
Overall Inference Rate (Gigaedges Processed per Second) and Runtime Performance (Seconds) of SNIG, BF, and GPipe* Across One, Two, Three, and Four GPUs

		Number of GPUs										
		1		2			3			4		
Neurons	Layers	BF	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG
1024	120	**345.93** (0.682s)	295.28 (0.799s)	576.84 (0.409s)	**589.82** (0.400s)	455.46 (0.518s)	**761.06** (0.310s)	695.95 (0.339s)	689.85 (0.342s)	867.38 (0.272s)	768.50 (0.307s)	**1248.30** (0.189s)
	480	477.83 (1.975s)	**586.52** (1.609s)	801.11 (1.178s)	**1016.93** (0.928s)	926.12 (1.019s)	1061.55 (0.889s)	1273.57 (0.741s)	**1348.16** (0.700s)	1112.87 (0.848s)	1483.83 (0.636s)	**1982.60** (0.476s)
	1920	524.50 (7.197s)	**718.74** (5.252s)	852.50 (4.428s)	**1187.81** (3.178s)	1184.45 (3.187s)	1133.59 (3.330s)	1575.48 (2.396s)	**1647.69** (2.291s)	1220.45 (3.093s)	1876.17 (2.012s)	**2159.53** (1.748s)
4096	120	409.42 (2.305s)	**586.52** (1.609s)	746.02 (1.265s)	934.37 (1.010s)	**980.99** (0.962s)	1106.35 (0.853s)	1053.25 (0.896s)	**1460.86** (0.646s)	1385.78 (0.681s)	1165.08 (0.810s)	**2241.61** (0.421s)
	480	544.55 (6.932s)	**803.84** (4.696s)	962.73 (3.921s)	1376.68 (2.742s)	**1400.69** (2.695s)	1431.50 (2.637s)	1767.26 (2.136s)	**2062.77** (1.830s)	1743.59 (2.165s)	2069.5 (1.824s)	**2761.42** (1.367s)
	1920	586.38 (25.75s)	**867.28** (17.41s)	1032.09 (14.63s)	1551.53 (9.732s)	**1575.48** (9.584s)	1538.09 (9.817s)	2074.67 (7.278s)	**2284.34** (6.610s)	1879.21 (8.035s)	2506.97 (6.023s)	**2948.54** (5.121s)
16384	120	462.32 (8.165s)	**851.53** (4.433s)	881.36 (4.283s)	1290.55 (2.925s)	**1487.34** (2.538s)	1303.47 (2.896s)	1521.51 (2.481s)	**2183.26** (1.729s)	1621.50 (2.328s)	1684.45 (2.241s)	**2914.96** (1.295s)
	480	616.30 (24.50s)	**1076.99** (14.02s)	1137.01 (13.28s)	1887.67 (7.999s)	**1965.31** (7.683s)	1678.28 (8.997s)	2454.80 (6.151s)	**2824.44** (5.346s)	2072.39 (7.286s)	2894.28 (5.217s)	**3736.57** (4.041s)
	1920	663.34 (91.05s)	**1113.94** (54.22s)	1207.71 (50.01s)	2105.92 (28.68s)	**2127.43** (28.39s)	1808.86 (33.39s)	2817.06 (21.44s)	**3022.92** (19.98s)	2230.35 (27.08s)	3412.31 (17.70s)	**3963.12** (15.24s)
65536	120	28.79 (524.3s)	**1021.61** (14.78s)	57.52 (262.5s)	1323.35 (11.41s)	**1870.36** (8.073s)	1332.70 (11.33s)	1486.17 (10.16s)	**2705.51** (5.581s)	1652.74 (9.136s)	1565.85 (9.643s)	**3436.38** (4.394s)
	480	(>1800s)	**1404.60** (43.00s)	58.81 (1027s)	2083.40 (28.99s)	**2583.31** (23.38s)	1817.57 (33.23s)	2768.00 (21.82s)	**3784.33** (15.96s)	2241.94 (26.94s)	3222.94 (18.74s)	**5071.19** (11.91s)
	1920	(>1800s)	**1489.46** (162.2s)	(>1800s)	1501.50 (160.9s)	**2810.51** (85.96s)	1960.97 (123.2s)	1948.32 (124.0s)	**4149.63** (58.22s)	2450.47 (98.59s)	2784.27 (86.77s)	**5561.50** (43.44s)

Bold text represents the best solution in the corresponding benchmark. All results match the golden reference provided by the MIT/IEEE/Amazon Sparse DNN Challenge [19]. Since the GPipe method is staged on the number of GPUs, we do not report its runtime under one GPU.*

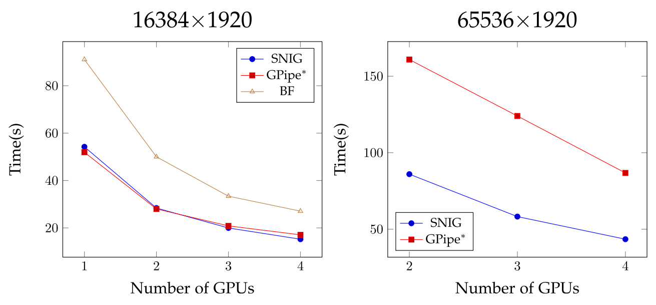


Fig. 8. Execution time with different numbers of GPUs.

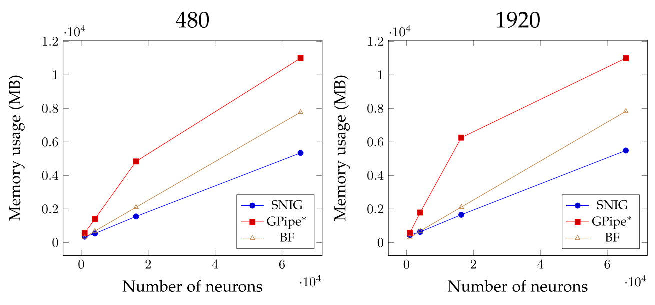


Fig. 10. Peak GPU memory usage under four GPUs.

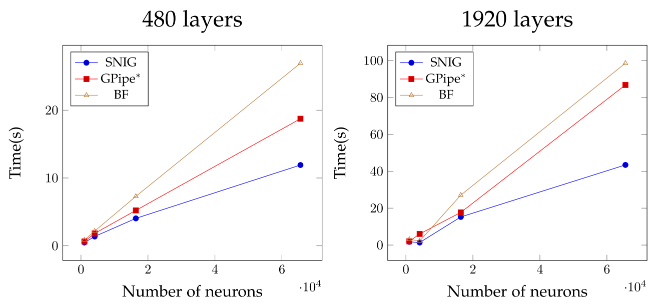


Fig. 9. Execution time with different neurons under four GPUs.

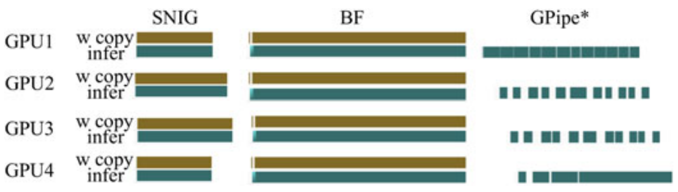


Fig. 11. Execution timeline of each method on completing 65536 neurons and 1920 layers under four GPUs.

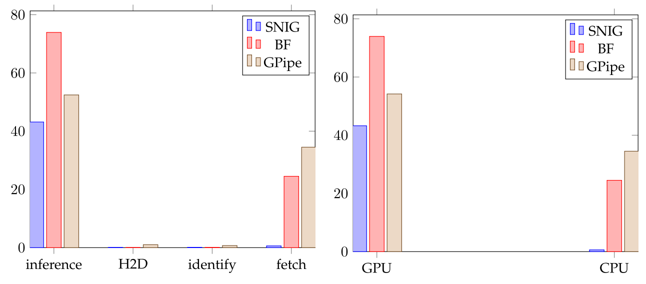


Fig. 12. CPU-GPU runtime breakdown of each method on the 65536 × 1920 benchmark under four GPUs. The execution time of GPU includes inference kernels, H2D, and identify kernels.

three, and four GPUs. The result of BF method is different from BF paper due to different GPU platforms. SNIG outperforms BF and GPipe* across nearly all benchmarks. With four GPUs, SNIG is 2.3× faster than BF on the largest DNN of 65536 neurons and 1920 layers and is 2.2× faster than GPipe* on the DNN of 65536 neurons and 120 layers. The BF method failed to finish the largest DNN of 65536 neurons and 1920 layers within a reasonable amount of time (> 1800 seconds) under one and two GPUs. This is because BF requires the entire input data to sit in the GPU under unified memory addressing to implement load balancing. CUDA will keep fetching in and out data between CPUs and GPUs if partitioned data does not fit in a GPU's memory. Its kernel design is architecturally constrained by the number of GPUs and available memory. Similar problems exist in the GPipe* method as well since GPipe* requires the entire model to sit in GPUs. We observe long runtime of GPipe* to complete the DNNs of 65536 neurons and 1920 layers.

Fig. 8 plots the scalability over the increasing number of GPUs. Our runtime scales the best among the three methods. In the 16384 × 1920 scenario, SNIG outperforms BF by 1.7×, 1.8×, 1.7×, and 1.8× at one, two, three, and four GPUs, respectively. In the 65536×1920 scenario, SNIG outperforms GPipe* by 1.9×, 2.1×, 2.0× at two, three, and four GPUs, respectively. We attribute this to the synchronization overhead of both methods (BF at each iteration, GPipe* at each pipeline stage). Fig. 9 plots the scalability over the increasing number of neurons. SNIG outperforms BF and GPipe* in all scenarios. The growth rate of our runtime is much slower than BF and GPipe*, due to our in-kernel pruning strategy and task parallelism. Fig. 10 illustrates the peak GPU memory usage of each method. Both SNIG and BF demand less memory than GPipe* because of buffered rolling swap, whereas GPipe* stages the model across GPUs. Our memory is fewer than BF due to batched input data.

Fig. 11 plots a partial GPU execution timeline of each method using the data extracted from Nvidia Profiler (nvprof) [4] under the same time scale. Since SNIG and BF do not pipeline the model across GPUs, both methods require weight copy during the inference iterations. However, the time for data transfers is largely overlapped with the kernel computation (i.e., task parallelism in SNIG and stream parallelism in BF). In SNIG, each GPU performs the inference on a data batch independently, and thus the runtime of each GPU is different. The execution timeline of GPipe* at each GPU is more fragmented and discontinued than SNIG and BF. This is because computation and GPU-to-GPU data transfers at each pipeline level need to synchronize before moving to the next stage. For example, we can clearly see several white spaces between successive GPU operations at GPU 2 and GPU 3, which is taken by CPU. Fig. 12 shows the CPU-GPU runtime breakdown of each method on the 65536 × 1920 benchmark using four GPUs. We can clearly see the advantage of SNIG. By leveraging CUDA Graph, we achieve end-to-end task parallelism on GPU.

4.3 Parameter Sensitivity

Fig. 13 shows the impact of different block dimensions. All implementations have the same trend and perform better at lower dim_x, especially under a large number of neurons. All kernels read input data along y dimension and iteratively access weights along x dimension. Since weights are sparse matrices, the overhead is dominated by reading input data.

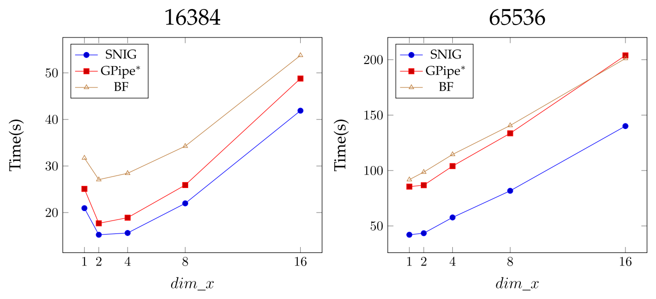


Fig. 13. Execution time with different block dimensions (dim_x, dim_y) on 1920 layers under four GPUs. The total number of threads $dim\_x \times dim\_y$ remains 1024.

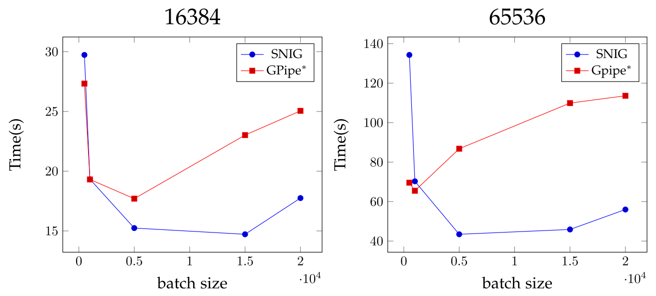


Fig. 14. Execution time for different batch sizes with 1920 layers using four GPUs.

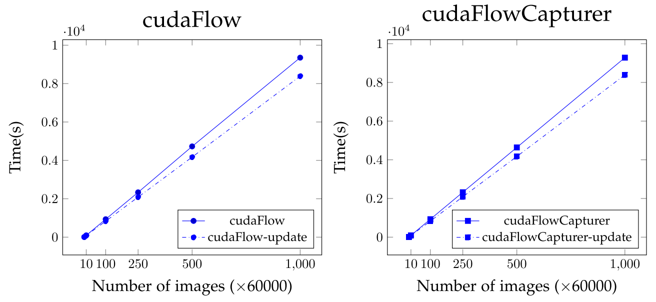


Fig. 15. Execution time of cudaFlow, cudaFlow-update, cudaFlowCapturer, and cudaFlowCapturer-update on 1024×1920 benchmark with different numbers of images using one GPU. The batch size is 1000.

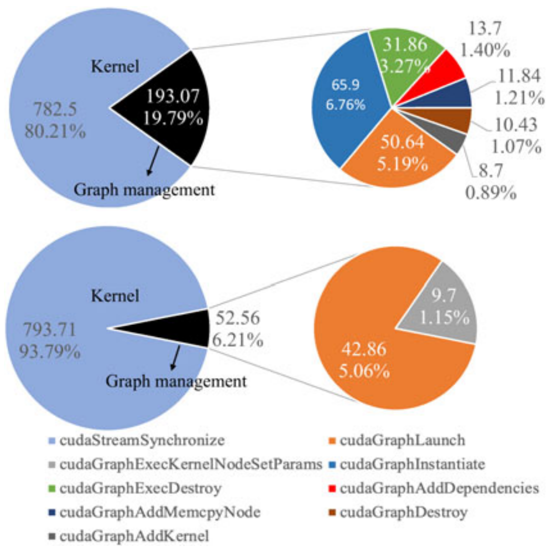


Fig. 16. Runtime breakdown (seconds & percentage) of cudaFlow (top) and cudaFlow-update (bottom) on 1024×1920 benchmark with 6M images. The right side shows detailed breakdown of CUDA Graph function calls.

Fig. 14 shows the impact of different input batch sizes in SNIG and GPipe*. Partitioning input data with too small batch size results in a lousy performance, while a bigger batch size doesn't gain speedup. GPipe* has a higher growth rate of runtime than SNIG. We attribute this to the architecture of GPipe* and GPU memory limitation. Since GPipe* pipelines computation across GPUs, large input batch size of large DNNs causes long CPU-GPU and GPU-GPU data communication times. SNIG does not require any GPU-GPU data transfers.

4.4 CUDA Graph Update

The previous sections discuss SNIG based on the 2020 HPEC Sparse DNN Graph Challenge environment [21], which targets only 60K input images. Since the input number is small, the advantage of CUDA Graph Update is not clear. However, many real-world image inference applications request large numbers of inputs, and the effect of updating CUDA graphs becomes significant. Specifically, instead of repetitively constructing and destroying a new cudaFlow at each batch iteration, it is desirable to create a cudaFlow once in the beginning and update its parameters for the rest of iterations. Therefore, in this experiment, we enlarge the input size by duplicating 60K images 10×, 100×, 250×, 500×, 1000× and perform the inference on the 1024×1920 benchmark using one GPU. Since SNIG constructs an independent cudaFlow for each GPU, the data of one GPU is sufficient to demonstrate the advantage of CUDA Graph Update. Hereafter, *cudaFlow* represents SNIG without using updating methods, *cudaFlow-update* represents SNIG using updating methods, *cudaFlowCapturer* represents SNIG using implicit graph capture, and *cudaFlowCapturer-update* represents SNIG using implicit graph capture and updating methods.

Fig. 15 shows the execution time of each method on the 1024×1920 benchmark using one GPU. As the number of input images increases, the gap between cudaFlow and cudaFlow-update becomes remarkable (left figure). For example, at 60K images (i.e., HPEC Challenge Specification) the difference between cudaFlow and cudaFlow-update is about 1 second, whereas at 60M images, the difference becomes 963 seconds (10% improvement by cudaFlow-update). Since cudaFlow iteratively defines a graph, instantiates an executable graph, and destroys a graph/executable graph at each batch iteration, the overhead of CUDA Graph function calls becomes significant as the number of images grows. By contrast, cudaFlow-update creates and destroys the graph once and updates its parameters for the rest of batch iterations. Similar performance improvement can be observed in cudaFlowCapturer (right figure). At 60M images, cudaFlowCapturer-update brings about 6% improvement over cudaFlowCapturer. The improvement is less than cudaFlow-update because the update method of cudaFlowCapturer is applied on a per-graph basis instead of per-node basis.

Fig. 16 shows the runtime breakdown of cudaFlow and cudaFlow-update on 6M input images based on the data extracted from NVIDIA Visual Profiler [4]. The right side shows the detailed breakdown of all CUDA Graph function calls. Compared with cudaFlow, cudaFlow-update largely reduces the overhead of CUDA Graph function

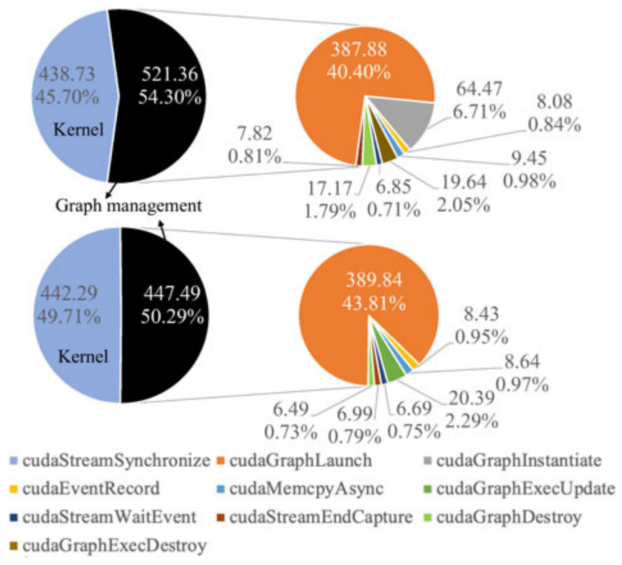


Fig. 17. Runtime breakdown (seconds & percentage) of cudaFlowCapturer (top) and cudaFlowCapturer-update (bottom) on 1024×1920 benchmark with 6M images. The right side shows detailed breakdown of CUDA Graph function calls.

calls by 73% (193.07 seconds versus 52.56 seconds). That is because cudaFlow needs to iteratively rebuild CUDA graphs, the overheads of `cudaGraphInstantiate`, `cudaGraphExecDestroy`, `cudaGraphDestroy`, and so on are much larger than the update counterpart `cudaGraphExecKernelNodeSetParms` in cudaFlow-update. For example, as shown in Fig. 16, the `cudaGraphExecKernelNodeSetParms` takes only about 9.7 seconds, whereas the time to rebuild CUDA graphs is 142.43 seconds.

Fig. 17 shows the runtime breakdown of cudaFlowCapturer and cudaFlowCapturer-update. Unlike cudaFlow that explicitly constructs a CUDA graph, cudaFlowCapturer implicitly captures a CUDA graph using streams andevents(e.g., `cudaStreamEndCapture`, `cudaEventRecord`, `cudaStreamWaitEvent`). The key difference between cudaFlowCapturer and cudaFlowCapturer-update is that cudaFlowCapturer repetitively issues `cudaGraphInstantiate` (shown as grey area) from a captured CUDA graph over batch iterations rather than updating the existing executable graph with that captured CUDA graph using `cudaGraphExecUpdate`. For example, cuda FlowCapturer spends 64.47 seconds on repetitively instantiating an executable CUDA graph, whereas cudaFlowCapturer-update spends <1 second on instantiating an executable graph once and 20.39 seconds for updating that executable graph for the rest of batch iterations. Since implicit CUDA Graph construction requires both methods to re-capture a new CUDA graph, the time spent on event and stream managements are roughly the same (22.11 versus 24.12).

4.5 A100 GPU

The experiments in the previous sections are based on RTX-2080 Ti GPU, which has only 11 GB memory. Considering this memory capacity, the batch size that achieves the best inference performance is 5K based on our experiment. Larger batch size will result in fetching in and out data between CPUs and GPUs because the data cannot fit in a GPU. This problem can be overcome by high-end GPUs that are particularly designed for large-scale machine learning workloads, such as the new Nvidia A100 GPU that has 80 GB memory. The large memory capacity allows us to load a much larger machine learning model and dataset into GPU at a time, significantly reducing the data-movement overheads in our batch iterations. In this section, we compare the runtime performance of SNIG on different benchmarks among one RTX-2080 Ti GPU (SNIG-RTX-2080) and one A100 GPU (SNIG-A100, BF-A100, GPipe*-A100).

We first evaluate the performance based on the HPEC Graph Challenge environment. Since A100 GPU can accommodate the entire 60K images, we set batch size of 60K for SNIG-A100 and GPipe*-A100 to compute the inference in just a single data batch iteration. SNIG-RTX-2080 uses a batch size of 5K images. Table 4 compares the execution time among SNIG-RTX-2080 , SNIG-A100, BF-A100, and GPipe*-A100 on four benchmarks with the largest layer count (1920) at four neuron numbers (1024, 4096, 16384, and 65536). In the 16384×1920 benchmark, SNIG-A100 is $3.04\times$ faster than SNIG-RTX-2080. SNIG-A100 with 60K batch size only fetches input images once from CPU and does not need to update the cudaFlow, whereas SNIG-RTX-2080 with 5K batch size requires the GPU to iteratively request input images from CPU and updates the cudaFlow multiple times. SNIG-A100 outperforms BF-A100 acrosss four benchmarks due to our in-kernel pruning strategy and task parallelism. On the other hand, SNIG-A100 has similar performance to GPipe*-A100 since both methods compute the inference on the entire input data at one time without additional data piping between CPU and GPU.

As we presented in Section 4.4, real-world image inference applications can contain large numbers of input images that go beyond the memory capacity of A100. In this case, SNIG-A100 needs to partition the input into several data batch iterations and update the cudaFlow between successive batch iterations. Fig. 18 compares the execution time between SNIG-RTX-2080 and SNIG-A100 on the 1024×1920 benchmark at different numbers of input images using cudaFlow-update and cudaFlowCapturer-update implementations (similar to Section 4.4) at a batch size of 5K. With cudaFlow-update (left side), SNIG-A100 is consistently faster than SNIG-RTX-2080. The largest speedup we have observed is $1.78\times$ at 30M images, and the performance gap continues to enlarge as we increase the number of input images. Similar data is also observed in cudaFlowCapturer-update (right side).

TABLE 4
Runtime Comparison (Seconds) Between One RTX-2080 Ti GPU (SNIG-RTX-2080) and One A100 GPU (SNIG-A100, BF-A100, GPipe*-A100) on Four Benchmarks at the Largest Layer Count With Different Neurons

		RTX-2080	A100		
Neurons	Layers	SNIG	SNIG	BF	GPipe*
1024	1920	5.25s	1.93s	3.62s	1.89s
4096	1920	17.41s	6.04s	10.37s	6.24s
16384	1920	54.22s	17.80s	35.38s	18.82s
65536	1920	162.2s	69.45s	134.82s	70.60s

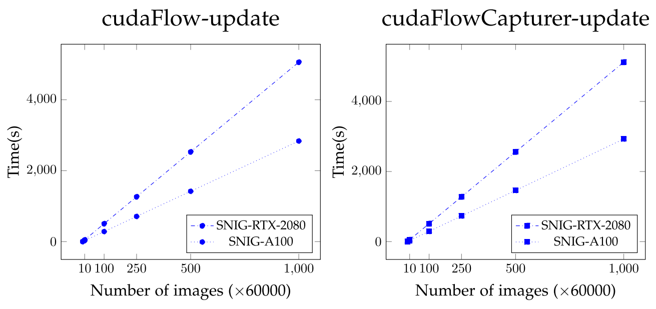


Fig. 18. Execution time of cudaFlow-update and cudaFlowCapturer-update on the 1024 × 1920 benchmark with different numbers of images using one A100 GPU and one RTX-2080 Ti GPU. The batch size is 5000.

4.6 Effectiveness of Our Kernel Algorithms

To demonstrate the effciency of our kernels, we replace incremental memory resetting with `cudaMemset` and forward feeding with BF's kernel, respectively. Table 5 compares execution time of SNIG, SNIG without incremental memory resetting (SNIG-w/o-inc), and SNIG without forward feeding (SNIG-w/o-ff) on four benchmarks using one A100 GPU. SNIG is consistently faster than SNIG-w/o-inc. That is because SNIG performs rolling swap between iterations for storage optimization. Without incremental memory resetting, we need to call `cudaMemset` at each iteration to reset result buffers, thus inducing huge overhead. On the other hand, SNIG outperforms SNIG-w/o-ff across all benchmarks. Forward feeding allows SNIG to efficiently skip sections that contain only zero elements by inspecting boolean buffers, whereas BF's kernel needs to check one element at a time.

5 Prior Work

The BF method [5] is the champion-award solution in 2019 HPEC Sparse DNN Graph Challenge [1]. In BF method, each GPU owns a part of the input matrix and computes the inference kernel iteratively by one OpenMP thread. To achieve load balancing, BF method requires communication between CPUs and GPUs at each inference iteration, resulting in huge overhead. Besides, BF requires the entire input data to sit in GPUs for implementing load balancing. Similar problems also exist in other pipeline-based frameworks. For example, GPipe [13] proposes pipelining computation across GPUs and synchronizing data transfers stage by stage. The efficiency and scalability are largely limited by the size of partitioned data and available GPU resources that decide the degree of pipeline parallelism. Hidayetoğlu *et al.* [11] propose register tiling, shared-memory tiling, and compact index representation to implement an optimized kernel fused with ReLU activation for sparse DNN inference. Their kernel algorithm targets improving memory bandwidth and irregular memory accesses induced by the irregular sparsity at each inference iteration. Like the BF method, their solution requires communication between CPUs and GPUs at each inference iteration. FlexFlow [16] is a deep learning framework that automatically finds parallelization strategies over a defined search space for accelerating DNN training. They use a guided randomized search procedure to explore the space of possible strategies via a predictor of DNN performance. However, they do not target large-scale machine learning inference workload.

TABLE 5
Runtime Comparison (Seconds) Among SNIG, SNIG Without Incremental Memory Resetting (SNIG-w/o-inc), and SNIG Without Forward Feeding (SNIG-w/o-ff) on Four Benchmarks Using One A100 GPU

Neurons	Layers	SNIG	SNIG-w/o-inc	SNIG-w/o-ff
1024	1920	1.93s	2.90s	27.88s
4096	1920	6.04s	8.03s	107.13s
16384	1920	17.80s	25.51s	420.86s
65536	1920	69.45s	103.54s	1824.06s

There is a great deal amount of research on general sparse matrix-matrix multiplication (SpGEMM) using GPUs [8]. However, there are two challenges that prevent us from directly using them for reaching the best performance in HPEC Sparse DNN Graph Challenge. First, most SpGEMM algorithms assume the matrix size fits in GPU, which is not possible in our case. Second, existing SpGEMM algorithms target standalone SpGEMM problem instances, rather than the entire sparse DNN inference workload. This prevents us from leveraging advanced CUDA Graph parallelism that combines customized partitioning and pruning strategies across inference iterations to maximize the performance.

6 Conclusion

In this paper, we have introduced SNIG, an efficient inference engine for large sparse DNNs. We have described the inference workload in a *task graph* comprising both data- and model-level parallelisms. Our decomposition method can scale to arbitrary sizes of DNN and input data under different numbers of GPUs. With four GPUs, SNIG is 2.3× faster than BF and is 2.0× faster than GPipe* on the largest DNN of 65536 neurons and 1920 layers (more than 4 billion nonzero parameters). By using CUDA Graph Update, we have shown further 10% performance improvement compared to SNIG without updating methods. Our future work will research new pipeline-based task graph schedulers for training large neural networks and apply our algorithms to other practical models, such as CNNs and GNNs.

Acknowledgments

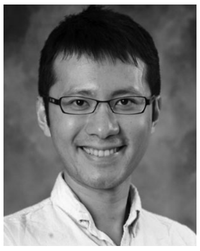
Preliminary version of this article has received the Champion Award in the 2020 IEEE High-performance and Extreme Computing Conference (HPEC) Graph Challenge in Spare Neural Network [21]. The authors appreciate H. Ren, Y. Zhang, and B. Khailany at Nvidia for their help with running SNIG on the A100 machine.

References

[1] HPEC Graph Challenge Champions, 2022. [Online]. Available: https://graphchallenge.mit.edu/champions
[2] NVIDIA Ampere Architecture In-Depth, 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
[3] Nvidia CUDA Graph, 2022. [Online]. Available: https://devblogs.nvidia.com/cuda-10-features-revealed/
[4] Nvidia Visual Profiler, 2022. [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler

[5] M. Bisson and M. Fatica, “A GPU implementation of the sparse deep neural network graph challenge,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2019, pp. 1–8.
[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 4171–4186.
[7] J. A. Ellis and S. Rajamanickam, “Scalable inference for sparse deep neural networks using Kokkos kernels,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2019, pp. 1–7.
[8] J. Gao, W. Ji, Z. Tan, and Y. Zhao, “A systematic survey of general sparse matrix-matrix multiplication,” 2020, *arXiv:2002.11273*.
[9] M. Grossman, C. Thiele, M. Araya-Polo, F. Frank, F. O. Alpak, and V. Sarkar, “A survey of sparse matrix-vector multiplication performance on large matrices,” 2016, *arXiv:1608.00636*.
[10] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–14.
[11] M. Hidayeto ğlu *et al.*, “At-scale sparse deep neural network inference with efficient GPU implementation,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2020, pp. 1–7.
[12] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, “Taskflow: A lightweight parallel and heterogeneous task graph computing system,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 6, pp. 1303–1320, Jun. 2022.
[13] Y. Huang *et al.*, “GPipe: Efficient training of giant neural networks using pipeline parallelism,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 103–112.
[14] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5MB model size,” 2016, *arXiv:1602.07360*.
[15] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, “Exploring hidden dimensions in parallelizing convolutional neural networks,” in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 2279–2288.
[16] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” in *Proc. Mach. Learn. Syst.*, 2019, pp. 1–13.
[17] J. Kepner, V. Gadepally, H. Jananthan, L. Milechin, and S. Samsi, “Sparse deep neural network exact solutions,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2018, pp. 1–8.
[18] J. Kepner and R. Robinett, “RadiX-Net: Structured sparse matrices for deep neural networks,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2019, pp. 268–274.
[19] J. Kepner *et al.*, “Sparse deep neural network graph challenge,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2019, pp. 1–7.
[20] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” 2014, *arXiv:1404.5997*.
[21] D.-L. Lin and T.-W. Huang, “A novel inference algorithm for large sparse neural network using task graph parallelism,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2020, pp. 1–7.
[22] X. Liu, J. Pool, S. Han, and W. J. Dally, “Efficient sparse-winograd convolutional neural networks,” in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–10.
[23] B. McCann, J. Bradbury, C. Xiong, and R. Socher, “Learned in translation: Contextualized word vectors,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6294–6305.
[24] D. Narayanan *et al.* “PipeDream: Generalized pipeline parallelism for DNN training,” in *Proc. ACM Symp. Oper. Syst. Princ.*, 2019, pp. 1–15.
[25] A. Parashar *et al.* “SCNN: An accelerator for compressed-sparse convolutional neural networks,” in *Proc. Annu. Int. Symp. Comput. Archit.*, 2017, pp. 27–40.
[26] A. Petrowski, G. Dreyfus, and C. Girault, “Performance analysis of a pipelined backpropagation parallel algorithm,” *IEEE Trans. Neural Netw.*, vol. 4, no. 6, pp. 970–981, Nov. 1993.
[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI blog*, 2019, p. 9.
[28] M. Wang, C.-C. Huang, and J. Li, “Unifying data, model and hybrid parallelism in deep learning via tensor tiling,” 2018, *arXiv:1805.04170*.
[29] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, “Balanced sparsity for efficient DNN inference on GPU,” in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 5676–5683.

Dian-Lun Lin received the BS degree from the Department of Electrical Engineering, Taiwan’s Cheng Kung University, Tainan, Taiwan, and the MS degree from the Department of Computer Science, National Taiwan University, Taipei, Taiwan. He is currently working toward the PhD degree with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, Utah. His research interests include parallel and heterogeneous computing.



Tsung-Wei Huang received the BS and MS degrees from the Department of Computer Science, National Cheng Kung University (NCKU), Tainan, Taiwan, in 2010 and 2011, respectively, and the PhD degree from the Electrical and Computer Engineering (ECE) Department, University of Illinois at Urbana-Champaign (UIUC), Champaign, Illinois. He is currently an assistant professor with the ECE Department, University of Utah. He has been building software systems for parallel computing and timing analysis.