A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism

Dian-Lun Lin
Dept. of Electrical and Computer Engineering
University of Utah
dian-lun.lin@utah.edu

Tsung-Wei Huang
Dept. of Electrical and Computer Engineering
University of Utah
tsung-wei.huang@utah.edu

***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 paper, 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.**

***Index Terms*—Sparse Neural Network, Task Graph Parallelism**

I. Introduction

Larger deep neural network (DNN) models have brought significant quality improvement to several fields, including natural language processing, speech recognition, and image classification [1]–[3]. 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 [4]–[6]. 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 [7]. Also, sparse DNN inference presents unique computational challenges from training, because the kernel efficiency largely depends on non-zero 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 [8]. Table I 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.

Neurons/Layers	120	480	1920	Bias	Size	Image 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

TABLE I
THE STATISTICS OF EACH DNN BENCHMARK IN THE CHALLENGE [8].

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]–[12]. 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 [13]–[16]. 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 [17]–[20]. The idea of the pipeline is simple and easy to implement, but it suffers from many performance problems, including synchronous execution, imbalanced pipeline stages, and limited pipeline depth.

As a consequence, we introduce SNIG[1], 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 [21], [22] to enable efficient decom-

[1]source code: https://github.com/dian-lun-lin/SNIG

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

position of model and data parallelisms. Our decomposition strategy transforms a partitioned inference workload into a *task dependency graph* that flows CPU-GPU operations naturally with the graph structure, providing improved scheduling efficiency and runtime asynchrony. Compared with pipeline-based frameworks, 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 [8]. SNIG is able to complete all DNNs using only one RTX 2080 Ti GPU of 11 GB memory, and we solve the largest DNN by 2.27× faster than the 2019 champion solution developed by Bisson and Fatica ("BF" method for brevity) [17]. Compared with a pipeline baseline inspired by GPipe [18], SNIG is faster at almost all networks (up to 2.19× speed-up) and scales better on multiple GPUs. We believe SNIG stands out as a unique inference engine for large sparse DNNs, given the ensemble of algorithm tradeoffs and decomposition decisions we have made.

II. Problem Formulation of Large Sparse DNN Inference

We target on the 2019 HPEC Sparse DNN Challenge, which is based on a mathematically well-defined DNN inference computation and can be implemented in any programming environment [8]. 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 [23]. 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 I 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 and discouraged.

III. State of the Art: The BF and Pipeline Methods

The BF method [17] is implemented with CUDA+OpenMP. Each GPU owns a part of the input matrix and computes the inference kernel iteratively by one OpenMP thread. At each iteration, each GPU executes two kernels, one for the inference and the other for calculating the non-empty row indices in the resulting matrix. After all GPUs complete execution, the OpenMP threads compute the new global list of non-empty rows and repartition the non-empty rows evenly among the GPUs. However, such a load-balancing method requires communication between CPUs and GPUs at each iteration, resulting in huge overhead. Also, to compute the list of non-empty rows, all GPUs need to be synchronized at each iteration. Synchronization can lead to unnecessary waiting time and waste computing power of GPUs. 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 [18] 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.

IV. SNIG

At a high level, SNIG describes the inference workload in a *task graph* comprising both data- and model-level parallelisms. Our task graph can scale to arbitrary sizes of DNN and input data under different numbers of GPUs. We develop an *efficient kernel* inside the task graph that computes only necessary entries during the inference iterations. Our in-kernel pruning strategy avoids unwanted computation incurred by sparsified network and data, in no need of additional CPU-GPU or GPU-GPU synchronization to redistribute input data among GPUs.

A. Task graph parallelism

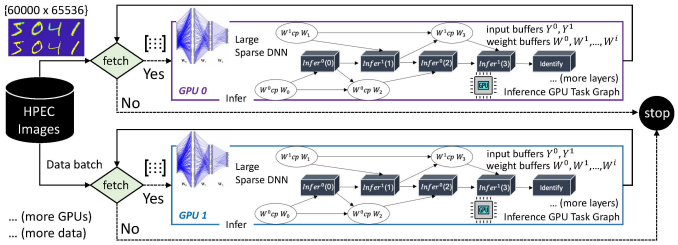

Fig. 1. Architecture of SNIG.

Figure 1 shows the overview of SNIG. SNIG defines the inference workload as a task dependency graph that iterates two stages: *fetch* and *infer*. At the fetch stage, a CPU task grabs a batch of input data of up to size $batch\_size$. 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 size of data. At the infer stage, a GPU task computes the inference of the batch on a GPU. Each GPU task consists of a *GPU task dependency graph* where each node represents one of the three GPU operations, host-to-device (H2D) copy, device-to-host (D2H) copy, and kernel tasks; each edge represents the dependency of two GPU operations. We leverage the power of modern *cudaGraph* [21] to offload a GPU task dependency graph using a single CPU call, thus reducing overheads. The architecture of SNIG is decentralized. There is no local or global CPU-GPU synchronization during the inference on a dataset.

We transpose weight matrices and store them using the Compressed Sparse Column (CSC) format. Since preloading all models to the GPU is impossible due to memory limit, we

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

only keep up to $num\_weights$ weight buffers (W^0, W^1, ..., $W^{num\_weights-1}$) on a GPU at a time. All weight buffers have the same size equal to the maximum size of W_l. More weight buffers result in a higher overlap between data communication and kernel computation. Since the inference at one layer only depends on the result from the previous layer, we allocate for each GPU two *result buffers* Y^0 and Y^1 each of size $batch\_size \times num\_neurons$ (number of neurons) to perform rolling swap for space optimization. Each buffer can be accessed via modulo operation on 2; $Infer^{l\%2}(l)$ represents applying the inference kernel to W_l using $Y^{l\%2}$ as input and $Y^{(l+1)\%2}$ as output. After completing the inference at the last layer, the GPU identifies the categories (predicted digits). Users can configure different $batch\_size$ and $num\_weights$ based on available GPU memory to fit arbitrary sizes of models and input data.

B. Inference kernel

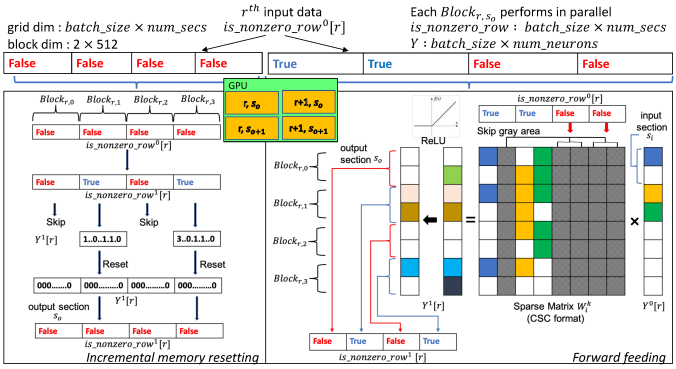

Fig. 2. Illustration of our inference kernel (Algorithm 1).

At the *infer* stage, our inference kernel consists of two parts: *forward feeding* and *incremental memory resetting*. Figure 2 illustrates one iteration of one row of input data in our kernel. To improve parallelism, we divide each input data into num_secs sections where each of sec_size is $num\_neurons/num\_secs$. Since each GPU keeps Y^0 and Y^1 to perform rolling swap, we allocate for each GPU two $batch\_size \times num\_secs$ boolean buffers, $is\_nonzero\_row^0$ and $is\_nonzero\_row^1$, to record whether a section of data contains nonzero elements. At the beginning of the inference kernel, we inspect each entry in $is\_nonzero\_row^0[r]$. If there exists at least one true value, meaning that there is at least one nonzero element in r^{th} input data, we enter *forward feeding*. The *forward feeding* performs matrix multiplication followed by ReLU and passes the result to the next layer via rolling swap. We skip input section s_i ($Y^0[r][k], sec\_size \times s_i \leq k < sec\_size \times (s_i + 1)$) that contains only zero entries indicated by $is\_nonzero\_row^0[r][s_i]$ to avoid unnecessary computation. During the matrix multiplication, we can further skip zero input entries. Taking advantage of rolling swap, we perform *incremental memory resetting* to reset buffers. If all entries in $is\_nonzero\_row^0[r]$ are false, we reset the output section s_o ($Y^1[r][k], sec\_size \times s_o \leq k < sec\_size \times (s_o + 1)$) including nonzero elements based on $is\_nonzero\_row^1[r][s_o]$. This largely avoids the overhead to reset the entire linear buffer for the next iteration to use. Our implementation computes each output section s_o of each data in parallel and calculates only necessary entries during inference iterations.

Algorithm 1 presents the details of our kernel. The grid dimension is $(batch\_size, num\_secs)$ and the block dimension is (2, 512). We allocate $4 \times sec\_size$ byte of external shared memory. The kernel is launched by $<<<(batch\_size, num\_secs)$, (2, 512), $4 \times sec\_size>>>$. Each block computes an output section s_o of one row of input data independently.

At the beginning, each block $Block_{r,s_o}$ in the grid determines to execute either *forward feeding* or *incremental memory resetting*. We use is_all_zero to record if all entries in $is\_nonzero\_row^0[r]$ are false (line 5-8). If is_all_zero is true, that is, all elements in $Y^0[r]$ are zero, $Block_{r,s_o}$ enters *incremental memory resetting*. During *incremental memory resetting*, if $is\_nonzero\_row^1[r][s_o]$ is true, $Block_{r,s_o}$ resets all elements of s_o to zero and toggles $is\_nonzeero\_row^1[r][s_o]$ to false (line 10-16). Otherwise, it returns directly (line 17).

$Block_{r,s_o}$ starts *forward feeding* if is_all_zero is false (line 19-52). Each block declares a shared memory array $results$ size of sec_size to store results (line 19) and initializes $results$ to the bias value directly (line 20-22). To avoid synchronization, we use a boolean array $is\_nonzero$ size of 2 to record whether $results$ has nonzero values (line 23-24, line 49). $Block_{r,s_o}$ iterates all input sections to compute results (line 26). If the current entry in $is\_nonzero\_row^0[r]$ is false, meaning that the current input section s_i contains only zero elements, we skip all elements in s_i directly (line 27-29). Otherwise, all threads along y dimension loop through all entries in s_i (line 31). We further skip to the next one if the current input value is zero (line 33-35). All threads along x dimension read col_w (line 36-37) and iterate the weight values and the weight row indices (line 38-40). To compute each s_o 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 nonzero index of the weight matrix. In line 36-37, we read column indices of the weight matrix via adding the offset. Then, we multiply each nonzero input entry with weight value and add the result to the corresponding location of $results$ (line 41).

After computing $results$, $Block_{r,s_o}$ loops through the $results$ (line 46). It computes ReLU, writes the result to each element in s_o, and sets $is\_nonzero[1]$ to true if there exists a nonzero result in s_o (line 47-49). Finally, we toggle $is\_nonzero\_row^1[r][s]$ to either true or false based on $is\_nonzero[1]$ (line 52).

V. Experimental Results

We evaluate SNIG's performance on the official MIT/IEEE/Amazon HPEC Sparse DNN Challenge Dataset [8]. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

Algorithm 1: Inference kernel

Input: col_w: array of column offsets of the weight matrix
Input: row_w: array of row indices
Input: val_w: array of values

$r \leftarrow$ **block**.x
$s_o \leftarrow$ **block**.y
$tid \leftarrow$ **thread**.y * **blockDim**.x + **thread**.x
$num\_threads \leftarrow$ **blockDim**.x * **blockDim**.y
$is\_all\_zero \leftarrow$ **true**
for $s_i \leftarrow 0$; $s_i < num\_secs$; $++s_i$ **do**
  is_all_zero &= $!is\_nonzero\_row^0[r][s_i]$
end
if $is\_all\_zero == true$ **then**
  if $is\_nonzero\_row^1[r][s_o] == true$ **then**
    for $j \leftarrow tid$; $j < sec\_size$; j += $num\_threads$ **do**
      $Y^1[r][sec\_size * s_o + j] = 0$
    end
    __syncthreads()
    $is\_nonzero\_row^1[r][s_o] \leftarrow$ **false**
  end
  return
end
extern **__shared__** $results[]$
for $k \leftarrow tid$; $k < sec\_size$; k += $num\_threads$ **do**
  $results[k] \leftarrow bias$
end
__shared__ $is\_nonzero[2]$
$is\_nonzero[1] \leftarrow$ **false**
__syncthreads()
for $s_i \leftarrow 0$; $s_i < num\_secs$; $++s_i$ **do**
  if $!is\_nonzero\_row^0[r][s_i]$ **then**
    continue
  end
  $j \leftarrow$ **thread**.y + $s_i * sec\_size$
  for j; $j < (s_i + 1) * sec\_size$; j += **blockDim**.y **do**
    $y^{val} \leftarrow Y^0[r][j]$
    if $y^{val} == 0$ **then**
      continue
    end
    $w^- \leftarrow col\_w[s_o * num\_nurons + j]$ + **thread**.x
    $w^+ \leftarrow col\_w[s_o * num\_neurons + j + 1]$
    for $k \leftarrow w^-$; $k < w^+$; k += $blockDim.x$ **do**
      $w^{row} \leftarrow row\_w[k]$
      $w^{val} \leftarrow val\_w[k]$
      atomicAdd(&$results[w^{row}$ - $s_o * sec\_size]$, $y^{val} * w^{val}$)
    end
  end
end
__syncthreads()
for $i \leftarrow tid$; $i < sec\_size$; i += $num\_threads$ **do**
  $v \leftarrow$ min(32, max($results[i]$, 0))
  $Y^1[r][s_o * sec\_size + i] \leftarrow v$
  $is\_nonzero[v \neq 0] \leftarrow$ **true**
end
__syncthreads()
$is\_nonzero\_row^1[r][s_o] = is\_nonzero[1]$

64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz, 4 GeForce RTX 2080 Ti GPUs with 11 GB memory, and 256 GB RAM. We compiled all programs using Nvidia CUDA nvcc 10.1 on a host compiler of GNU GCC-8.3.0 with C++14 standards `-std=c++14` and optimization flags `-O2` enabled. All data is an average of ten runs with `float` type.

A. Baseline

We consider BF and GPipe* methods for our baseline. The BF method is the champion solution of the 2019 HPEC Sparse DNN Challenge [17]. 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. We implemented the GPipe* method based on GPipe [18]. 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. For fair purposes, the inference kernel inside the pipeline is the same as SNIG. We configure the block dimension of all kernels to 2×512 , the batch size of input data to 5000 for SNIG and GPipe* , and the number of weight buffers to 2 for SNIG. We will discuss the effect of different parameters in the later section.

B. Performance Comparison

Table II compares the overall inference rate and runtime performance between SNIG, BF, and GPipe* using one, two, 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 4 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.

Figure 3 plots the scalability over increasing number of GPUs. Our runtime scales the best among the three methods. In the 16384×1920 scenario, SNIG speeds up BF by 1.7×, 1.8× , 1.7×, and 1.8× at 1, 2, 3, and 4 GPUs, respectively. In the 65536×1920 scenario, SNIG speeds up GPipe* by 1.9×, 2.1× , 2.0× at 2, 3, and 4 GPUs, respectively. We attribute this to the synchronization overhead of both methods (BF at each iteration, GPipe* at each pipeline stage). Figure 4 plots the scalability over increasing number of neurons. SNIG

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

		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)

TABLE II

OVERALL INFERENCE RATE (GIGAEDGES PROCESSED PER SECOND) AND RUNTIME PERFORMANCE (SECONDS) OF SNIG, BF, AND GPIPE* ACROSS ONE, TWO, THREE, AND FOUR GPUS. 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 [8]. SINCE THE GPIPE* METHOD IS STAGED ON THE NUMBER OF GPUS, WE DO NOT REPORT ITS RUNTIME UNDER ONE GPU.

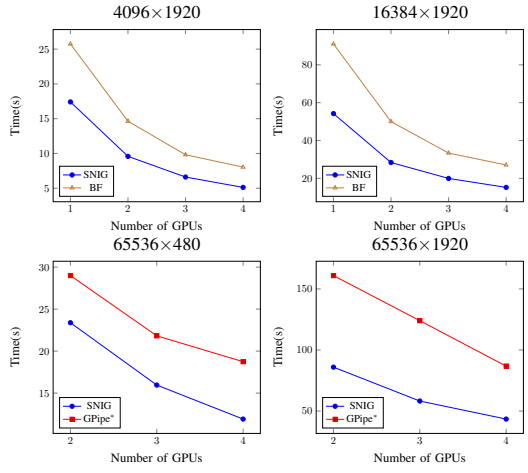

Fig. 3. Execution time with different numbers of GPUs.

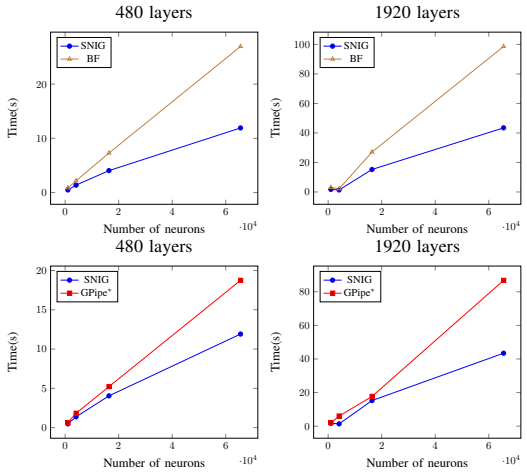

Fig. 4. Execution time with different neurons under 4 GPUs.

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. Figure 5 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.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

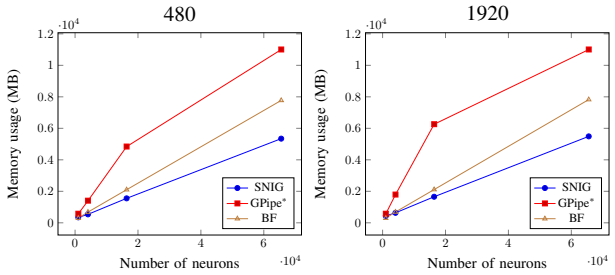

Fig. 5. Peak GPU memory usage under 4 GPUs.

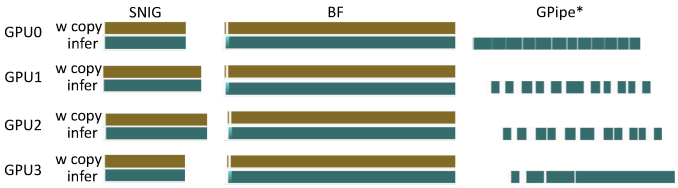

Fig. 6. Execution timeline of each method on completing 65536 neurons and 1920 layers under 4 GPUs.

Figure 6 plots a partial GPU execution timeline of each method using the data extracted from NVIDIA Visual Profiler [24] 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 1 and GPU 2.

C. Parameter Sensitivity

Figure 7 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. Figure 8 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.

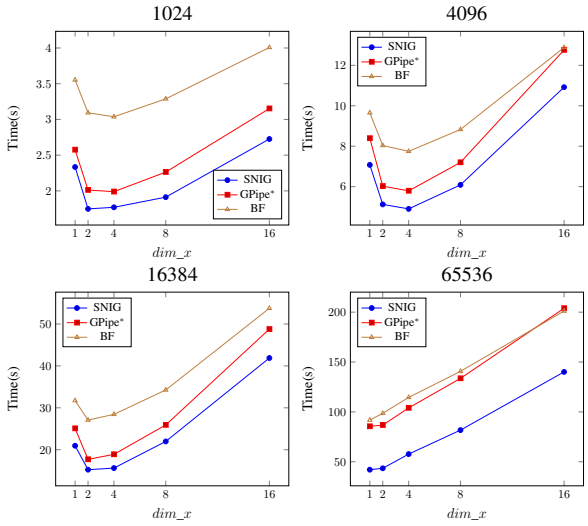

Fig. 7. Execution time with different block dimensions (dim_x, dim_y) on 1920 layers under 4 GPUs. The total number of threads $dim\_x \times dim\_y$ remains 1024.

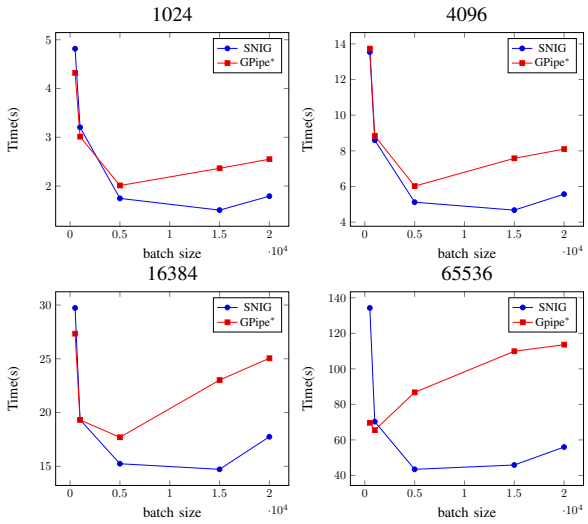

Fig. 8. Execution time with different batch sizes on 1920 layers under 4 GPUs.

VI. 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. Our in-kernel pruning strategy avoids unwanted computation incurred by sparsified network and data, in no need of additional CPU-GPU synchronization to repartition data. With 4 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).

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

REFERENCES

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *ACL*, 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[2] B. McCann, J. Bradbury, C. Xiong, and R. Socher, "Learned in Translation: Contextualized Word Vectors," *CoRR*, vol. abs/1708.00107, 2017. [Online]. Available: http://arxiv.org/abs/1708.00107

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2018.

[4] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149*, 2015. [Online]. Available: https://arxiv.org/abs/1510.00149

[5] 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," *arXiv:1602.07360*, 2016.

[6] J. Kepner, V. Gadepally, H. Jananthan, L. Milechin, and S. Samsi, "Sparse Deep Neural Network Exact Solutions," in *IEEE HPEC*, pp. 1–8.

[7] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-TensorFlow: Deep Learning for Supercomputers," in *NIPS*, 2018, pp. 10 414–10 423.

[8] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," *IEEE HPEC*, 2019. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2019.8916336

[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," *CoRR*, vol. abs/1404.5997, 2016. [Online]. Available: http://arxiv.org/abs/1608.00636

[10] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient Sparse-Winograd Convolutional Neural Networks," in *ICLR*, 2018. [Online]. Available: https://openreview.net/forum?id=HJzgZ3JCW

[11] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks," in *ACM ISCA*, 2017, p. 27–40. [Online]. Available: https://doi.org/10.1145/3079856.3080254

[12] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced Sparsity for Efficient DNN Inference on GPU," in *AAAI*, 2019.

[13] J. A. Ellis and S. Rajamanickam, "Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels," in *IEEE HPEC*, 2019, pp. 1–7.

[14] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks," *CoRR*, vol. abs/1802.04924, 2018. [Online]. Available: http://arxiv.org/abs/1802.04924

[15] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[16] M. Wang, C.-C. Huang, and J. Li, "Unifying Data, Model and Hybrid Parallelism in Deep Learning via Tensor Tiling," *CoRR*, vol. abs/1805.04170, 2018. [Online]. Available: http://arxiv.org/abs/1805.04170

[17] M. Bisson and M. Fatica, "A GPU Implementation of the Sparse Deep Neural Network Graph Challenge," in *IEEE HPEC*, 2019, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8916223

[18] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *NIPS*, 2019, pp. 103–112.

[19] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *ACM SOSP*, 2019, p. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[20] A. Petrowski, G. Dreyfus, and C. Girault, "Performance analysis of a pipelined backpropagation parallel algorithm," *IEEE Transactions on Neural Networks*, vol. 4, no. 6, pp. 970–981, 1993.

[21] Nvidia CUDA Graph, https://devblogs.nvidia.com/cuda-graphs/.

[22] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," 2019, pp. 974–983.

[23] J. Kepner and R. Robinett, "RadiX-Net: Structured Sparse Matrices for Deep Neural Networks," in *IEEE IPDPS Workshops*, 2019, pp. 268–274.

[24] Nvidia Visual Profiler, https://developer.nvidia.com/nvidia-visual-profiler.

APPENDIX A
BF METHOD WITHOUT NVLINK

To achieve load balancing, the BF method partitions the data based on nonzero entries evenly across all GPUs at each iteration (i.e., layer) and relies on NVLink to transfer data using unified addressing. However, this method is very time-consuming without NVLink, i.e., about 10× slower according to our offline experiment. Since we do not have NVLink, we partition the data evenly across all GPUs at the beginning and do not repartition them during the iterations. 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 Figure 9 below, the difference of the number of nonzero rows at each GPU is within 350 rows (<0.5% of the total rows) across first 15 iterations.

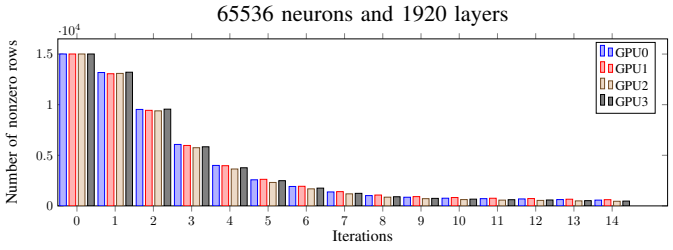

Fig. 9. Number of nonzero rows in 15 iterations on 4 GPUs using BF without NVLink

978-1-7281-9219-2/20/$31.00 ©2020 IEEE