A GPU Implementation of the Sparse Deep Neural Network Graph Challenge

Mauro Bisson, Massimiliano Fatica
NVIDIA Corporation
Santa Clara, CA 95050, USA

***Abstract*—This paper presents a CUDA implementation of the latest addition to the Graph Challenge, the inference computation on a collection of large sparse deep neural networks. A single Tesla V100 can compute the inference at 3.7 TeraEdges/s. Using the managed memory API available in CUDA allows for simple and efficient distribution of these computations across a multi-GPU NVIDIA DGX-2 server.**

I. Introduction

The MIT/IEEE/Amazon Graph Challenge, a recent effort aimed at driving progress in the graph analytics field, just added a new challenge that focuses on Deep Learning and Artificial Intelligence, research fields that have gained an incredible popularity in recent years. The latest challenge performs inference computations on a collection of large sparse deep neural networks and is related to the latest trends in Deep Learning research that focus on reducing storage and execution time by sparsifying the neural networks.

The use of GPUs in high performance computing, sometimes referred to as *GPU computing*, is now very popular due to the high computational power and high memory bandwidth of these devices coupled with the availability of high level programming languages and tools. In the previous years, we have presented GPU results for Pagerank Pipeline (a pre-challenge) [1] and for the Static Graph Challenge (SGC) [2], workloads that are usually not indentified as a good fit for GPUs given their irregular memory access patterns, that have set new records [3]–[5].

The aim of this work is to show that GPUs can also sustain high performance for this latest challenge. Given that GPUs are the hardware of choice for traditional Deep Learning workloads, this is not a surprise.

In the next sections, we describe our CUDA implementation and our matrix storage strategy and then present the results obtained running inference on all the datasets on Tesla V100 cards.

II. Sparse Deep Neural Network Challenge

The Sparse DNN challenge operates on datasets that can be downloaded from the Graph Challenge homepage[1]. Each dataset is comprised of a sparse matrix Y, containing the the input data for the network, 1920 layers of neurons stored in as many other sparse matrices Ws, truth categories (to validate outputs), and the bias values used for the inference. The input matrix Y has dimensions $N_{inputs} \times N_{neurons\_per\_layer}$ while layer matrices have dimensions $N_{neurons\_per\_layer} \times N_{neurons\_per\_layer}$. The number of inputs is fixed in all datasets to $60,000$ while the number of neurons per layer can be 1024, 4096, 16384, and 65536. For each number of neurons (per layer), the challenge requires to process the input matrix with three networks composed of the first 120, 480, and all of the 1920 layers from the corresponding W matrices. In total 12 DNNs are defined by the challenge ($[1024, 4096, 16384, 65536] \times [120, 480, 1920]$).

To ease the reading, in the following we will refer to the number of neurons per layer simply as the number of neurons. Moreover we will indicate with NI the number of inputs ($60,000$), with NN the number of neurons, and with NL the number of layers.

The challenge requires to run the following steps for each combination of neurons and number of layers:

1) load a DNN ($W_0, ...W_{NL-1}$) and its corresponding input Y_0 from file
2) create and set the appropriate bias vectors b
3) evaluate the Rectified Linear Unit (ReLU) activation function for all layers:

$$\textbf{for each } l \textbf{ in } (0, ..., NL-1) : Y_{l+1} = h(Y_l W_l + b)$$

4) identify the categories in the final Y with non zero entries
5) check that the computed categories are correct by comparing them with the given truth categories
6) compute and report time and performance :
(NI $\times$ #edges) / (time of steps 3 and 4).

The number of edges used at step 6 is the total number of non-zeroes in the W matrices, i.e. $\sum_{i=0}^{NL-1} nnz(W_i)$. The function $h(x)$ is the Rectified Linear Unit (ReLU) activation function, that saturates its input in the range $[0, 32]$ (the upper limitation to 32, not typically found in classical ReLUs, is imposed to avoid overflow). The bias values are all negatives to avoid fill in and they differ depending on the number of neurons. All the non-zero entries of the Y_0 input matrix are equal to one and all the non-zero entries of the Ws are equal to $1/16$ (resulting in weights into any neuron summing to 2, this choice seems to produce layer count independent properties). The classical way of applying a bias augmenting the matrix Y by a unit (column) vector and adding a row with the bias value to the weight matrix, cannot be used in this case, since it would cause

[1]http://graphchallenge.mit.edu/data-sets

fill in of Y. The bias needs to be applied only to the non zero elements.

The large sparse DNNs are synthetically generated by a RadiX-Net sparse DNN generator [7], that can specify the number of connections per neuron and can create an equal number of paths between all inputs/outputs and intermediate layers.

Table I reports the total number of edges (connections) for each of the 12 DNNs in the challenge and the corresponding bias values used for inference.

	Neurons per Layer			
Layers	1024	4096	16384	65536
120	3,932,160	15,728,640	62,914,560	251,658,240
480	15,728,640	62,914,560	251,658,240	1,006,632,960
1920	62,914,560	251,658,240	1,006,632,960	4,026,531,840
Bias	-0.30	-0.35	-0.40	-0.45

Table I
TOTAL NUMBER OF EDGES ($32 \times neurons/layer \times layers$) AND BIAS VALUES FOR THE 12 DNNS IN THE CHALLENGE.

III. IMPLEMENTATION

We implemented the challenge in a CUDA+OpenMP code that can use the resources of multiple GPUs connected to the same machine to perform the inference processing. It iterates though the supported number of neurons $NN = \{1024, 4096, 16384, 65536\}$ and, for each one, reads the corresponding input matrix Y^{NN}. Then it iterates through the supported number of layers $NL = \{120, 480, 1920\}$ and, for each one, reads the NL layer matrices $W^{NN}_{0,\ldots,NL-1}$ and calls the CUDA implementation of the `inferenceReLUvec`$(W^{NN}_{0,\ldots,NL-1}, b_{NN}, Y^{NN})$ function on each GPU. That function performs NL consecutive updates of Y^{NN} by multiplying it with the current W^{NN} matrix, adding the bias, and then applying the ReLU.

The inference function is called by one OpenMP thread per GPU, each one on a part of the Y^{NN} matrix. During the NL iterations of the inference function, each GPU executes two kernels, one for the multiplication+ReLU of its own part of Y^{NN} and one to compute the non-empty row indices in the resulting matrix. Finally, the threads update the partitioning of Y^{NN} and the next iteration is started. More details are given in Section III-C.

The code can run in both single and double precision and the latter results in a 40% penalty in performance. Since inference is typically performed in low precision, we chose to use single precision in order to better approximate real-world cases.

A. Data structures

Since the W matrices are only read and never modified, we store them as Compressed Sparse Row (CSR). This format allows to drastically limit their memory footprint and to access their rows efficiently. Moreover, in order to divide them in chunks that can be processed independently (the reason for this is given in Section III-C), we partition each $NN \times NN$ matrix W into $N_{SLAB} = (NN/COL_{BLK})$ CSRs each of dimensions $NN \times COL_{BLK}$ (where COL_{BLK} is a divisor of NN). In other words, we partition each layer matrix into vertical slabs of the same size. Note that from the CSR representation point of view, this arrangement can easily be realized by generating the CSR of the matrix with dimensions $(N_{SLAB} * NN) \times NN$ obtained from W by shifting down each slab s by $s * NN$ rows. To do that, it is sufficient to replace the row indices of the non-zeroes read from files with their shifted values: $i \rightarrow i + NN * (j/COL_{BLK})$. Indicating with `roff[]` the array of row offsets, this representation allows to directly access the rows of slab s by accessing `roff[s*NN:(s+1)*NN-1]`. The only drawback of this partitioned representation is that the size of the row offset array increases by a factor eqaul to N_{SLAB}. However, since in our runs the number of slabs can grow up to 16 and given the possible values of the number of neurons (≤ 65536), the increase in memory requirement is completely negligible.

The Y matrix, on the other hand, can change after every iteration and thus storing it as a CSR would impose a high maintenance cost. We considered using the more update-friendly format ELLPACK which requires storage space equal to the number of rows (NI) times the maximum number of non-zeroes on a row. However, as we'll discuss later in Section IV, since after a short number of inference iterations the Y contains (exclusively) full rows, this format would require space for exactly $NI \times NN$ elements, thus turning the 2D array of columns into a conspicuous waste of memory. For this reason, and since the full Y for each DNN easily fits in the memory of current generation GPUs, we stored it as a dense $NI \times NN$ matrix. This allows to completely avoid any overhead due to the management of a sparse representation. We complement Y with two additional arrays of size NI. One, containing the list of its non-empty rows, is used to limit the dot products in the matrix-matrix multiplications to only the rows that can produce non-zeroes. The second, which contains the number of non-zeroes per row, is used to efficiently generate the first array after each update. More details about these arrays are given in Section III-C.

The computational workload is distributed among the GPUs following a data parallel approach by partitioning the Y matrix into horizontal slabs containing the same number of non-empty rows. In this way each GPU can execute the multiplication+ReLU on its own part of the matrix independently of the others. In addition, since the number of non-empty rows vary during the inference computation, the partitioning of Y must be updated periodically during the iterations in order to keep the workload balanced among the GPUs.

Even with our choice of storage formats for the Y and the W matrices, the memory required for the 1920 matrices W is twice the amount required for the Y, for any number of neurons. This make impractical to pre-load all of them to the global memory of the GPUs before the inference kernel calls. For this reason, we read all the layer matrices into pinned host memory and allocate device memory on each GPU only for

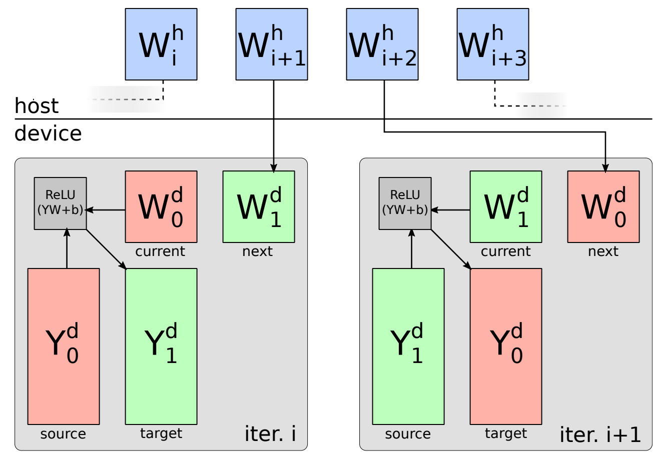
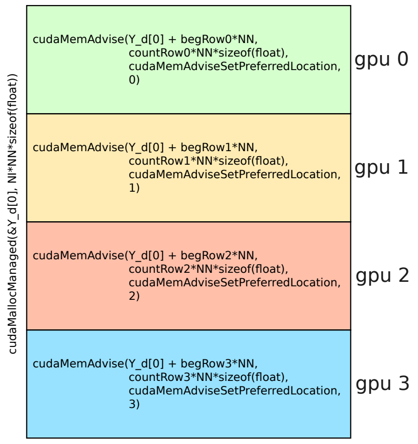


Figure 1. Double buffering scheme used for inference computation. During each iteration, the matrix in the source Y buffer is multiplied with the matrix in the current W buffer and the saturated result is stored in the target Y buffer. Concurrently, the next W matrix is copied from host memory to the next W buffer in device memory. Then the buffers are swapped and the next iteration is executed.


```
cudaMallocManaged(&Y_d[0], NI*NN*sizeof(float))
```

```
cudaMemAdvise(Y_d[0] + begRow0*NN,
              countRow0*NN*sizeof(float),
              cudaMemAdviseSetPreferredLocation,
              0)
```

```
cudaMemAdvise(Y_d[0] + begRow1*NN,
              countRow1*NN*sizeof(float),
              cudaMemAdviseSetPreferredLocation,
              1)
```

```
cudaMemAdvise(Y_d[0] + begRow2*NN,
              countRow2*NN*sizeof(float),
              cudaMemAdviseSetPreferredLocation,
              2)
```

```
cudaMemAdvise(Y_d[0] + begRow3*NN,
              countRow3*NN*sizeof(float),
              cudaMemAdviseSetPreferredLocation,
              3)
```

Figure 2. Example of Unified Memory calls required to partition a buffer among 4 GPUs, enabling direct memory access among devices via NVLink.

two of them. We also use two Y buffers in order to hold the input and output of each matrix-matrix multiplication.

During inference iterations, the compute kernel running on each GPU reads the current W from one buffer while the CSR required for the next iteration is copied from host memory to the other buffer. Likewise, it reads the rows in its partition of Y from the source buffer and writes the output in the same partition of the destination buffer. In order to perform the copies with a single `cudaMemcpy()` call per GPU, the arrays of each CSR are packed one after the other inside a single memory allocation.

The time required to perform the host-to-device (H2D) copies is negligible with respect to the time required by the compute kernel thus the copies are completely hidden by the computations.

Figure 1 shows the double buffering strategy used for the Y and W matrices.

B. Multi-GPU setup

On the system side, there are now servers like the NVIDIA DGX-2 with all GPUs connected via NVLink and NVSwitch. These types of systems are bringing SMP-like capabilities to multi-GPU programming, as GPUs on the node can access data on other GPUs in a fast and transparent way.

In this work we exploited the capabilities of the CUDA Unified Memory system that allows for memory allocations that span multiple GPUs connected through NVLink. When one GPU tries to access data stored on another GPU, the driver automatically migrates the corresponding memory pages.

We took advantage of this system to perform the initial partitioning of the Y matrix (both buffers), before the inference function is called. Inside the function, rows are automatically migrated among GPUs according to the changes on the distribution of the non-empty rows.

To perform the initial partitioning it is sufficient to allocate the entire $NI \times NN$ matrix via a single call to the `cudaMallocManaged()` function and then, for each slab, setting the preferred location equal to the id of target GPU. This is done by using the `cudaMemAdvise()` call specifying the `cudaMemAdviseSetPreferredLocation` advice and the memory range for each slab. Figure 2 shows an example of the Unified Memory calls and the buffer regions they need to be called onto.

We execute those calls also for the two arrays containing the non-empty row indices and the row lengths. In total, since we use two buffers for the Y matrix, six `cudaMemAdvise()` calls are required per GPU.

This setup does not require any explicit exchange of data among GPUs (like it would normally be necessary in a CUDA+MPI setup). Rather, the kernels are simply launched on each GPU with the same buffer pointers. When a GPU executes a load on a word belonging to another GPU then a data transfer through NVLink automatically takes place.

C. Inference kernel implementation

In this section we'll describe the our implementation of the `inferenceReLUvec()` function. It is called, once per DNN, by one OpenMP thread per GPU. Upon entry, each slab of the input matrix Y^{NN} and the whole first layer matrix W_0^{NN} are ready in the device memory of each GPU. The inference is computed iteratively by performing NL updates of the matrix Y^{NN}. Each thread updates its part by launching a CUDA kernel that reads it from the source Y buffer and multiplies it by the current W matrix, computes the ReLU, and stores the result into the same section of the destination Y buffer. In order to overlap this computation with the H2D copy of the next layer matrix, the kernel and the memory copy are launched on two distinct streams. After the kernel is completed, the OpenMP threads compute the new global list of non-empty rows and the partitioning is updated by dividing them evenly among the GPUs. Finally, each thread

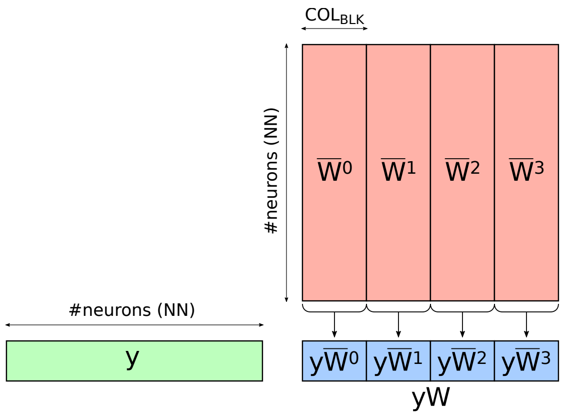


Figure 3. Subdivision of the product between a row of the input matrix Y and a layer matrix W into four independent products of the row with four vertical slabs of W.

synchronizes on its H2D copy, swaps the Y buffers, and a new iteration in started.

Most of the complexity of this procedure lies in the update kernel. It is launched with a number of thread blocks (or Cooperative Thread Arrays, CTAs) equal to the number of non-empty rows in the current Y slab and each block computes the product of one row with the current W matrix. Since we store these matrices as CSRs, it is possible to perform this operation efficiently by reading only the rows of W indexed by the column indices of the non-zeroes in the row of Y (instead of reading each column, as it would be necessary with formats storing the values by column). The general idea is that the multiplication of a row vector $y_{1\times N}$ by a matrix $W_{N\times N}$ can be expressed as the sum of the rows of W each scaled by one element of y:

$$y \cdot W = \sum_{j=0}^{N} y_j * W_{j,\cdot} \qquad (1)$$

This approach can be implemented directly and efficiently in a CUDA kernel by having the block that process row y to accumulate in a temporary array the rows of W corresponding to the non-zeroes of y, each one scaled by the right value. In order to speed up the accumulations, we keep the temporary array in shared memory. However, given the limited amount of that memory space, using a float array with size equal to the number of neurons for each block could either severely limit the occupancy of the kernel or not be at all possible (for example the case $NN = 65536$ would require 256KB of shared memory per block, more than the total available amount). For this reason, as anticipated in Section III-A, we partition each W matrix in $N_{SLAB} = NN/COL_{BLK}$ vertical slabs $\overline{W}$ of dimension $NN \times COL_{BLK}$. Denoting with $\bigoplus$ the concatenation of two vectors, Eq. 1 can be rewritten using the $\overline{W}$ matrices as:

$$y \cdot W = \bigoplus_{i=0}^{N_{SLAB}} (\sum_{j=0}^{N} y_j * \overline{W}^{i}_{j,\cdot}) \qquad (2)$$

In this way the multiplication between y and W is broken into N_{SLAB} independent multiplications between Y and the $\overline{W}$s, each one requiring a temporary array of size COL_{BLK}. The COL_{BLK} value can be tuned to make the shared memory amount required per block small enough not to limit the occupancy. Figure 3 shows a graphical representation of the computation expressed by Eq. 2.

Algorithm 1 CUDA kernel for ReLU(Y*W+b).

Input: Y0[NI][NN], matrix Y
Input: nerowsY, number of non-empty rows of Y
Input: rowsY0[NI], non-empty row indices of Y
Input: rlenY0[NI], row lengths of Y
Input: COL_BLK, integer divisor of NN
Input: roffW[NN*Nslab-1], array of row offsets of partitioned W matrix
Input: valsW[nnz], array of values
Input: colsW[nnz], array of column indices
Input: bias
Output: Y1[NI][NN], reult matrix ReLU(Y*W+b)
Output: rlenY1[NI], row lengths of result matrix

```
if (blockIdx.x >= nerowsY) then
    return
end if
__shared__ float shRow[COL_BLK]
tid = threadIdx.y*blockDim.x + threadIdx.x
rid = rowsY0[blockIdx.x]
__syncthreads()
if (tid == 0) then
    rlenY0[rid] = 0;
    rlenY1[rid] = 0;
end if
for (i = 0; i < NN/COL_BLK; i++) do
    __syncthreads()
    for (j = threadIdx.x; j < COL_BLK; j++) do
        shRow[j] = 0;
    end for
    __syncthreads()
    for (j = threadIdx.y; j < NN; j += blockDim.y) do
        valY = Y0[rid][j]
        if (valY == 0) continue
        begOffW = roffW[i*NN + j] + threadIdx.x
        endOffW = roffW[i*NN + j+1]
        for (k = begOffW; k < endOffW; k += blockDim.x) do
            colW = colsW[k]
            valW = valsW[k]
            atomicAdd(&shRow[colW-i*COL_BLK], valY*valW)
        end for
    end for
    __syncthreads()
    count = 0
    for (j = 0; j < COL_BLK; j += blockDim.x*blockDim.y) do
        v = (j+tid < COL_BLK) ? shRow[j+tid]+bias : -1
        count += __syncthreads_count(v > 0)
        Y1[rid][i*COL_BLK + j+tid] = min(32, max(0, v))
    end for
    if (tid == 0) then
        rlenY1[rid] += count
    end if
end for
```

Algorithm 1 shows a simplified pseudocode for our update kernel that takes advantage of the above multiplication strategy. The kernel is launched with a 2D block per non-empty row of Y. The `(dimX,dimY)` dimensions of a block depend on the number of neurons of the DNN being processed and we call each of the `dimY` sequences of consecutive `dimX` threads, a sub-CTA of the block. Each block uses an amount of shared memory equal to the size of a float array with COL_{BLK} elements (line 4). Initially, each thread reads the index of the non-empty row to be multiplied by the layer matrix (line 6) and, after the value has been read by the whole block, one thread resets the length of that row in the row-length arrays of both buffers (lines 8-11). At this point the block cycles through the $\overline{W}$ matrices (line 12). Since each product is computed by consecutive accumulations into the shared array, this is set to zero at the beginning of the loop (lines 14-16), and then the threads loop along row `rid` of $Y0$ using one sub-CTA per element (line 18). If the current value is equal to zero, then the sub-CTA skips to the next one (line 19-20), otherwise the row $\overline{W}_j^i$ is processed. Its offset is read from the row offset array (inside the i-th chunk of size NN, lines 21-22), and then the sub-CTA loops through the row (line 23). Each non-zero is multiplied by the non-zero read from $Y0$ and the result is added to the location of the shared array holding the partial result for that column index (lines 24-26). Since the $\overline{W}$ CSRs contain column indices in consecutive ranges of size COL_{BLK}, the shared write is executed at a location obtained by offsetting the column index by a value equal to the index of the first column of slab $\overline{W}^i$. Moreover, since multiple sub-CTAs may update the same element, the accumulation is performed via an atomic operation. After the whole line of $Y0$ is processed, the content of the shared array is copied into the destination buffer $Y1$. This is done by having the whole block loop through the shared array (line 31). Each thread reads an element, adds the bias, participates in a synchronization that counts the number of positive values (to keep track of the number of non-zeroes), and writes to the destination buffer the value saturated between 0 and 32 (lines 32-34). Finally, the number of non-zeroes written to the destination buffer is accumulated into the destination row-length array (lines 36-38) and the processing of the next $\overline{W}$ matrix takes place in the next iteration of the outermost loop.

Note that the arrays `rlenY0` and `rlenY1` are not used in the kernel. Rather, they are used after its execution to compute the list of non-empty rows in the destination $Y1$ buffer. This is done by compacting the indices of the non-zero entries in `rlenY1` inside the `rowsY1` array (used by the kernel in the next iteration, after the buffers swap). Setting them to zero in the kernel allows to avoid an explicit reset call at each iteration of the calling function.

D. Optimizations

The main target for optimization in Algorithm 1 is the multiple row reads of the Y matrix performed by each block, more precisely once per $\overline{W}$ matrix. This redundant memory traffic can impact performance if the rows cannot be cached effectively, especially when running with large number of neurons. For this reason in our implementation each block reads its row once, stores it in registers, and then broadcasts its content as needed across its sub-CTAs. Before the loop at line 12, each thread reads up to NN/TH_PER_BLK elements in a local array that is indexed statically by having all indices known at compile time. The loop at line 18 is then replaced by two nested loops, one iterating over the NN/TH_PER_BLK elements of the local array, that determines the value to be processed, and the innermost one that iterates `blockDim.x` times, where each sub-lane broadcasts its local Y value to its sub-CTA.

IV. Results

The organizers of the challenge provided performance measurements [8] of the reference Matlab code as a baseline to which implementors can compare to. These results, obtained on an Intel Knights Landing (KNL) 1.3 GHz, are shown in Table II.

		Layers		
CPU	Neurons/layer	120	480	1920
KNL 1.3 GHz	1024	0.376 (626s)	0.386 (2440s)	0.386 (9760s)
	4096	0.385 (2446s)	0.369 (10229s)	0.375 (40245s)
	16384	0.344 (10956s)	0.333 (45268s)	0.336 (179401s)
	65536	0.329 (45813s)	0.299 (202393s)	n.a. n.a.
Core i7 3.5 GHz	1024	1.89 (125s)	2.07 (456s)	2.12 (1782s)

Table II
GigaEdges processed per second and runtime measures of the reference Matlab code run on an Intel KNL 1.3 GHz (top, provided by the organizers) and on an Intel Core i7 3.5GHz (bottom).

We tested the reference code on a 3.50 GHz Intel Core i7-5930K CPU using Matlab version 2019a for the smaller case (1024 neuron per layers), with the three possible numbers of layers, 120, 480, and 1920, to assess the effect of a stronger CPU (the single core performance of KNL is quite limited). The inference on the Core i7 is five times faster, in the order of 1.8-2.0 GigaEdges/sec. The threshold operation of the maximum value has a significant impact on performance (more than 50% of the inference time is spent in this operation). In [8], processing rates for a parallel implementation using pMatlab are also plotted, with a maximum rate of 200 GigaEdges/s on the 1024/1920 DNN with 600 processors.

We run our code with each DNN on up to 16 V100 GPUs on an NVIDIA DGX-2 server. The results of our CUDA implementation are shown in Table III and plotted in Figure 4. The scaling results improve progressively with the size of the networks. The 1024/1920 case scales up to 4 GPUs, the 4096/1920 up to 8, the 16384/1920 up to 8 GPUs (and improves significantly with 16 GPUs), and the 65536/1920 scales up to 16 GPUs. This trend suggests that the current limiting factor to the scaling is the size of the networks. The output categories produced by every run matched exactly the reference categories provided by the organizers.

It is interesting to analyze the temporal evolution of the sparsity pattern. Figure 5 shows how the number of non-

		Number of GPUs				
Neurons	**Layers**	**1**	**2**	**4**	**8**	**16**
1204	120	2746.93 (0.086s)	3771.77 (0.063s)	**4517.35** (0.052s)	2389.74 (0.099s)	828.15 (0.285s)
	480	3085.20 (0.306s)	5385.83 (0.175s)	**7702.95** (0.123s)	5294.86 (0.178s)	2435.66 (0.387s)
	1920	3301.35 (1.143s)	5707.02 (0.661s)	**8877.71** (0.425s)	7892.19 (0.478s)	3887.10 (0.971s)
4096	120	2944.26 (0.321s)	4277.42 (0.221s)	6189.86 (0.152s)	**6541.21** (0.144s)	2422.33 (0.390s)
	480	3534.85 (1.068s)	5931.28 (0.636s)	8935.55 (0.422s)	**12310.41** (0.307s)	6919.26 (0.546s)
	1920	3711.09 (4.069s)	6173.09 (2.446s)	9428.95 (1.601s)	**14832.65** (1.018s)	11322.97 (1.334s)
16384	120	2227.10 (1.695s)	3905.96 (0.966s)	7139.07 (0.529s)	**10082.07** (0.374s)	6853.05 (0.551s)
	480	2821.50 (5.352s)	5537.99 (2.727s)	10716.12 (1.409s)	**15004.86** (1.006s)	13905.17 (1.086s)
	1920	3018.02 (20.012s)	5865.87 (10.297s)	11467.51 (5.267s)	16191.88 (3.730s)	**16696.51** (3.617s)
65536	120	2136.99 (7.066s)	3223.09 (4.685s)	5804.98 (2.601s)	8583.30 (1.759s)	**9388.46** (1.608s)
	480	3084.80 (19.579s)	5315.27 (11.363s)	8739.03 (6.911s)	14206.85 (4.251s)	**16378.68** (3.688s)
	1920	3470.47 (69.614s)	5874.49 (41.126s)	9534.25 (25.339s)	15399.49 (15.688s)	**17872.98** (13.517s)

Table III
GIGAEDGES PROCESSED PER SECOND AND RUNTIME FOR THE 12 DNNS MEASURED USING UP TO 16 V100 GPUS ON AN NVIDIA DGX-2 SERVER. THE ENTRIES IN BOLD ARE THE FASTEST RESULTS IN EACH CATEGORY.

zeroes and the percentage of full rows vary within the first 28 iterations of the `inferenceReLUvec()` function, for the cases 1024, 4096, 16384, and 65536, all with 1920 layers. The number of non-zero elements increase initially, after the first step, and then it decreases, while the rows start to fill. After 28/17/19/20 iterations for, respectively, the 1024/4096/16384/65536 neurons per layer case, the fill pattern stabilizes and does not change anymore.

V. CONCLUSIONS

We presented a CUDA implementation of the Sparse Deep Neural Network Graph Challenge, capable of using multiple GPUs connected to the same host machine. We measured the edge processing rate running on a DGX-2 server with all the DNNs provided by the organizers. A single V100 delivers up to 3.7 TeraEdges/s on the 4096/1920 DNN while the whole server reaches almost 18 TeraEdges/s on the 65536/1920 DNN. The processing rates of our solution are several orders of magnitude faster than the reference implementation [8]. Even a single V100 GPU outperforms by an order of magnitude the best reference result on 600 processors.

These results confirm the suitability of GPUs to accelerate Deep Learning workloads.

REFERENCES

[1] P. Dreher, C. Byun, C. Hill, V. Gadepally, B. Juszmaul and J. Kepner, "PageRank Pipeline Benchmark: Proposal for a Holistic System Benchmark for Big-Data Platforms", *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016

[2] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, J. Kepner, "Static Graph Challenge: Subgraph Isomorphism", *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017

[3] M. Bisson, E. Phillips and M. Fatica, "A CUDA implementation of the Pagerank Pipeline benchmark", *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016

[4] M. Bisson and M. Fatica, "Static graph challenge on GPU", *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017

[5] M. Bisson and M. Fatica, "Update on static graph challenge on GPU", *IEEE High Performance Extreme Computing Conference (HPEC)*, 2018

[6] M. Bisson and M. Fatica, "High Performance Exact Triangle Counting on GPUs", *IEEE Transactions on Parallel and Distributed Systems*, 2017, doi: 10.1109/TPDS.2017.2735405.

[7] R. Robinett, J. Kepner, RadiX-Net: Structured Sparse Matrices for Deep Neural Networks, *IEEE IPDPS GrAPL Workshop 2*,2019

[8] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett and S. Samsi, "Sparse Deep Neural Network Graph Challenge", https://graphchallenge.mit.edu/challenges

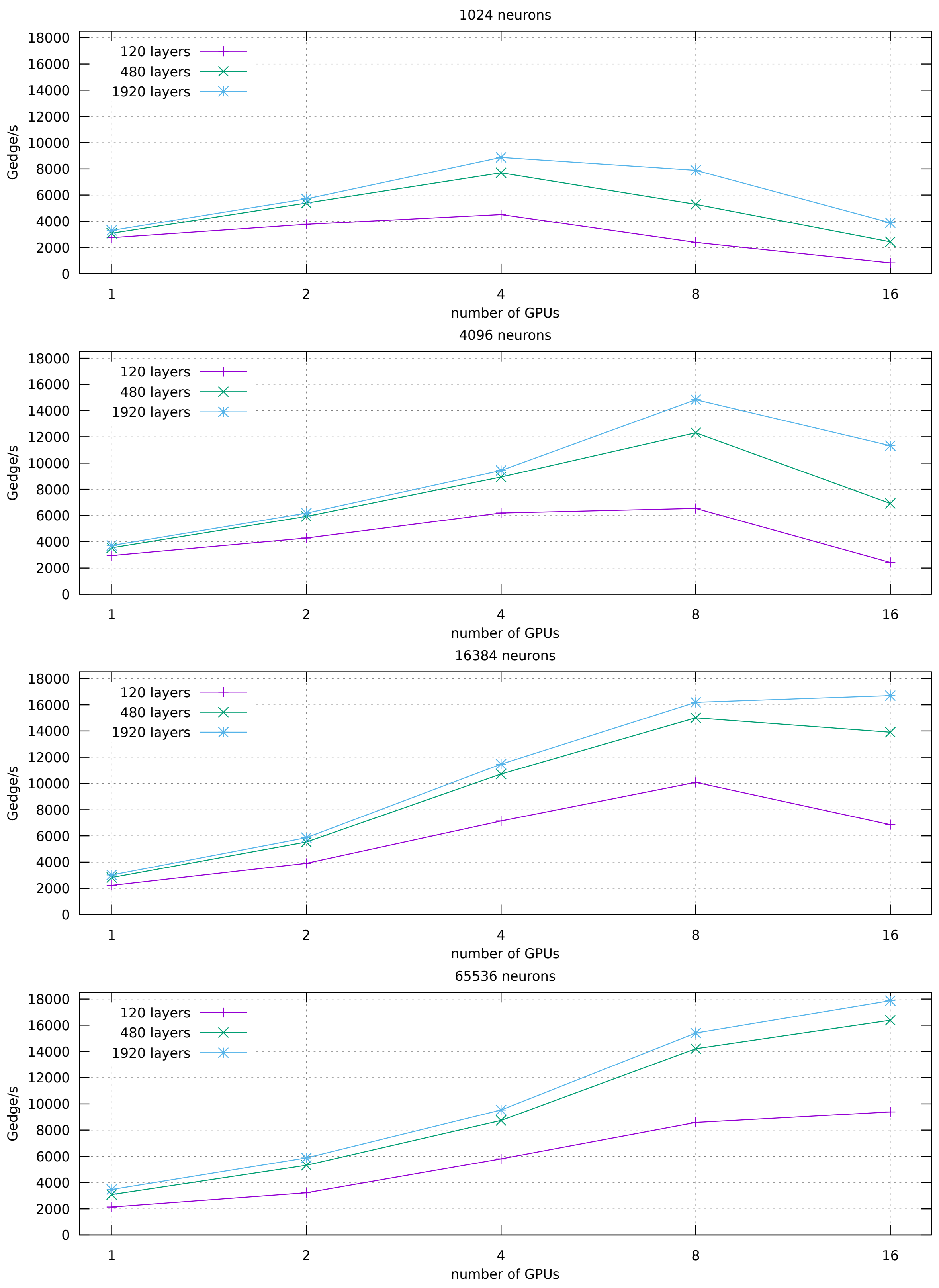


Figure 4. Strong scaling results using up to 16 V100 GPUs on an NVIDIA DGX-2 server.

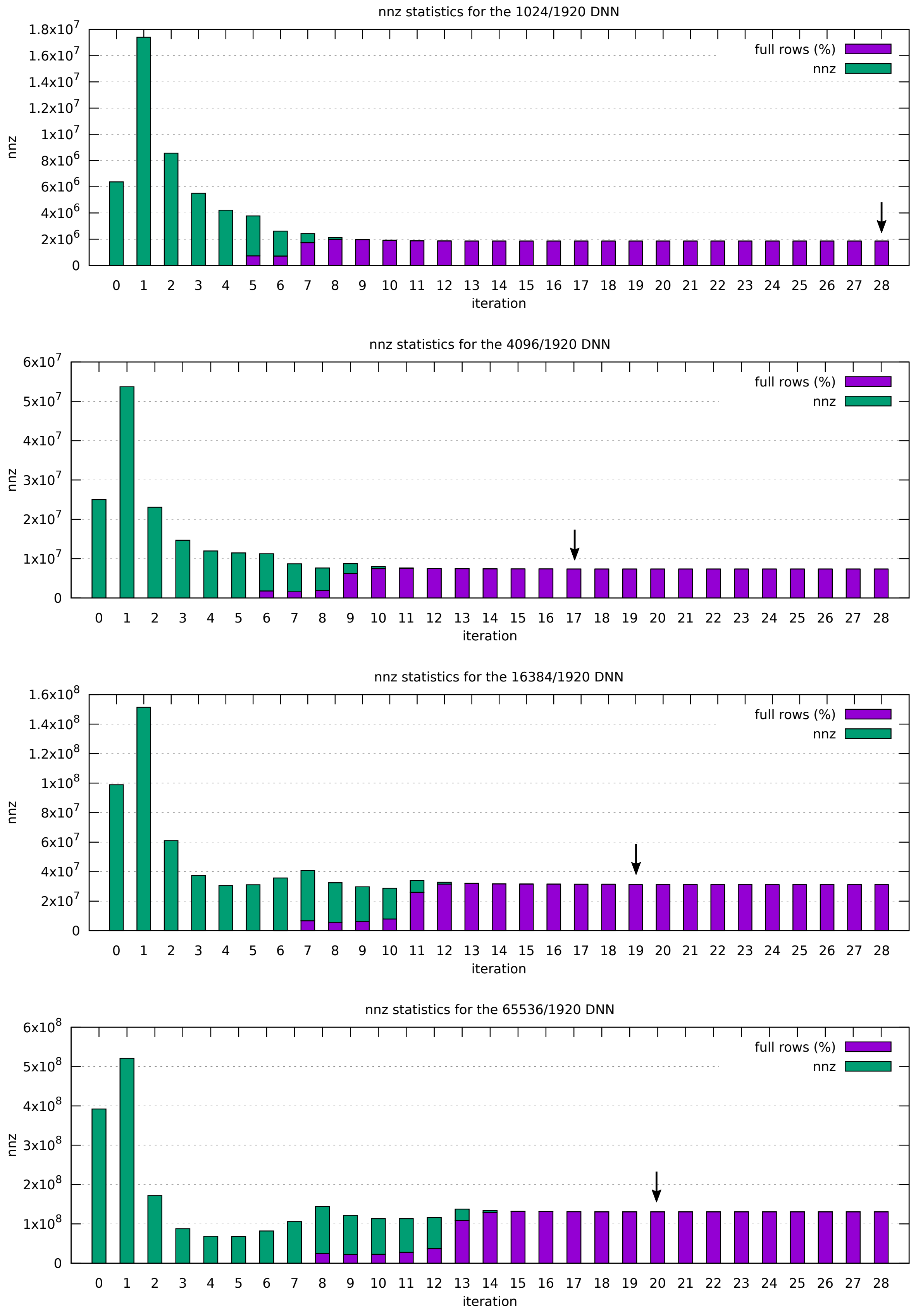


Figure 5. Number of non-zero entries of the matrix Y and percentage of the full rows over the non-empty rows, plotted over the initial iterations of the inference computation. Iteration zero corresponds to the initial values of the input matrices. The arrows show the iteration after which the pattern does not change anymore.