A GPU Implementation of the Sparse Deep Neural Network Graph Challenge

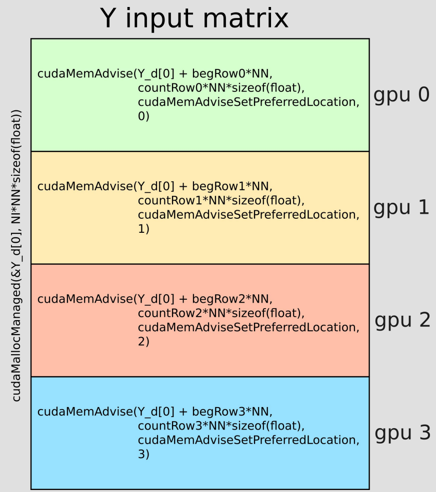
Mauro Bisson and Massimiliano Fatica
NVIDIA Corporation

Code overview

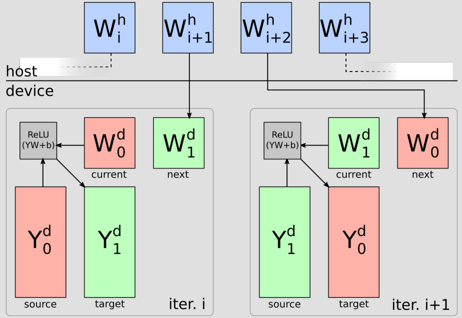
- **CUDA+OpenMP** to distribute computations on multi-GPU servers (NVIDIA DGX-2)
  - one OMP thread per GPU
  - one GPU per slab of input matrix Y
    - `inferenceReLUVec(`$W_{0...NL-1}$`, b, `$Y_{slab\_rowsxNN}$`)`
      - NL **sparse** matrix-matrix **products**
- During inference each GPU executes two **kernels** iteratively
  - one for $Y_{L+1} = \mathbf{ReLU}(Y_L W_L + b)$
  - one to compute **non-empty row indices** of Y_{L+1}
    - to limit access to meaningful rows in the next iteration
- Can run in both **single** and **double** precision

Multi-GPU setup and buffering scheme

- Input matrix `Y` partitioned into **horizontal** slabs
  - each slab can be multiplied by the same `W` **independently**
- **Partitioning** implemented using **Unified Memory**
  - single allocation of shared buffer via **cudaMallocManaged()**
  - initial calls to **cudaMemAdvice()**
  - **no explicit** exchange of data among GPUs
  - rows migrated automatically via **NVLink** during inference (based on the changes in the distribution of non-empty rows)
- Requires GPUs connected via **NVLink** (DGX-2)



- **Double buffering** scheme for matrices **`Y`** and **`W`**s
  - all **`W`**s allocated in **pinned host memory** (up to 1920)
    - memory for only **two** of them is **allocated** on each GPU
    - H2D copy of W_{L+1} **overlapped** with $Y_{L+1} = \mathrm{ReLU}(W_L Y_L + b)$
  - two device buffers for **`Y`** on each GPU
    - **Input** Y_L and **output** Y_{L+1}



Matrix data structures

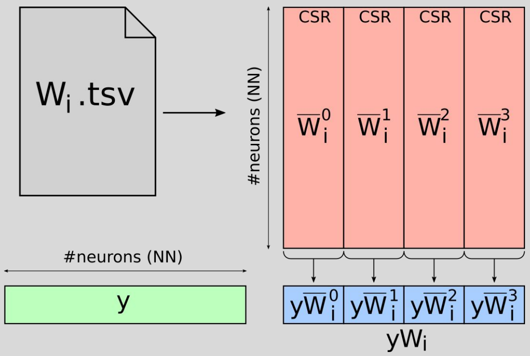
Sparse layer <u>matrices **W**</u> are read only:

- no need for update => stored as **CSR**
  - O(**nnz**(**W**)) memory required
  - **efficient** access to rows
- each W **split** into vertical slabs and stored as multiple CSRs

Sparse input <u>matrix **Y**</u> stored as... :

- ...CSR? **Pattern** can **change** at each inference step
  - high maintenance cost
- ...ELLPACK? Requires storage space **NIx(max nnz/row)**x**2** (col. indices + values)
  - low maintenance cost
  - rows can (and do!) become full during inference thus memory requirement would grow to exactly **NIxNN**x**2**
    - 50% memory waste (col index buffer unneeded)

...**dense** NIxNN matrix (up to 16GB of mem for largest case)



Row **y** can be multiplied by each slab independently, using less temporary storage than that required for the whole **W**

- size of slabs can be tweaked to control kernel occupancy

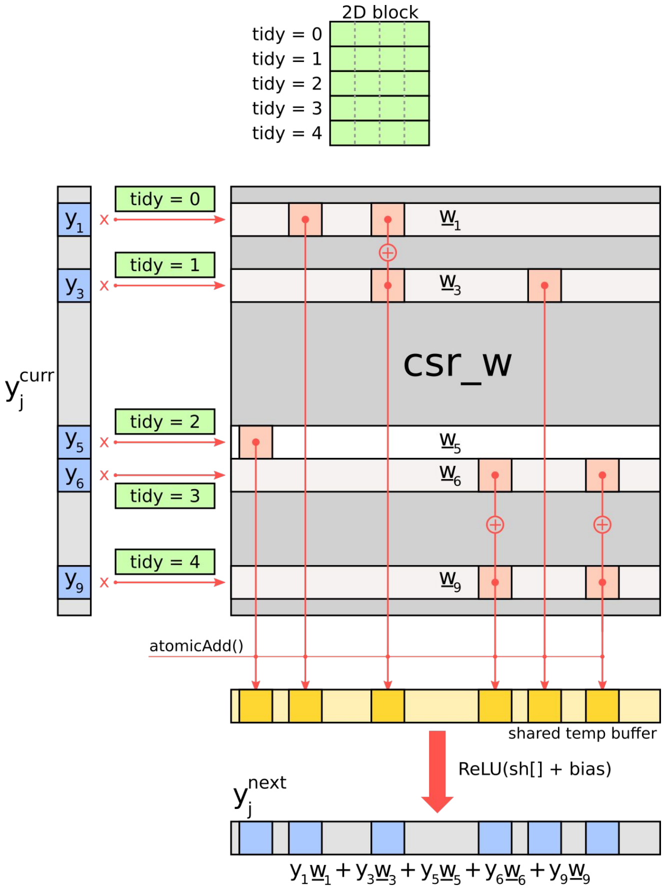
Sparse yW product implementation

- Since matrices **Y** and **W**s are **sparse**, computing **yW** as scalar products between row **y** and each column of **W** results in a large number of **unnecessary** accesses to **W**
  - the whole matrix **W** would be read for each **y**

- **Memory traffic** can be reduced drastically by performing the product as:

$$\underline{y} \cdot W = \sum_{j=0}^{N} y_j \cdot W_{j,\star}$$

  - for each **y**, **only** the non-zeroes in **W** that are necessary to the product are read



Inference results on DGX-2 (V100)

- Obtained on up to **16 V100 GPUs** of an NVIDIA DGX-2 server, single prec
- **GigaEdges** processed per second and **runtime** of inference for all the 12 DNNs in the Challenge
- Entries in bold are the **fastest** results in each category.
- A single Tesla V100 can perform inference at **3.7 TeraEdges/sec**
- 16 Tesla V100 reach **~18 TeraEdges/sec**

		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)

Thanks!