Efficient Inference on GPUs for the Sparse Deep Neural Network Graph Challenge 2020

Mert Hidayetoğlu, Carl Pearson, Vikram Sharma Mailthody,
Eiman Ebrahimi[‡], Jinjun Xiong[§], Rakesh Nagi, Wen-mei W. Hwu
University of Illinois at Urbana-Champaign, [‡]NVIDIA, [§]IBM Research
{hidayet2, pearson, vsm2}@illinois.edu, eebrahimi@nvidia.com, jinjun@us.ibm.com, {nagi,w-hwu}@illinois.edu

***Abstract*—This paper presents GPU performance optimization and scaling results for the Sparse Deep Neural Network Challenge 2020. Demands for network quality have have increased rapidly, pushing the size and thus the memory requirements of many neural networks beyond the capacity of available accelerators. Sparse deep neural networks (SpDNN) have shown promise for reigning in the memory footprint of large neural networks. However, there is room for improvement in implementing SpDNN operations on GPUs. This work presents optimized sparse matrix multiplication (SpMM) kernels fused with the ReLU function. The optimized kernels reuse input feature maps from the shared memory and sparse weights from registers. For multi-GPU parallelism, our SpDNN implementation duplicates weights and statically partition the feature maps across GPUs. Results for the challenge benchmarks show that the proposed kernel design and multi-GPU parallelization achieve up to 180 TeraEdges per second inference throughput. These results are up to 4.3× faster for a single GPU and an order of magnitude faster at full scale than those of the champion of the 2019 Sparse Deep Neural Network Graph Challenge for the same generation of NVIDIA V100 GPUs. Using the same implementation, we also show single-GPU throughput on NVIDIA A100 is 2.37× faster than V100.**

I. Introduction

Deep learning (DL) has seen great progress over the last decade and demonstrated substantial accuracy improvement over a range of machine learning tasks such as image classification [1], object recognition [2], language modeling [3], [4], and language translation [5]. Scaling up DL models along with increased amount of training data have proven to be an effective approach to improve the model accuracy [3], [4], [6]. The progress of DL applications involves not only the algorithmic improvements, but also the unprecedented computational throughput provided by GPUs.

On several machine learning tasks, the size of state-of-the-art deep neural networks (DNN) has grown beyond the memory limits of available accelerators [3], [6]. To address this, the DL community has taken an algorithmic approach of sparsifying the DNN using techniques such as pruning [7]–[9]. These optimization convert a dense DNN into a sparse DNN.

Sparse DNNs present unique scalablity challenges. Realizing this, in 2019 MIT/IEEE/Amazon proposed Sparse DNN graph challenge as an extension to the Graph Challenge [10]–[14]. The Sparse DNN Challenge is created by leveraging the collective knowledge of machine learning, high-performance computing and graph analytics communities on emerging AI systems. The objective of the sparse DNN graph challenge is to build scalable algorithms and systems for sparse AI analytics.

The sparse DNN challenge provides model structure, trained model weights, and inputs as a foundation for comparison of inference performance. Several models with varying number of layers (120 to 1920 layers) and neuron connections (4 Million to 4 Billion) are posted in the challenge. Such a large variation allows system and algorithm designers to evaluate efficient, scalable implementations against a variety of network types.

In this work, we propose efficient sparse algorithms and demonstrate performent kernel implementations for sparse DNN inference on GPUs. GPUs have become a de-facto accelerator for DNN computation. However, traditionally GPUs are designed for *dense* DNN operations and their performance is considerably reduced for *sparse* DNN computations. Using a baseline sparse DNN implementation, we identify the causes of performance bottleneck. In our baseline implementation, we store the weights as sparse matrices in the compressed sparse row (CSR) storage format while the inputs are represented as dense matrices in the column-major data layout in memory. We observe that irregular memory accesses to the input matrix and redundant accesses to the weight matrices are the primary causes for inefficiency in the naive sparse DNN implementation.

To address this, we propose a few optimizations, namely, register tiling, shared-memory tiling, and compact index representation to create optimized fused kernel implementation for sparse DNN computation. In the optimized fused kernel, the weights are stored in sliced ELLPACK format for efficient memory access while the inputs are retained in the column-major layout in memory. Together these optimizations minimize the total number of irregular memory access and redundant access to the global memory, thus providing significant performance improvements. Although variants of these optimizations have been applied in SpMM kernels [15], in this work we tune them for sparse DNN computation.

Our evaluation shows, compared to the baseline fused kernel implementation, the optimized fused kernel provides up to 11.84× speed up. Overall, the proposed optimizations achieve up to 14.30 TeraEdges (10^{12} edges) per second performance on a single V100 GPU. Compared with prior champions, our single GPU implementation on NVIDIA V100 GPU is up to 4.3× faster. Compared to an implementation based off of NVIDIA's cuSPARSE, our proposed kernels can provide 200× speedup. In addition, we evaluate the performance of our optimized fused kernel on the latest generation GPU, Ampere A100, and show that out-of-the box execution of our optimized

kernel can achieve up to 20.99 TeraEdges per second (2.37× faster than V100 GPU).

To measure the algorithm scalablity, we perform strong scaling using batch parallelism and scale up to 768 V100 GPUs on Summit at Oak Ridge National Lab. On a single node, when the number of V100 GPUs are increased from one to six, optimized fused kernel achieves up to 5.37× speed up, thus providing a scaling efficiency of 89.5% on a node. Our evaluation shows that, our proposed implementation can provide up to 46.8× speed up when the number of GPUs is increased to 768 (or 128 nodes with 6 GPUs in each node) compared to single V100 GPU implementation. Compared to 2019 Sparse DNN Champions, our scale-out implementation is more than 10× faster, providing up to 180 TeraEdges per second.

Overall, we make the following main contributions:

1) We present novel SpMM algorithms that exploit data reuse during DNN inference. The performance of the algorithms is further enhanced by integrating the ReLU layer.
2) We perform extensive benchmarking on OLCF Summit as well as the new A100 GPUs for the Sparse Deep Neural Network Challenge 2020 dataset.

The rest of the paper is organized as follows. We provide a brief background on sparse DNN graph challenge and discuss the limitations of naive sparse DNN inference implementation in § II. We discuss the proposed single GPU optimizations for sparse DNN inference in § III. We discuss our results in § IV and conclude in § V.

II. Background & Motivation

In this section, we provide a brief overview of the sparse DNN challenge and discuss the limitations in naive sparse DNN inference implementation.

A. Overview of Sparse DNN Challenge

The Sparse DNN graph challenge specifies a collection of large sparse DNNs models [10], [11]. that are representative of the latest trends in addressing challenging machine learning tasks. The challenge provides model structure (number of layers and size of layer), and model weights for computing sparse DNN inference on a given input dataset:

1) Formulation of Sparse Layer: Each layer in the sparse DNN challenge can be described with Equation (1) [10].

$$\boldsymbol{Y}_{l+1} = \text{ReLU}(\boldsymbol{W}_l \times \boldsymbol{Y}_l + \boldsymbol{B}) \quad (1)$$

where l is the layer number, L is the total number of layers, $\boldsymbol{Y}_l$ and $\boldsymbol{Y}_{l+1}$ are $N \times M$ matrices of M input features of length N, respectively, $\boldsymbol{W}_l$ is an $N \times N$ matrix of activation weights, $\boldsymbol{B}_l$ is an $N \times M$ bias matrix for each output, and ReLU is the activation function. Here, ReLU activation function promotes sparsity and is defined as $\text{ReLU}(x) = \max\{0, \min\{x, 32\}\}$. The weight matrix represents a general pattern of neuron connections: for a fully-connected layer, $\boldsymbol{W}_l$ would be entirely non-zero, for a convolution layer, $\boldsymbol{W}_l$ would be banded, and for a general sparse layer, $\boldsymbol{W}_l$ is a general sparse matrix.

2) Steps Involved in Sparse DNN Challenge: Algorithm 1 describes the high-level steps involved in computing sparse DNN inference in the sparse DNN graph challenge [10], [16]. The challenge provides datasets comprising of input data for the neural network, weights for each layer in the network, bias values for each layer and finaly the ground truth to validate if the results are correct while computing inference.

Algorithm 1 Outline of Sparse DNN Challenge Algorithm

1: Read input $\boldsymbol{Y}_0$ and model weights ($\boldsymbol{W}_0$,$\boldsymbol{W}_1$ ... $\boldsymbol{W}_{L-1}$) from binary files
2: Initialize bias vector $\boldsymbol{B}$ with a constant
3: Evaluate Equation (1) for all the layers, starting from $l = 0$ and until $l = L - 1$
4: Use $\boldsymbol{Y}_L$ to determine categories and compare with the ground truth.
5: Measure and report performance

The input to the neural network is an interpolated sparse version derived from the MNIST dataset and consists of 60,000 images stored in TSV format. Each of the MNIST 28×28 pixel images are resized to 32×32 (1024 neurons), 64×64 (4096 neurons), 128×128 (16384 neurons) and 256×256 (65536 neurons) pixels. All resized images are thresholded to ensure values reside between 0 to 1. Then, they are linearized and stacked together to obtain 60k1K, 60k4K, 60k16K, and 60k64K input feature matrices. This allows to store each image in a single row in the TSV while each column representing a non-zero pixel location with a value of 1. The challenge also provides three different sparse DNN models comprising of 120, 480, and 1920 layers. Thus effectively creates a total of 12 DNNs ($\{1024, 4096, 16384, 65536\}$ neurons × $\{120, 480, 1920\}$ layers).

Since such large sparse DNN networks are not publicly available, the challenge generates the weight matrix using RadiX-Net synthetic sparse DNN generator [8]. The synthetic generator is capable of creating pre-determined DNNs with 32 connections per neuron and ensuring there exists an equal number of paths between inputs and intermediate layers. All neurons have weights and biases set to 1/16. However, our kernel design and implementation could work on real-life data with any arbitrary bias values and any sparse layer with arbitrary sparsity pattern.

B. The Baseline GPU Implementation

We present a baseline implementation of sparse DNN using GPUs and discuss the bottlenecks to motivate our optimizations. The baseline implementation consists a sparse×dense SpMM kernel fused with ReLU function for GPU execution. In this implementation, shown in Listing 1, each thread produces a single output element in $\boldsymbol{Y}_{l+1}$, and gathers data from the corresponding column of $\boldsymbol{Y}$. Each thread reads a row of `windex` (column indices of non-zero weight elements in the CSR format) and `wvalue` (the value of the non-zero weight elements) from sparse $\boldsymbol{W}_l$ matrix stored in CSR format, as depicted in Figure 1. Note that `windex` and `wvalue` have

the same layout and access patterns so we omitted `wvalue` from Figure 1 Also, each `wdispl` element gives the starting location of its corresponding row of non-zero elements in the CSR format. That is, all `windex` elements in Figure 1 are stored in a linear array with the starting point of each row of non-zero elements delineated by the `wdispl` elements.

Listing 1: Baseline Implementation

```
1  __device__ float __ReLU(float x){
2    return x<0.0?0.0:x>32.0?32.0:x;
3  };
4  __global__ void fused_ReLU(float *yout, float *yin, int neuron,
5   int *wdispl, int *windex, float *wvalue, float *bias, int *active,
6   int *category){
7    int xoff = blockIdx.x*blockDim.x+threadIdx.x;
8    int yoff = category[blockIdx.y]*neuron;
9    float acc = 0;
10   for(int n = wdispl[xoff]; n < wdispl[xoff+1]; n++)
11     acc += yin[yoff+windex[n]]*wvalue[n];
12   acc = __ReLU(acc+bias[xoff]);
13   if(acc > 0){
14     yout[blockIdx.y*neuron+xoff] = acc;
15     atomicAdd(active+blockIdx.y,1);
16   }
17 }
18
19 //INFERENCE LOOP AT HOST CPU
20 for(int l = 0; l < layer; l++){
21   dim3 grid(neuron/blocksize,mybatch);
22   dim3 block(blocksize);
23   cudaMemset(active_d,0,sizeof(int)*mybatch);
24   fused_ReLU<<<grid,block>>>(nextfeat_d,currfeat_d,neuron,
25    wdispl_d[l],wind_d[l],wval_d[l],bias_d,active_d,categories_d);
26   cudaMemcpy(active,active_d,sizeof(int)*mybatch,D2H);
27   int feature = 0;
28   for(int k = 0; k < mybatch; k++){
29     if(active[k]){
30       globalcategories[feature] = globalcategories[k];
31       categories[feature] = k;
32       feature++;
33     }
34   }
35   cudaMemcpy(categories_d,categories,sizeof(int)*feature,H2D);
36   mybatch = feature;
37   FEATPREC *tempfeat_d = currfeat_d;
38   currfeat_d = nextfeat_d;
39   nextfeat_d = tempfeat_d;
40 }
```

For simplicity in the figure, we assume a toy example where each block consists of four threads and the warp size is two. As a result, the GPU kernel deploys $M \times N/B$ thread blocks, where M is the number of output feature elemnts (pixels), and B is the block size. Here, each block accesses a portion of $\boldsymbol{W}_l$. As each thread accesses the input, it performs a fused multiply-and-add (FMA) operation with the corresponding activation weight, accumulating the result in its register. Finally, each thread adds the bias and activate (or deactivate) the output, and write the output to global memory.

While the baseline implementation exhibits a high degree of thread-level parallelism for large output features, it is inefficient for several reasons. First, the global memory accesses to the input matrix are not only uncoalesced, but also irregular: Figure 1 highlights the memory accesses of the first thread of the first block. An iregular subset of the $\boldsymbol{Y}_l$ elements are read by each thread. Second, threads that generate equivalent elements in different output maps redundantly reads the same row of the $\boldsymbol{W}_l$ matrix. As a result the weight matrix is read M times. This results in wasting memory bandwidth that could have been used for the needed computation. Third, the load imbalance among threads caused by different number of non-zero elements in adjacent rows of $\boldsymbol{W}_l$ yields thread divergence within warps. We address these limitation in the next section.

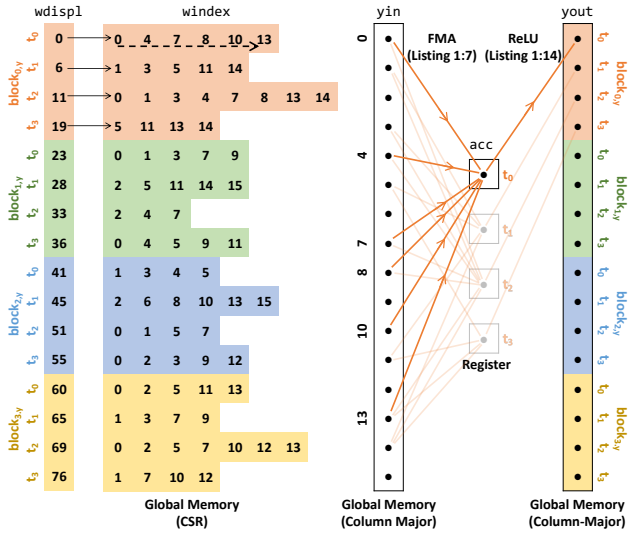

Fig. 1: Baseline fused kernel implementation. Each thread irregularly accesses the input features and accumulates the output in register.

III. Proposed Kernel Design

In this section we discuss in detail the three performance optimizations and two memory optimizations that we propose to improve the inference rate of the baseline implementation. These optimizations involve register tiling, shared memory tiling, efficient access to weight matrix $\boldsymbol{W}_l$, compact index representation and out-of-core storage of weight matrix $\boldsymbol{W}_l$. Listing 2 shows the optimized-fused kernel implementation for sparse DNN computation. Next we will describe each of these optimization in detail.

A. Performance Optimizations

1) Register Tiling: The most obvious inefficiency of the baseline fused kernel (shown in Listing 1, line 11) is the duplicated access to the weight matrix across output features. To provide reuse of the weight matrix $\boldsymbol{W}_l$ from register, the optimized kernel groups (`MINIBATCH`) multiple active features and performs the ReLU computation for all of them in a single step. This reuses the index (`windex`) and value (`wvalue`) elements in each thread for `MINIBATCH` features as shown in Listing 2, lines 23 and 24, where `MINIBATCH` is the number of features in each minibatch.

When `MINIBATCH` is increased, the register data reuse improves, thus increasing the arithmetic intensity. This saves memory bandwidth for reading the weight matrices, but also increases register useage. A `MINIBATCH` value of 12 is selected for optimal performance, which balances increased reuse with memory spills from increased register pressure.

2) Shared-Memory Tiling: We now address the duplicate and irregular accesses to input features. Each input feature element is potentially accessed multiple times by multiple threads and they are accessed in an irregular patterns controlled by the weight sparsity pattern expressed by `windex`. This wastes memory bandwidth and increases access latency. For example, for $block_{0,y}$ in Figure 1, both t_0 and t_2 access `yin[4]` but at different iterations for their execution.

Listing 2: Optimized Kernel Implementation

```
1  __global__ void opt_ReLU(float *yout, float *yin, int neuron,
2   int *buffdispl, int *mapdispl, unsigned short *map,
3   int *wdispl, unsigned short *windex, float *wvalue,
4   float *bias, int *active, int *category){
5    __shared__ float shared[BUFFSIZE];
6   int wind = threadIdx.x%WARPSIZE;
7   int yoff = blockIdx.y*MINIBATCH;
8   int xoff = blockIdx.x*blockDim.x+threadIdx.x;
9   float acc[MINIBATCH] = {0.0};
10  for(int buff = buffdispl[blockIdx.x];
11      buff < buffdispl[blockIdx.x+1]; buff++){
12    int mapnz = mapdispl[buff+1]-mapdispl[buff];
13    for(int n = threadIdx.x; n < mapnz; n += blockDim.x){
14      int ind = map[mapdispl[buff]+n];
15      for(int f = 0; f < MINIBATCH; f++){
16        shared[f*buffsize+n] = yin[category[yoff+f]*neuron+ind];
17    }
18    __syncthreads();
19    int warp = (buff*blockDim.x+threadIdx.x)/WARPSIZE;
20    for(int m = wdispl[warp]; m < wdispl[warp+1]; m++){
21      int ind = windex[m*WARPSIZE+wind];
22      float val = wvalue[m*WARPSIZE+wind];
23      for(int f = 0; f < MINIBATCH; f++)
24        acc[f] += shared[f*buffsize+ind]*val;
25    }
26    __syncthreads();
27  }
28  for(int f = 0; f < MINIBATCH; f++){
29    acc[f] = __ReLU(acc[f]+bias[xoff]));
30    if(acc[f] > 0){
31      yout[(yoff+f)*neuron+xoff] = acc[f];
32      atomicAdd(active+blockIdx.y*MINIBATCH+f,1);
33    }
34  }
35 }
```

To address this inefficiency, we stage irregular accesses through shared memory. This is supported by preprocessing the `windex` rows for each thread block and build a preloading list (`map`) of all the `yin` elements collectively accessed by all the threads in the thread block. For example, the preload list for $block_{0,y}$ in Figure 1 would be [0,1,3,4,5,7,8,10,11,13,14].

During execution, the thread block will collect these elements into consecutive entries of a `buffer` array in the shared memory. For example, `yin[0]` will be loaded into `buffer[0]`, `yin[1]` into `buffer[1]`, `yin[3]` into `buffer[2]`, and so on. The required tiling data structures are constructed once prior to inference and are reused from global memory for computation of all features.

The second part of the preprocessing step is to update the `windex` so that the stored indices become indicies into the `buffer`. For example, `windex` for $t_{0,y}$ will be updated from [0, 4, 7, ...] to [0, 3, 5, ...]. During kernel execution, all threads of a thread block cooperatively collect all the needed `yin` elements into its shared memory.

When input feature access footprint of a thread block is larger than the shared-memory size, the irregular accesses are performed in multiple stagings, as illustrated in Figure 2(a). Assume that the shared memory can only accommodate six `yin` elements for each thread block, the footprint for $block_{0,y}$ will be divided into two stages. The footprint of each stage is mapped into the `buffer` array independently. Thus, for $block_{0,y}$, `yin[8]`, `yin[10]`, and `yin[13]` that are accessed in the second stage are mapped into `buffer[0]`, `buffer[1]`, `buffer[3]`, as illustrated in Figure 2(b).

For each stage, Listing 2, line 12−18, shows loading of feature data to shared memory using the `map` data structure. Then feature data is accessed irregularly from shared memory at line 24.

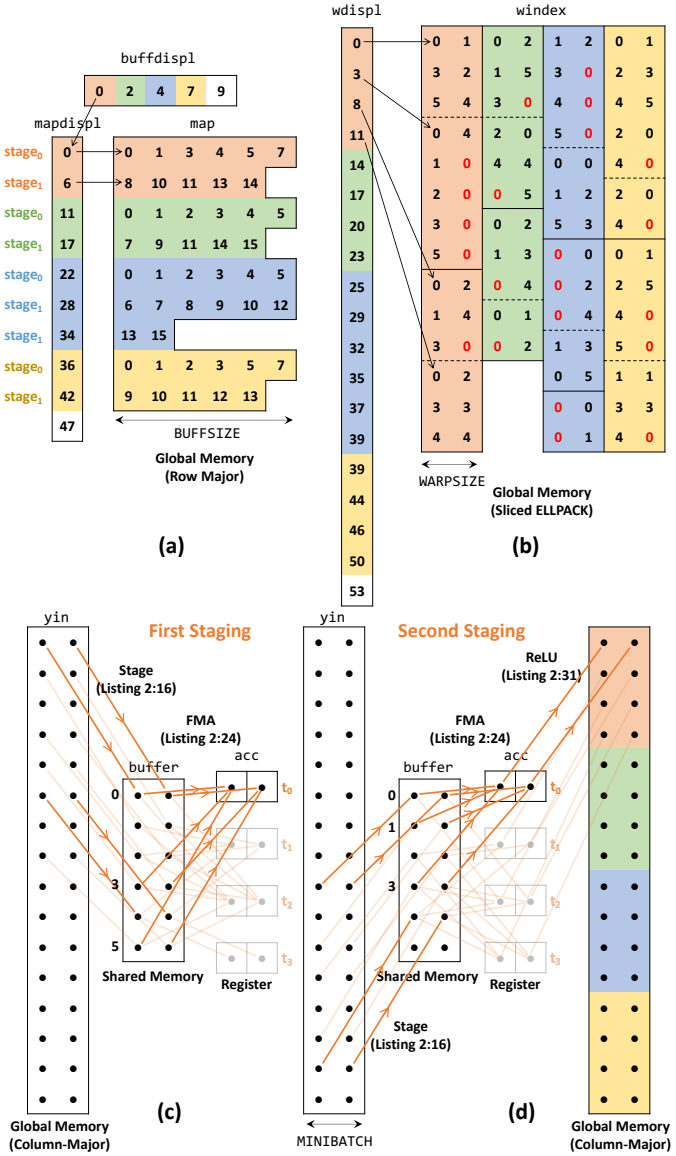

Fig. 2: Optimized-fused kernel execution example.

3) Efficient Access to Weight Matrix: Even though the baseline CSR storage is efficient in terms of memory footprint, the access is not coalesced among threads in each warp. To address this, we store the weight matrix in a transposed sliced-ELL storage format with zero-padding in warp granularity. Figure 2 shows the `wdispl` and `windex` data structures corresponding to the CSR representation shown in Figure 1.

The first two columns of `windex` (in orange) in Figure 2(b) shows the layout of the elements for $block_{0,y}$. The top two sections separated by the dashed line each contains the `windex` elements accessed by the a warp of the block during the first stage of execution. The bottom two sections are accessed by the same two warps during their second stage of execution. The `wdispl` elements marks the positions of the dash lines and solid lines for every warp and every thread block.

The padded zeros are highlighted with red font color. The dashed lines represents the boundary between warps (each block has two warps in this example) and solid lines represent

TABLE I: Inference Throughput (TeraEdges/Second)

Neurons	Layers	Single V100	Single A100	Number of V100 GPUs (Six per Node)								
				3	6	12	24	48	96	192	384	768
1024	**120**	10.51 (0.225s)	16.74 (1.59×)	18.92	22.46	25.52	28.52	27.77	**29.17**	27.89	29.12	29.13
	480	12.87 (0.073s)	20.99 (1.63×)	21.47	24.34	26.92	28.73	28.43	**29.30**	28.80	29.10	23.06
	1920	14.30 (0.264s)	20.68 (1.45×)	22.26	24.77	27.33	28.70	28.58	28.60	28.73	**28.83**	28.83
4096	**120**	9.45 (0.100s)	14.27 (1.51×)	20.69	31.36	47.82	62.03	70.31	75.81	79.11	81.13	**82.20**
	480	11.74 (0.322s)	18.63 (1.59×)	28.18	40.58	56.54	67.63	73.16	77.27	80.02	79.97	**82.22**
	1920	13.88 (1.08s)	19.86 (1.43×)	30.53	44.48	62.74	72.57	73.72	76.25	79.99	80.67	**82.32**
16384	**120**	6.15 (0.614s)	11.60 (1.89×)	16.31	28.85	50.74	64.33	89.18	111.44	**146.88**	114.87	111.30
	480	7.45 (2.027s)	14.31 (1.92×)	19.82	32.88	50.83	71.45	95.78	112.61	138.62	138.30	**139.44**
	1920	7.84 (7.704s)	15.27 (1.95×)	20.86	33.62	57.08	77.73	104.83	120.63	146.11	146.30	**146.40**
65536	**120**	3.47 (4.352s)	8.15 (2.35×)	10.90	18.77	34.20	51.14	73.67	100.72	162.19	173.25	**179.58**
	480	3.83 (15.769s)	9.08 (2.37×)	12.13	20.39	37.63	56.66	75.29	108.06	166.15	**170.26**	169.30
	1920	3.93 (61.474s)	9.33 (2.37×)	12.47	20.88	38.81	58.08	77.55	112.01	167.43	170.06	**171.37**

the boundaries between buffer stages, i.e., all blocks except block 2 involve two stagings and block 3 involves three stagings. In this example, the zero padding overhead is 27.5% for warp granularity, however, it would be 80% and 100% with zero padding in tile and layer granularity, respectively. Warp-level padding introduces a small number of zeros while maintaining coalesced (efficient) memory access.

B. Memory Optimizations

1) Out-of-Core Storage and Overlapping Strategy: Data parallelism provides good scaling in an embarrassingly parallel fashion. However, duplicating weights in each GPU can make large networks infeasible for GPUs with limited memory capacity. As a remedy, we implement an out-of-core storage algorithm that loads the required weight data structures to GPU memory from CPU memory when needed. Even though the out-of-core algorithm saves significant amount of memory, it has a data transfer overhead. We address this overhead by hiding the data transfer behind the GPU kernel with a double-buffering and overlapping strategy.

Double-buffering involves a pair of buffers in the GPU memory. While layer l uses weights from one buffer, weights for layer $l+1$ are moved into the other buffer. When layer l and the copy are both finished, the buffer pointers are swapped and the same procedure is followed for the next layer. The data transfers are completely hidden behind the inference computation in our implementation.

2) Compact Representation and Batching: In order to reduce the memory footprint, we store the `map` and `windex` data structures with two-byte `unsigned short` data type. That reduces the memory footprint (and hence the data transfer time) by approximately 33%.

We create batches for inputs to further reduce the memory consuming from output features during inference computation. Batching has no significant overhead since we overlap transfer of weights during the GPU computation as discussed in § III-B1. As a result, we can fit even the largest inference problem in a single V100 GPU with 16 GB memory.

IV. Evaluation

A. Experimental Setup

We use the Summit [17] system at Oak Ridge National Lab. Summit comprises 4,608 compute nodes, each with 6 16GB V100 GPUs. Triplets of GPUs are associated with each CPU; GPUs 0-2 with socket 0 and GPUs 3-5 with socket 1. Within a triplet, components are fully-connected by NVLink 2.0 x2 links, for 100 GB/s bidirectional bandwidth. Between triplets, the sockets are connected with a 64 GB/s X-bus SMP interconnect. The network is a non-blocking fat tree of EDR InfiniBand with 23 GB/s node injection bandwidth. The throughput numbers are calculated by calculating input edges over inference time. Inference time includes overlapped data copy time of weight and inputs to the GPUs. Our evaluation is carried out using Spectrum MPI 10.3.1.2, XL compiler 16.1.1, nvcc 10.1.234, and CUDA driver 418.116;

B. Single GPU performance

1) Single Volta V100 GPU performance: The first column of Table I shows the single-GPU throughput of our optimized implementation. Compared to the baseline implementation, our optimizations provide $5.56\times - 11.84\times$ performance improvement. Thus, we do not report the baseline performance in detail due to lack of space. As the depth of the network increases, the throughput increases due to increased average sparsity in the features. This means more feature rows are entirely zero, and no thread-block is mapped to them. As the number of neurons increases, the throughput decreases due to two effects. First, increased zero-padding in the sliced ELLPACK format causes more wasted work and memory bandwidth. Second, less reuse from shared memory, since a given set of outputs is less likely to reuse the same footprint of features.

TABLE II: Performance Comparison with 2019 (TeraEdges/second). Here "sp" stands for Speed up.

		This Work	Bisson & Fatica [16] 2019 Champion		Davis et al. [18] 2019 Champion		Ellis & Rajamanickam [19] 2019 Innovation		Wang et al. [20] 2019 Student Innov.		Wang et al. [21] 2019 Finalist	
Neurons	**Layers**	**Throughput**	**Throughput**	**sp**	**Throughput**	**sp**	**Throughput**	**sp**	**Throughput**	**sp**	**Throughput**	**sp**
1k	**120**	3.069E+13	4.517E+12	6.79	1.533E+11	200.20	2.760E+11	111.20	2.760E+11	111.20	8.434E+10	363.90
	480	3.111E+13	7.703E+12	4.04	2.935E+11	106.00	2.800E+11	111.11	2.800E+11	111.11	9.643E+10	322.61
	1920	3.100E+13	8.878E+12	3.49	2.754E+11	112.56	2.800E+11	110.71	2.800E+11	110.71	9.600E+10	322.90
4k	**120**	8.075E+13	6.541E+12	12.34	1.388E+11	581.77	2.120E+11	380.90	2.120E+11	380.90	6.506E+10	1,241.23
	480	8.102E+13	1.231E+13	6.58	1.743E+11	464.83	2.160E+11	375.09	2.160E+11	375.09	6.679E+10	1,213.02
	1920	8.220E+13	1.483E+13	5.54	1.863E+11	441.22	2.160E+11	380.56	2.160E+11	380.56	6.617E+10	1,242.20
16k	**120**	1.497E+14	1.008E+13	14.85	1.048E+11	1,428.72	1.270E+11	1,178.98	1.270E+11	1,178.98	3.797E+10	3,942.89
	480	1.399E+14	1.500E+13	9.33	1.156E+11	1,210.55	1.280E+11	1,093.28	1.280E+11	1,093.28	3.747E+10	3,735.00
	1920	1.465E+14	1.670E+13	8.77	1.203E+11	1,217.79	1.310E+11	1,118.32	1.310E+11	1,118.32	3.750E+10	3,906.39
64k	**120**	1.729E+14	9.388E+12	18.42	1.050E+11	1,646.67	9.110E+10	1,897.91	9.110E+10	1,897.91	–	–
	480	1.700E+14	1.638E+13	10.38	1.091E+11	1,558.48	8.580E+10	1,981.70	8.580E+10	1,981.70	–	–
	1920	1.715E+14	1.787E+13	9.59	1.127E+11	1,521.56	8.430E+10	2,034.16	–	–	–	–

2) Single Ampere A100 GPU Result: We now compare the performance of the optimized fused kernel on the latest GPU, NVIDIA Ampere A100. A100 increases memory capacity from 16/32GB to 40GB, grows L2 cache from 6MB to 40MB, and brings $1.73\times$ global memory bandwidth and $1.24\times$ the single-precision floating point peak performance [22]. At the time of writing, the A100 is not available to the public, so we do not have a cluster for scaling studies. We restrict our study to the performance improvement of our optimized-fused kernel on out-of-the box single A100 GPU, without any A100-specific optimizations. We use CUDA 11 with NVIDIA driver 450.51 on an Ubuntu machine.

Table I shows the performance improvement achieved by the our optimized implementation on Ampere A100 GPU. Using the same code, A100 yields a $1.45\times$ to $2.37\times$ speed up. The improvement is more modest for smaller networks, for which the implementation makes only modest demand on the memory subsystem. For larger networks where the kernel relies more on cache and global memory performance, A100 provides a much larger speedup. Optimization specific to A100's architectural features should yield further improvement.

C. Multi-GPU Scaling on Summit GPU Cluster

In this evaluation, we discuss our multi-GPU implementation of the optimized-fused kernel. We use MPI to parallelize the sparse DNN inference across multiple GPUs on the Summit system This work adopts a "batch parallel" strategy, where weights are replicated between GPUs and the features are partitioned evenly across GPUs. Table I shows the throughput achieved on Summit for all network configurations on up to 768 GPUs. For the smallest network configuration, the strong-scaling limit is 16 nodes (96 GPUs), which is more than the 4-GPU scaling limit observed on a *single* node of the 2019 champions. Although strong scaling is observed, the largest parallel efficiency observed is 54% at 3 nodes. For larger configuration, strong scaling is observed out to 128 nodes (768 GPUs) and corresponds to 223.3 GigaEdges/s/GPU, and a parallel efficiency of 87.6% is observed for one full node (6 GPUs), and 82.6% at 2 nodes.

D. Comparison with Prior Work

Table II compares the fastest time from our submission with the fastest times from various 2019 submissions. Our speedup varies from $3.49\times$ to $18.42\times$ over the fastest champion from 2019, with one configuration achieving a $1600\times$ speedup over one of the 2019 champions. Part of the contribution is from the kernel implementation: the single-GPU speedup over Bisson & Fatica [16] varies from $4.3\times$ for the 1024-neuron 120-layer to $1.13\times$ for the 65526-neuron 1920-layer configurations. The other part is the load-balancing, which provides better scaling, allowing us to effectively utilize large numbers of GPUs.

1) Comparison with cuSparse Library: Wang et al. [21] used cuSparse on a V100 in their challenge submission in 2019. Like this work, they do not operate on empty feature rows, making it possible for a direct comparison. As they report single-GPU times on V100, we are able to compare our single-GPU implementation to the fastest results they achieved with cuSparse. The last columns of Table II contain their single-GPU performance. The speedup of our single-GPU implementation varies from $210\times$ for 4096n/1920l to 125 for 1024n/120l over their cuSparse implementation.

V. Conclusion

In this work, we show that a baseline sparse DNN kernel performance is limited by memory bandwidth and irregular memory accesses. To address this, we propose five optimizations: register tiling, shared-memory tiling, efficient memory access to weight matrices, compact index representation and streaming. Our optimized implementation provides up to 180 TeraEdges per second inference throughput. These results are up to $4.3\times$ faster on a single V100 GPU and an order of magnitude faster at scale when compared with 2019 Champions. We also show that proposed implementation when executed with the latest Ampere A100 GPU, without any optimization, achieves up to $2.37\times$ speed up over V100 GPU.

REFERENCES

[1] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, p. 47804789, Jul 2019.

[2] B. Zoph, G. Ghiasi, T.-Y. Lin, Y. Cui, H. Liu, E. D. Cubuk, and Q. V. Le, "Rethinking pre-training and self-training," 2020.

[3] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2019.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[5] "Exploring massively multilingual, massive neural machine translation https://ai.googleblog.com/2019/10/exploring-massively-multilingual.html," October 2019.

[6] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2018.

[7] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *International Conference on Learning Representations (ICLR)*, 2016.

[8] J. Kepner and R. Robinett, "Radix-net: Structured sparse matrices for deep neural networks," *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019.

[9] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, (Long Beach, California, USA), pp. 6105–6114, PMLR, 09–15 Jun 2019.

[10] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, A. Reuther, R. Robinett, and S. Samsi, "Graphchallenge.org sparse deep neural network performance," 2020.

[11] J. Kepner, A. Simon, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," *arXiv:1909.05631v1 [cs.CV]*, Sep.2019.

[12] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance extreme Computing Conference (HPEC'18)*, (Boston, USA), 2018.

[13] C. Pearson, M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on triangle counting on gpu," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, (Boston, USA), 2019.

[14] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss decomposition on gpu," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, (Boston, USA), 2019.

[15] M. Hidayetoğlu, T. Biçer, S. G. de Gonzalo, B. Ren, D. Gürsoy, R. Kettimuthu, I. T. Foster, and W.-m. W. Hwu, "Memxct: Memory-centric x-ray ct reconstruction with massive parallelization," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 19, (New York, NY, USA), Association for Computing Machinery, 2019.

[16] M. Bisson and M. Fatica, "A gpu implementation of the sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2019.

[17] "Summit user guide," 2020.

[18] T. A. Davis, M. Aznaveh, and S. Kolodziej, "Write quick, run fast: Sparse deep neural network in 20 minutes of development time via suitesparse: Graphblas," in *2019 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2019.

[19] J. A. Ellis and S. Rajamanickam, "Scalable inference for sparse deep neural networks using kokkos kernels," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2019.

[20] X. Wang, Z. Lin, C. Yang, and J. D. Owens, "Accelerating dnn inference with graphblas and the gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2019.

[21] J. Wang, Z. Huang, L. Kong, J. Xiao, P. Wang, L. Zhang, and C. Li, "Performance of training sparse deep neural networks on gpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–5, IEEE, 2019.

[22] "NVIDIA Tesla A100 Tensor Core GPU Architecture." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.