Sparsity in ML: Understanding and Optimizing Sparsity in Neural Networks Running on Heterogeneous Systems

Organizers

Wen-mei Hwu, Jinjun Xiong, Mert Hidayetoğlu, Rakesh Nagi,

Jeff Pool, Sitao Huang, Vikram Mailthody

Tutorial | MLSys 2022 | Aug 30, 2022

https://mlsys.org/virtual/2022/tutorial/2199

Slack for discussions: https://bit.ly/3KLQEUN

Outline

1. Opening remarks and overview of sparsity in ML – 1 pm
2. Tiled SpMM and its performance model on GPUs – 1:50 pm
3. Sparse deep neural network inference on FPGAs – 2:45 pm
4. Break (30 mins)
5. 2:4 Sparsity on GPU Tensor Cores – 4 pm
6. Future work and closing remarks – 4:45 pm

Large models become increasingly popular and powerful

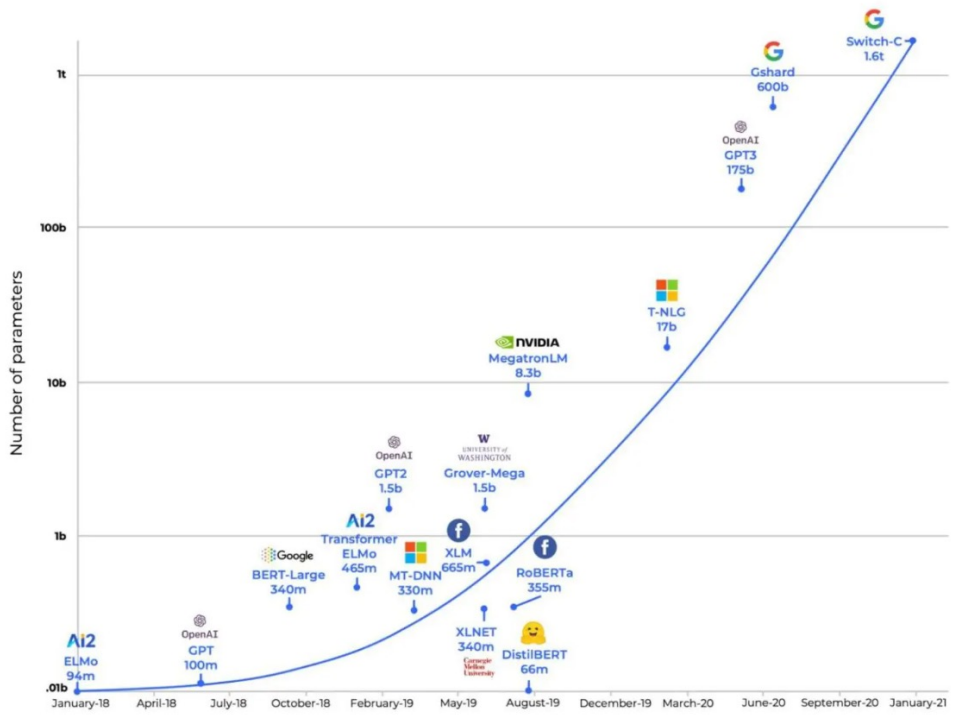
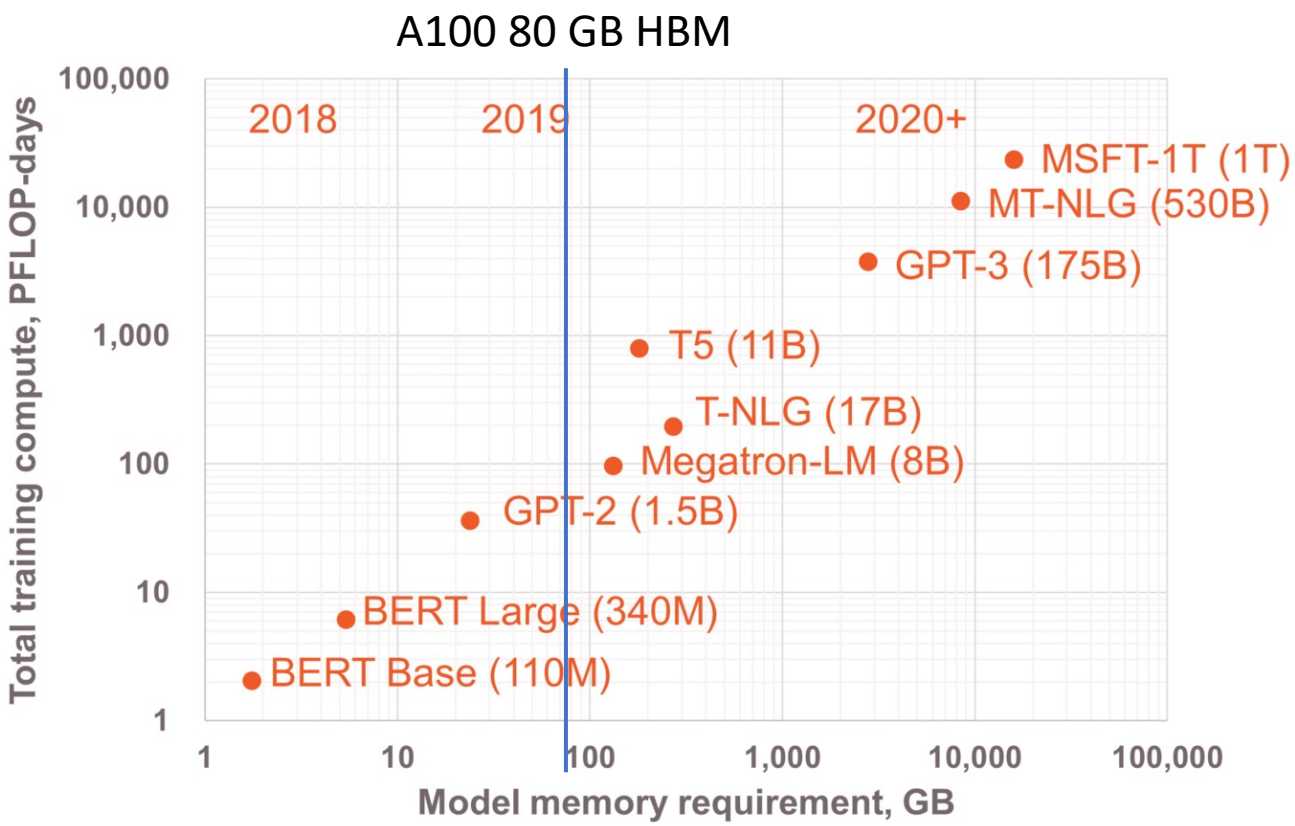


Fig: *Exponential growth of number of parameters in deep learning models*

Source: https://deci.ai/blog/sota-dnns-overview/



Source: Cerebras Systems, HotChips 2022

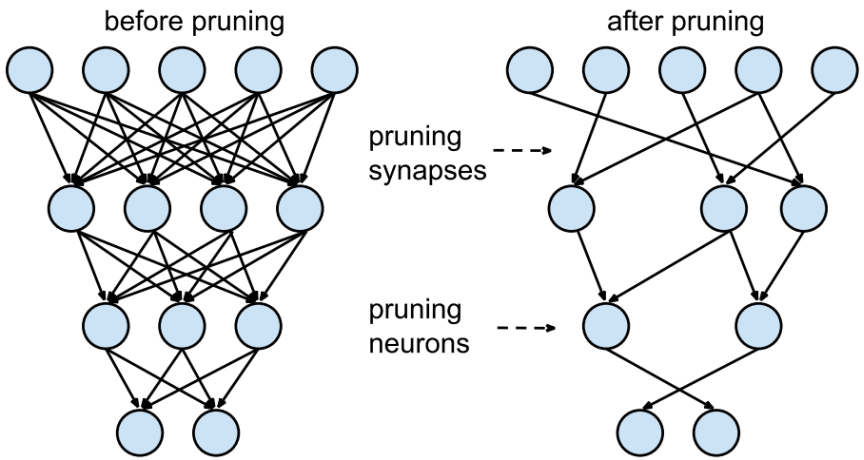
1000X in 2 years, trillion scale models just a year away

Compression to reduce model complexity and memory capacity

1. Down-sizing models: distillation, neural architectural search (NAS)
2. Operator factorization: tensor-decomposition $W[n \times m] = AB[n \times r][r \times m]$
3. Value quantization: low precision (TF32, FP16, FP8, INT8, Binary…)
4. Value compression: compression (gzip, Huffman, delta-encoding..)
5. Parameter sharing: reuse parameter across neuron (Shapeshifter)
6. **Sparsification: input sparsification, prune nodes and weights (layer-wise, channel, random, regularizer,…)**

Source: https://arxiv.org/pdf/2102.00554.pdf, https://www.youtube.com/watch?v=pAOzBbMxAmc&t=522s

Sparsity in ML

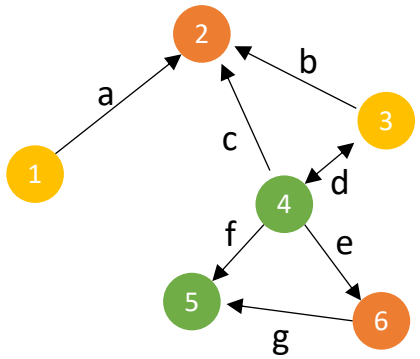


[Lecun et al. NIPS'89] [Han et al. NIPS'15]

$$\text{Activation}(\boldsymbol{A}\times\boldsymbol{B}) = \boldsymbol{C}$$

Sparse Layer Input Features Output Features

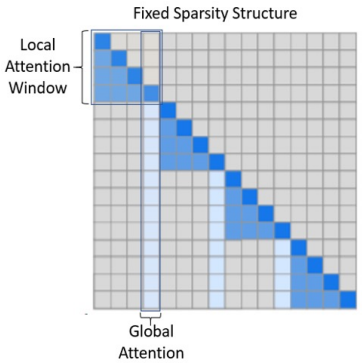
Graph Representation

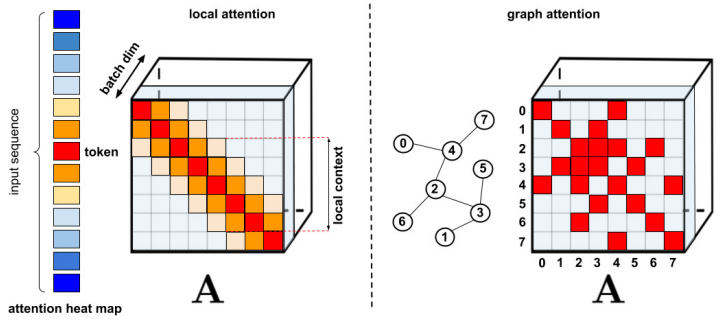


Matrix Representation

	1	2	3	4	5	6
1		a				
2						
3		b		d		
4		c	d		f	
5						e
6					g	

Sparse Attention Layers

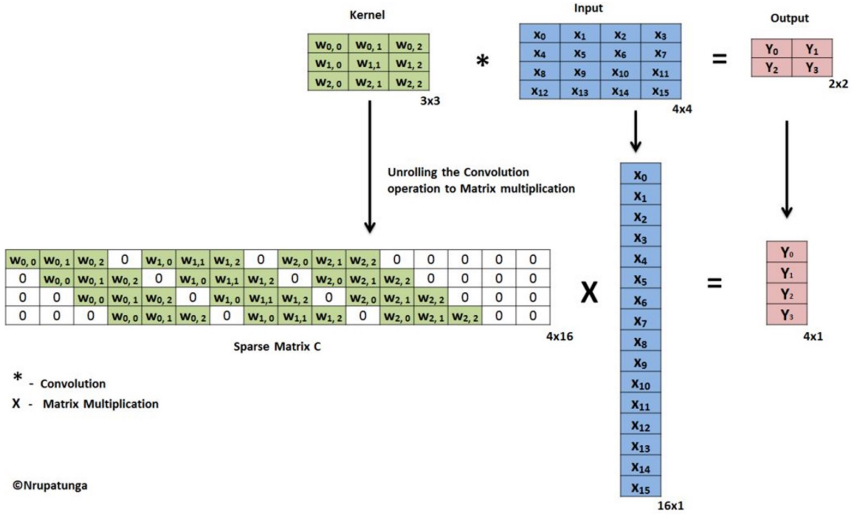




Google ai, "Rethinking Attention with Performers,"

Child, et al., "Generating Long Sequences with Sparse Transformers," Arxiv

Convolutional Neural Networks



Why would sparsity work? Because not all features matter

- Models with massive parameters do memorize input data
- Not all parameters matter (5% efficient connection can give similar accuracy as 100% model)
- Faster training with dense models, but sparse models have less overfitting problems

Extracting Training Data from Large Language Models

Nicholas Carlini[1] Florian Tramèr[2] Eric Wallace[3] Matthew Jagielski[4]
Ariel Herbert-Voss[5,6] Katherine Lee[1] Adam Roberts[1] Tom Brown[5]
Dawn Song[3] Úlfar Erlingsson[7] Alina Oprea[4] Colin Raffel[1]
[1]*Google* [2]*Stanford* [3]*UC Berkeley* [4]*Northeastern University* [5]*OpenAI* [6]*Harvard* [7]*Apple*

Abstract

It has become common to publish large (billion parameter) language models that have been trained on private datasets. This paper demonstrates that in such settings, an adversary can perform a *training data extraction attack* to recover individual training examples by querying the language model.

We demonstrate our attack on GPT-2, a language model trained on scrapes of the public Internet, and are able to extract hundreds of verbatim text sequences from the model's training data. These extracted examples include (public) personally identifiable information (names, phone numbers, and email addresses), IRC conversations, code, and 128-bit UUIDs. Our attack is possible even though each of the above sequences are included in just *one* document in the training data.

We comprehensively evaluate our extraction attack to understand the factors that contribute to its success. Worryingly, we find that larger models are more vulnerable than smaller models. We conclude by drawing lessons and discussing possible safeguards for training large language models.

1 Introduction

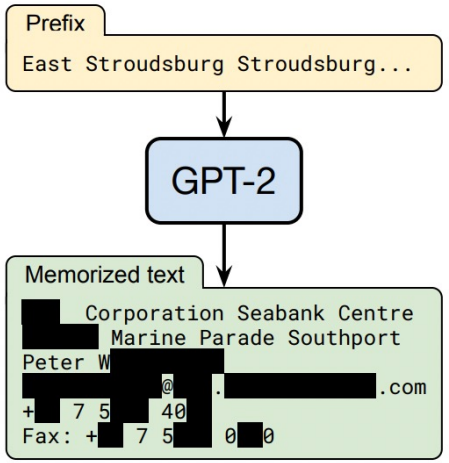


Figure 1: **Our extraction attack.** Given query access to a neural network language model, we extract an individual person's name, email address, phone number, fax number, and physical address. The example in this figure shows information that is all accurate so we redact it to protect privacy.

"Lottery Ticket Hypothesis" gives rise to sparsity

THE LOTTERY TICKET HYPOTHESIS: FINDING SPARSE, TRAINABLE NEURAL NETWORKS

Jonathan Frankle
MIT CSAIL
jfrankle@csail.mit.edu

Michael Carbin
MIT CSAIL
mcarbin@csail.mit.edu

ABSTRACT

"A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations"

We present an algorithm to identify winning tickets and a series of experiments that support the lottery ticket hypothesis and the importance of these fortuitous initializations. We consistently find winning tickets that are less than 10-20% of the size of several fully-connected and convolutional feed-forward architectures for MNIST and CIFAR10. Above this size, the winning tickets that we find learn faster than the original network and reach higher test accuracy.

The impact of sparsity

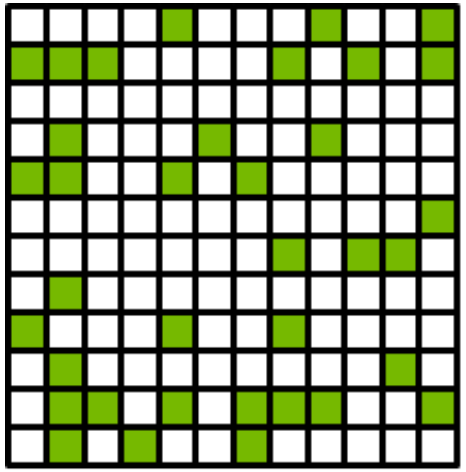
Sparsity Level ↑

Number of Ops 👍

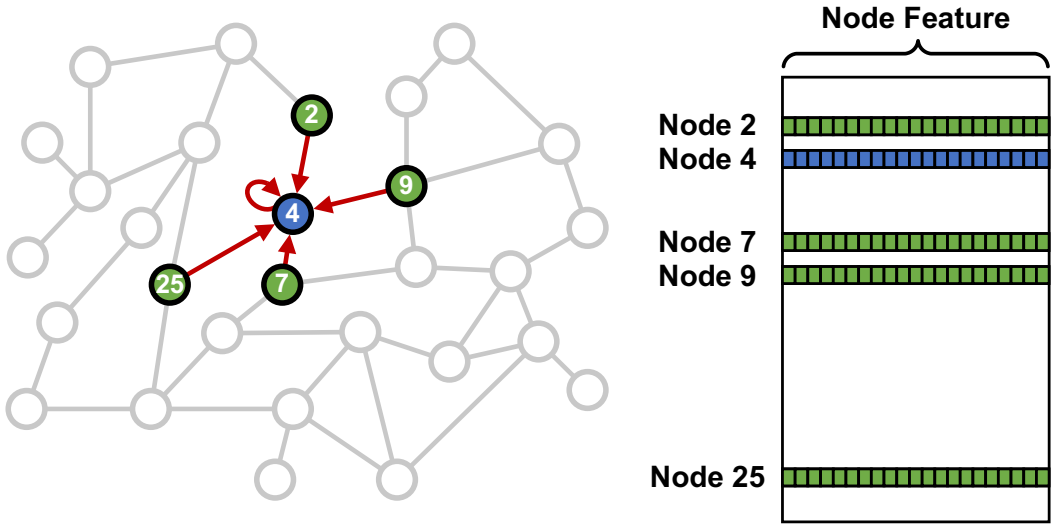
Accuracy 👎

Throughput ?

Two main modalities for accelerating DNN related sparse computation: focus of this tutorial



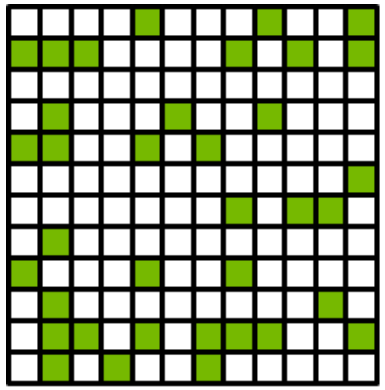
Sparse computation in the form of SpMM



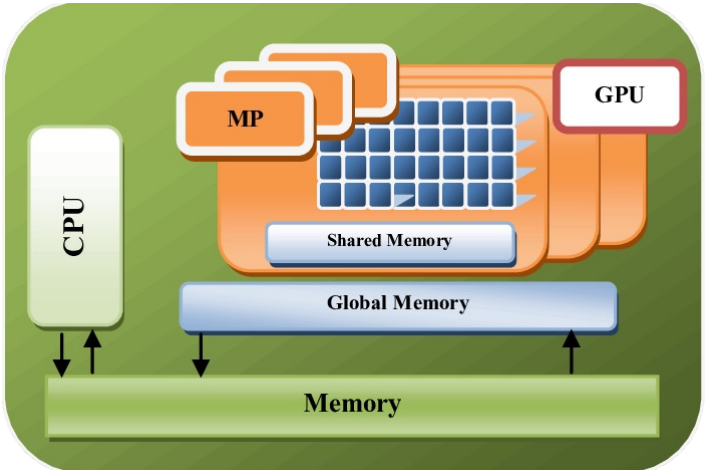
Sparse data access in the form of graph

Three levels of optimization opportunities for SpMM on heterogeneous systems

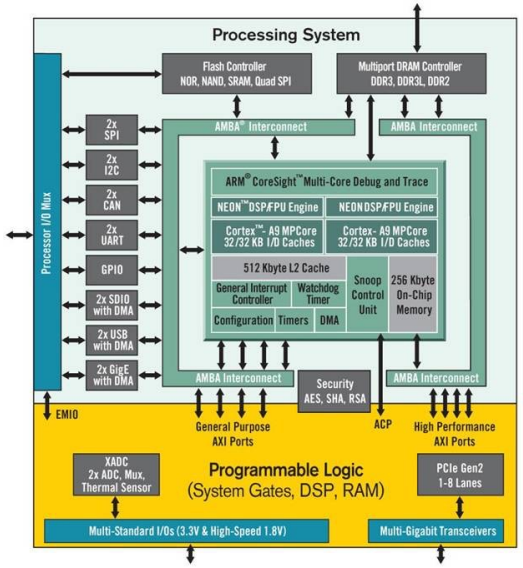
1. Optimize SpMM data structures and algorithms for generic hardware features



2. Co-optimize SpMM implementation for underlying hardware features

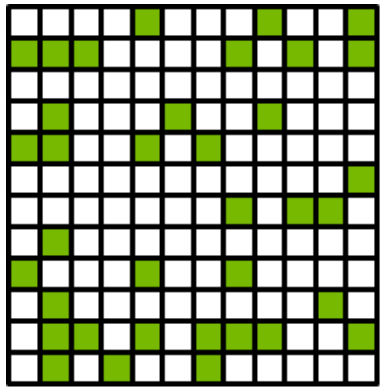


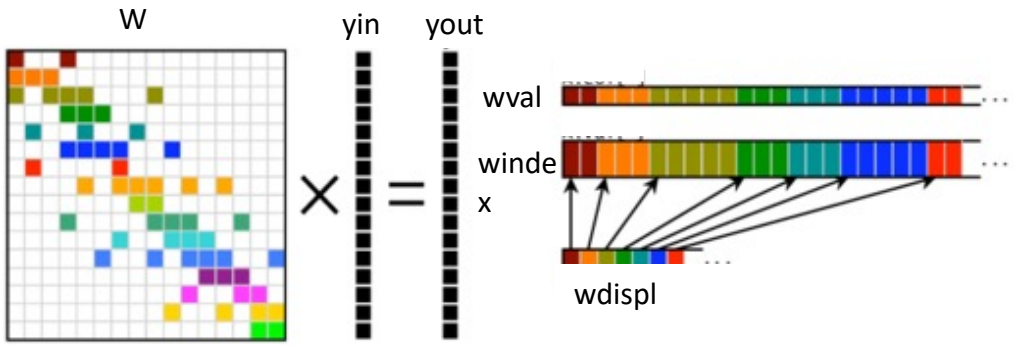
3. Optimize hardware with innovative architecture features



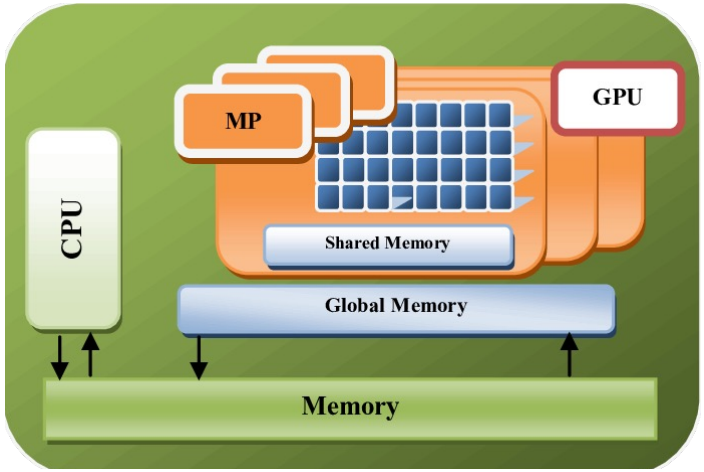
Tiled SpMM acceleration on GPUs – talk by Mert @ 1:50 pm

1. Optimize SpMM data structures and algorithms for generic hardware features

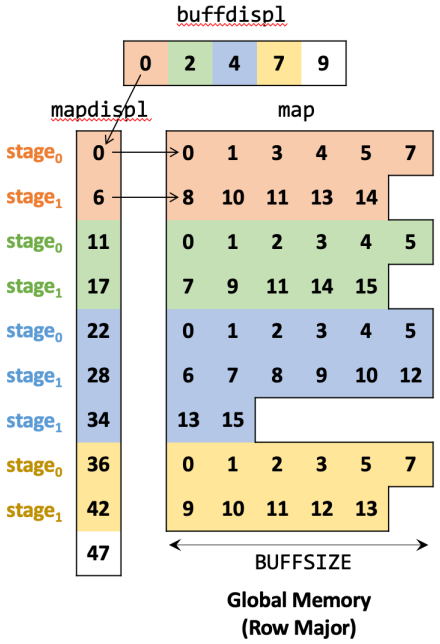




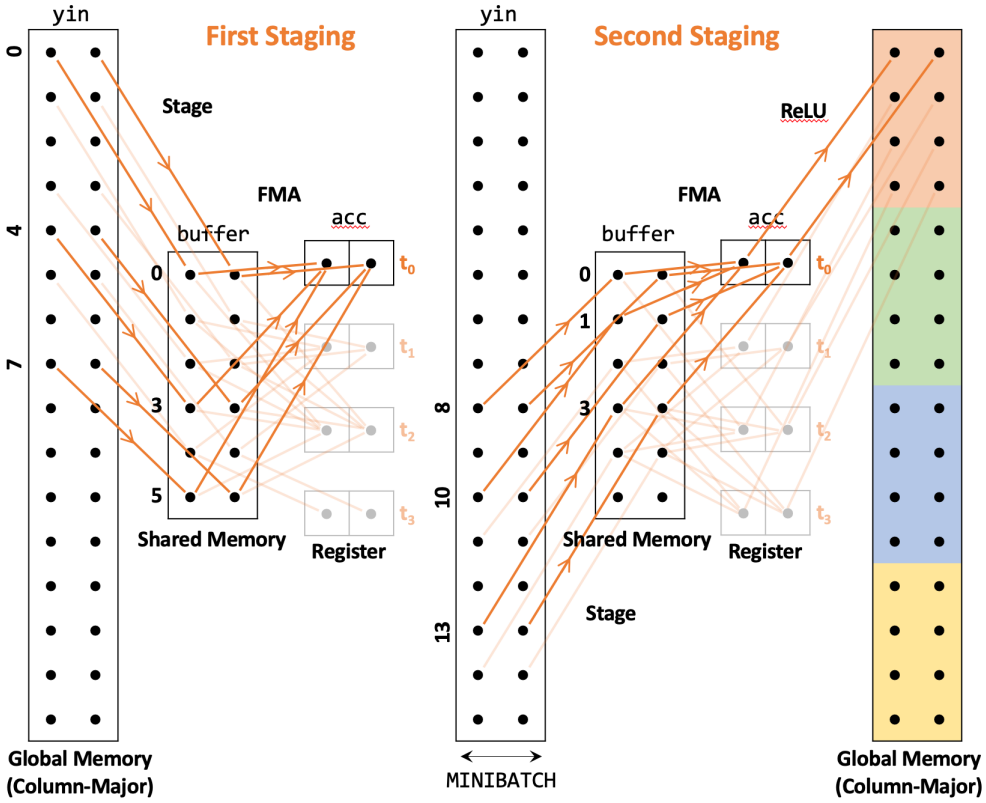
2. Co-optimize SpMM implementation for underlying hardware features



Intermediate Data Structures

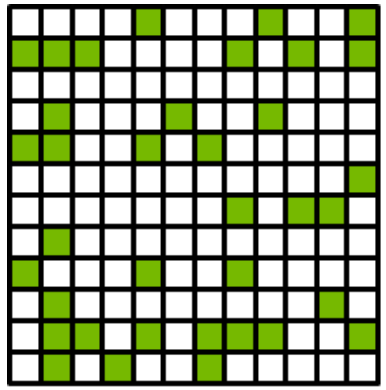


Multi-Level Input Buffering

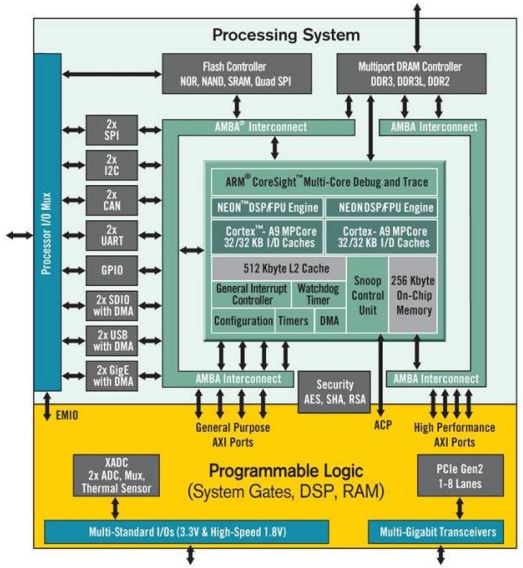


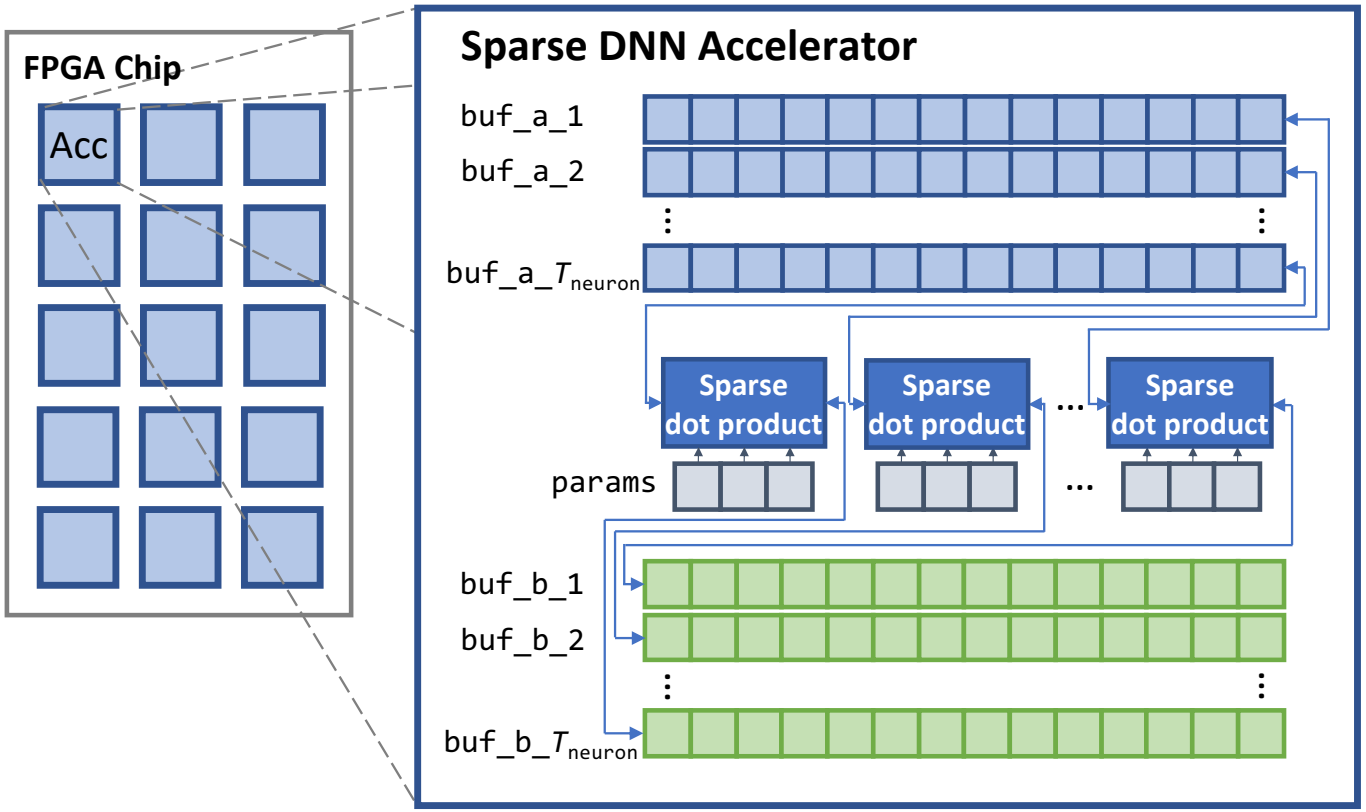
[Hidayetoglu et al: "At-Scale Sparse Deep Neural Network Inference With Efficient GPU Implementation," IEEE GraphChallenge 2020 (Champion)]

SpMM-based sparse DNN acceleration on FPGAs – talk by Sitao @ 2:45 pm



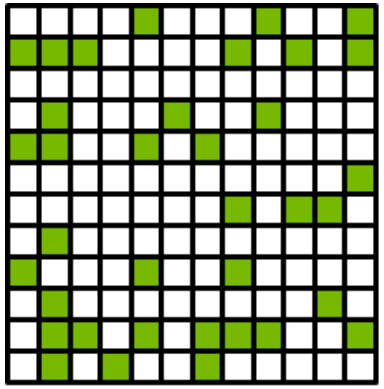
2. Co-optimize SpMM implementation for underlying hardware features



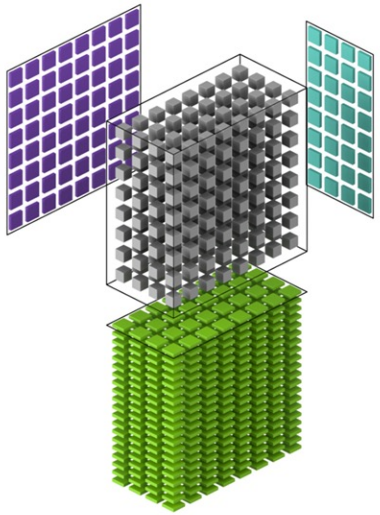


[Huang et al: "Accelerating Sparse Deep Neural Network Inference on FPGAs," IEEE GraphChallenge 2019 (Honorable Mention)]

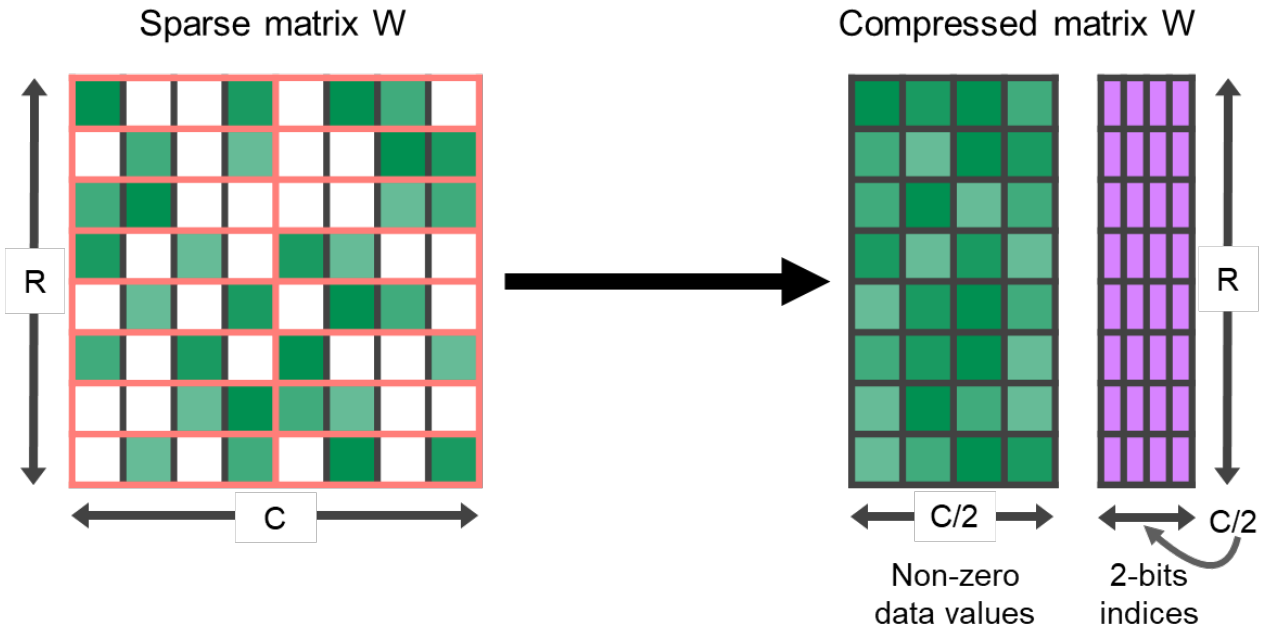
SpMM acceleration on 2:4 sparse tensor cores – talk by Jeff & Rakesh @ 4 pm



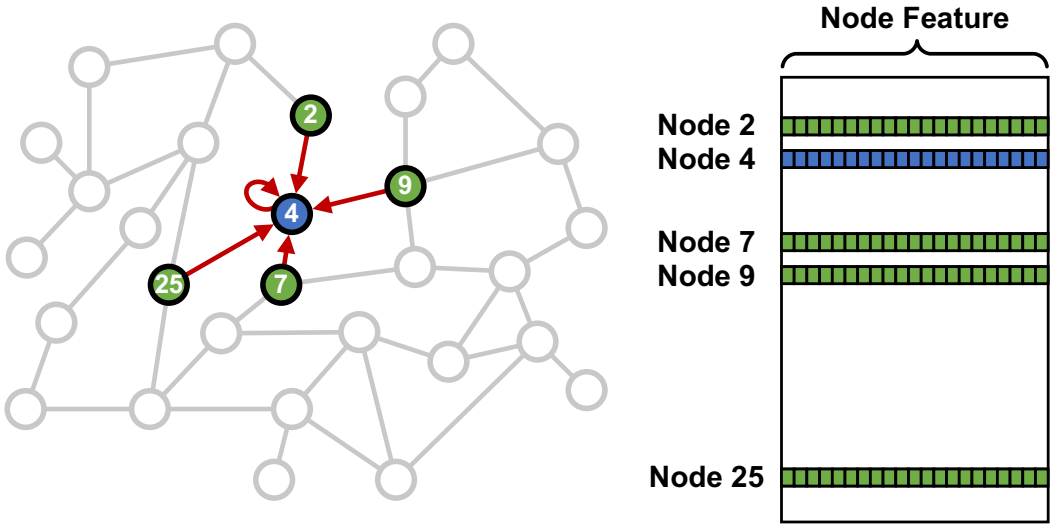
2. Co-optimize SpMM implementation for underlying hardware features



3. Optimize hardware with innovative architecture features

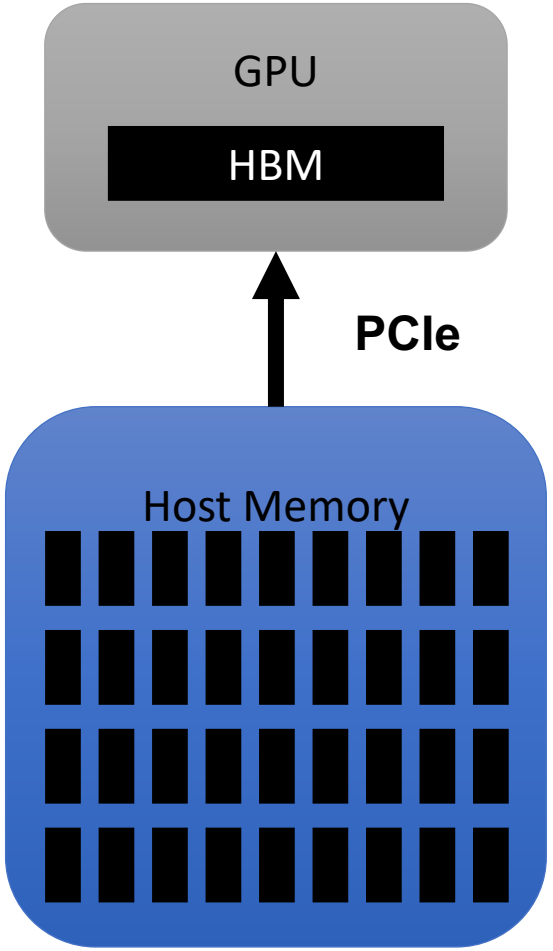


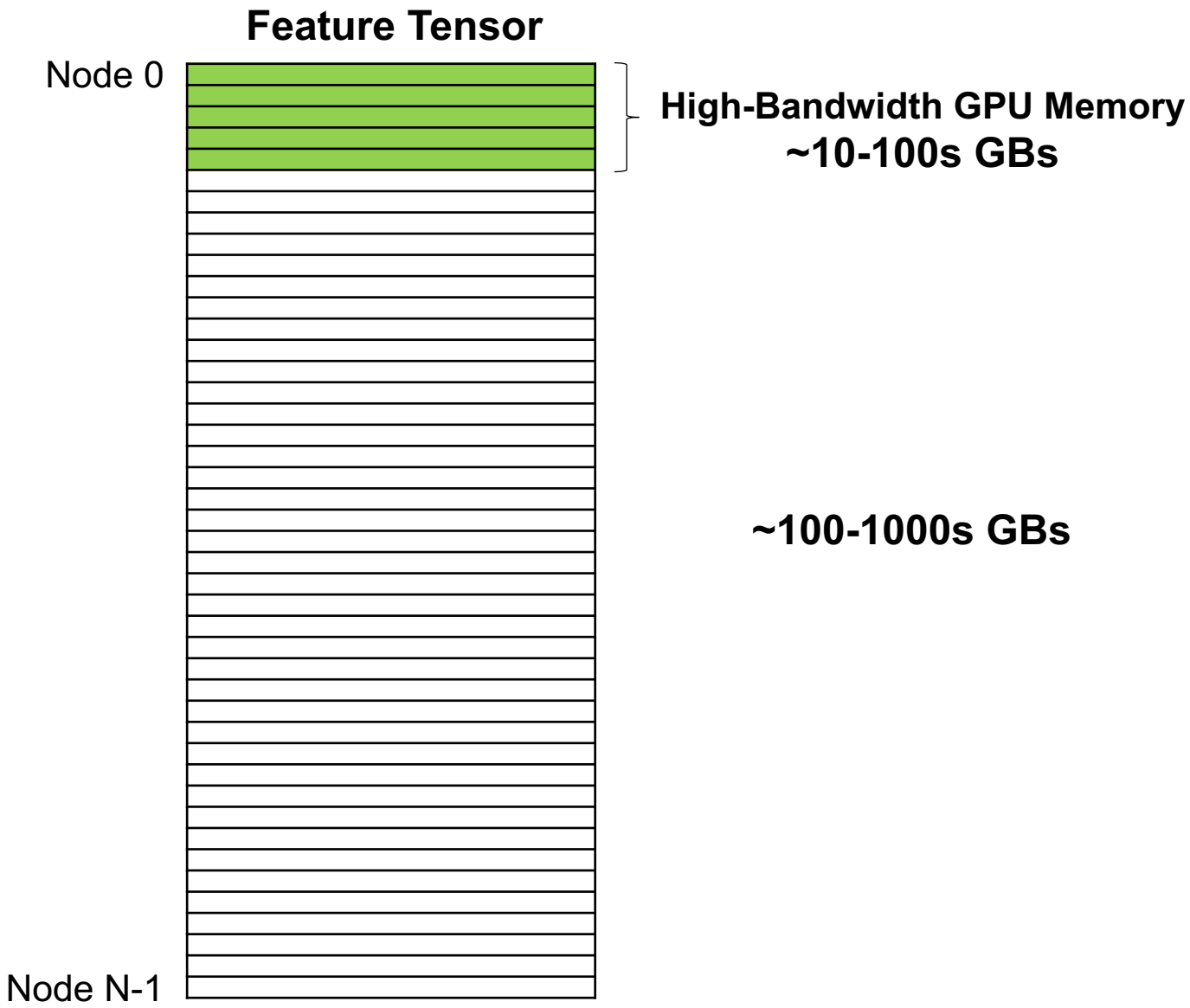
Acceleration graph related sparse computation on GPU-based heterogeneous systems



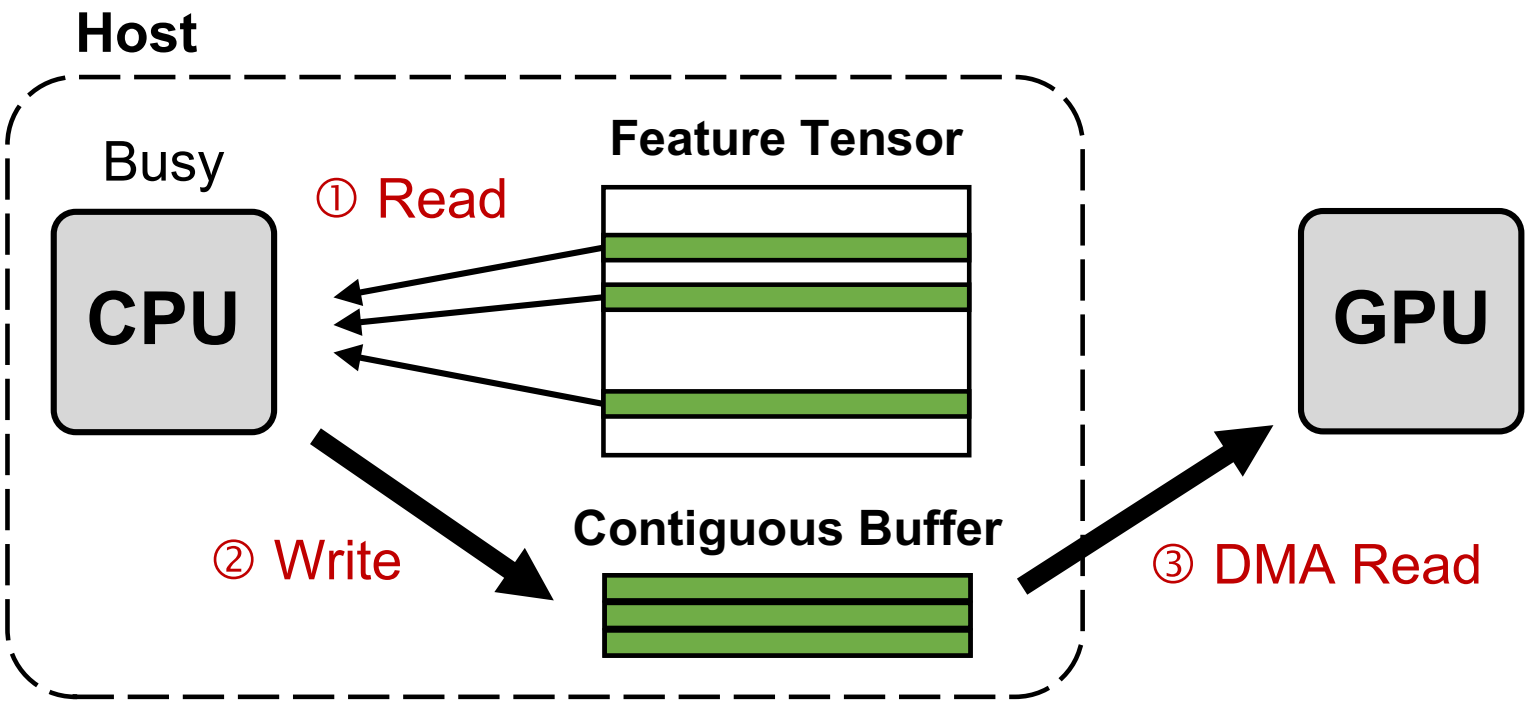
Sparse data access in the form of graph

Feature tensors of big graphs can't fit into GPU's limited memory





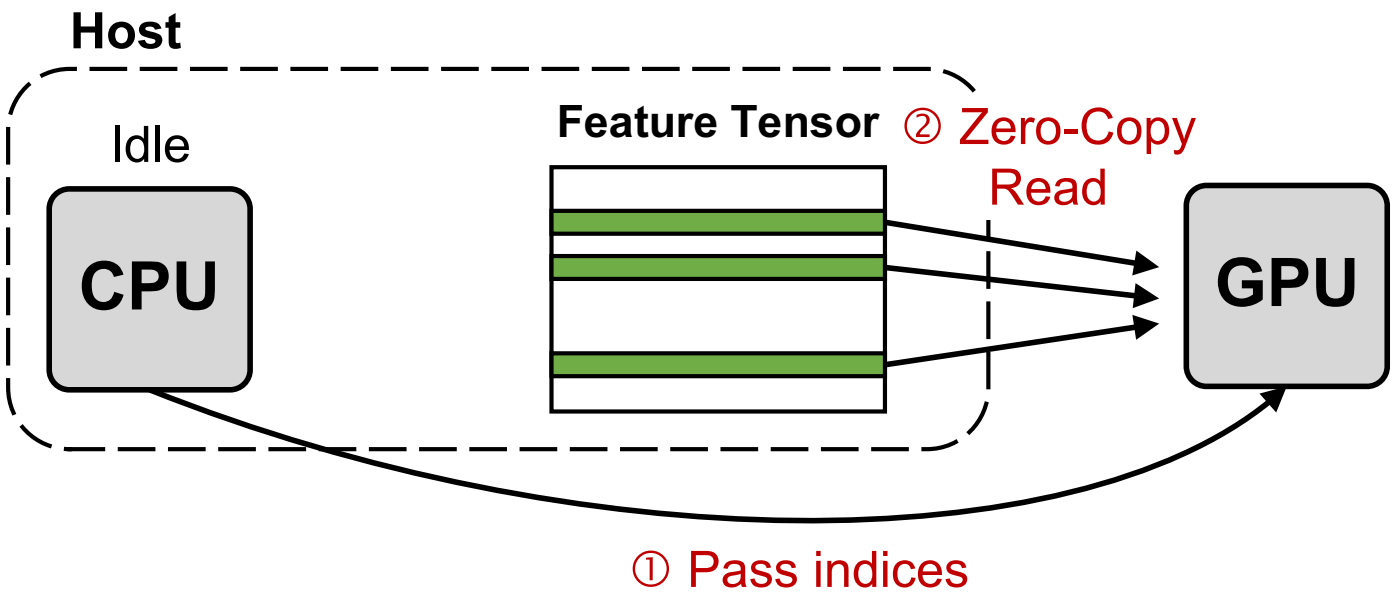
DMA transfer of data from CPU to GPU by transforming sparse data to dense data



Long data access latency and waste of host resources (consuming 3x more memory bandwidth)

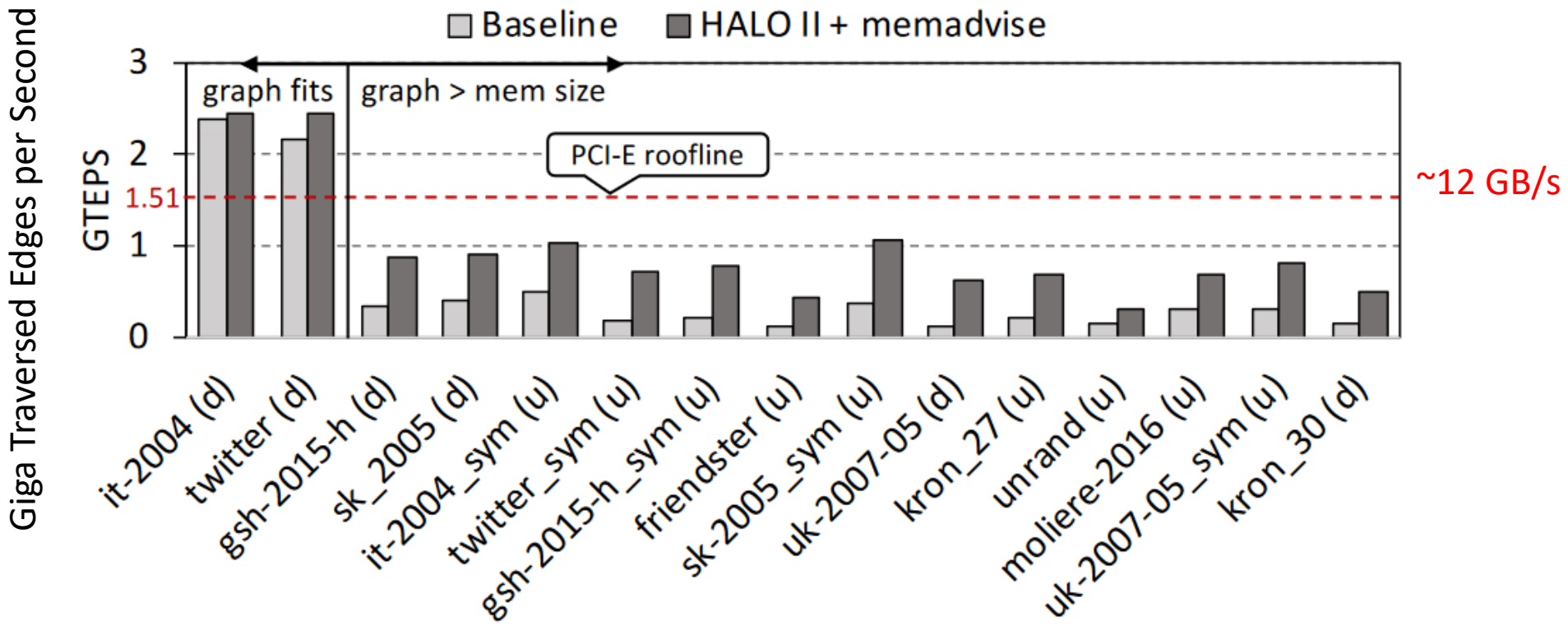
[Ying et al., *Graph Convolutional Neural Networks for Web-Scale Recommender Systems,* KDD 2018]

EMOGI: leverage zero-copy for GPUs to directly access node features on CPU memory



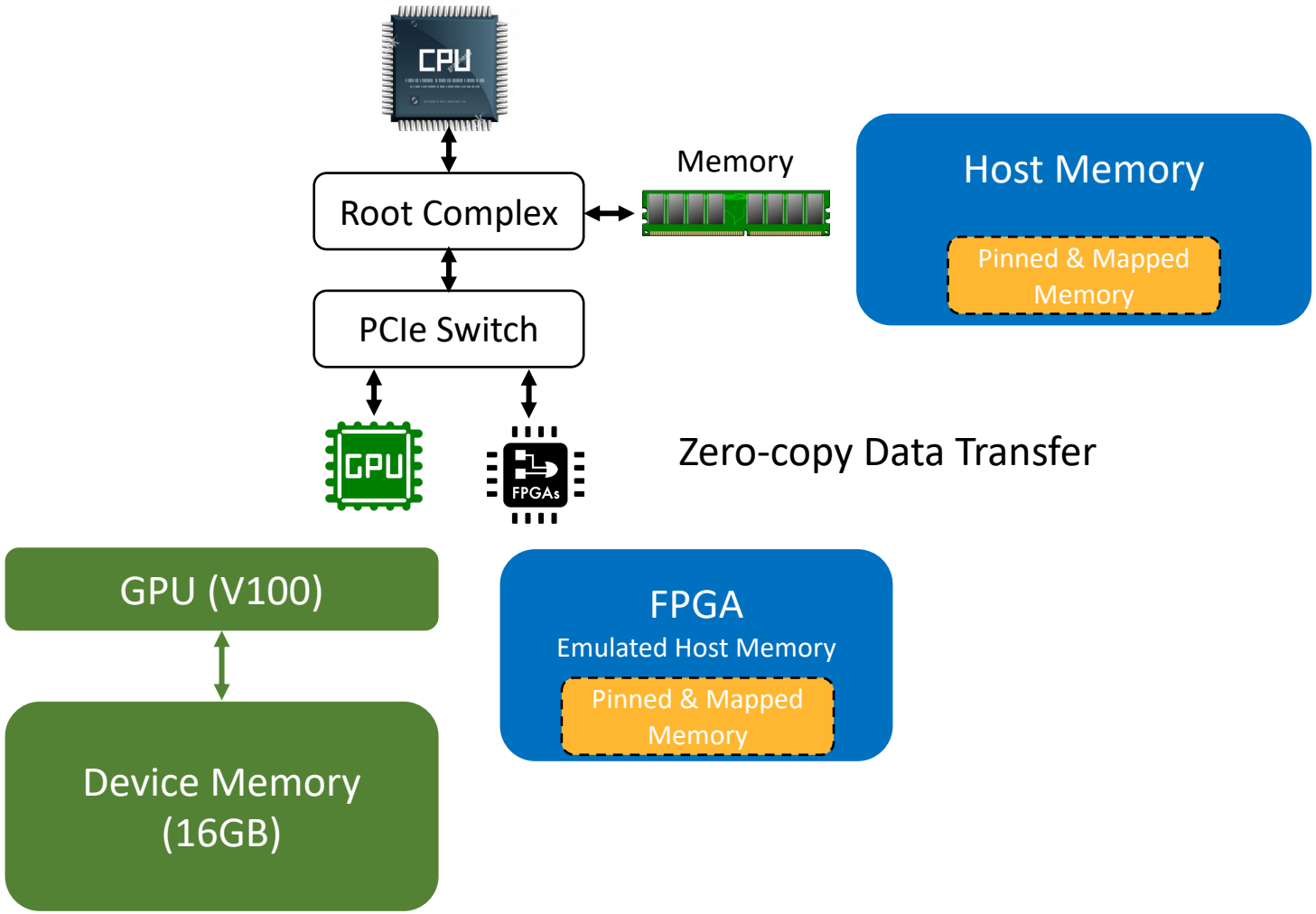
Myth: Accessing host memory through PCIe is slow

State-of-the-art graph processing was NOT even able to fully utilize the PCIe bandwidth



[P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader. “Traversing large graphs on GPUs with unified memory”. VLDB 2020.]

To understand why, we built a customized FPGA monitoring system on PCIe switch



FPGA Advertise itself as a Memory via BAR

↓

CPU mmap() the FPGA Memory to User Space

↓

CPU Passes the Pointer to GPU as Zero-Copy Region

↓

GPU's Access to Zero-Copy Region now Directed to FPGA

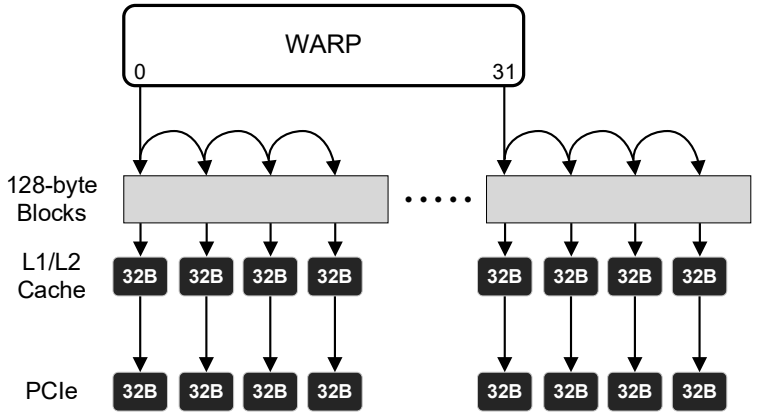
↓

FPGA is Programmed to Monitor PCIe Requests

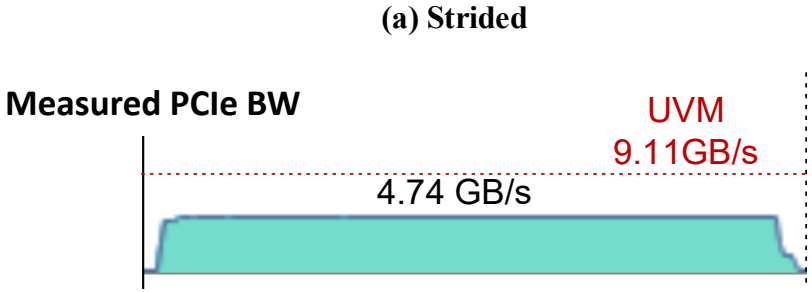
[Min, Mailthody, Qureshi, Xiong, Ebrahimi, and Hwu., EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs, VLDB'21]

Observation: tiled assignment results in 32-bytes PCIe packets

- Strided access: each warp thread takes an array of 32 floats, and iterates over one float at a time
- Whenever a thread crosses a 32-byte address boundary, a new 32-byte PCIe request is generated



(a) Strided



(a) Strided

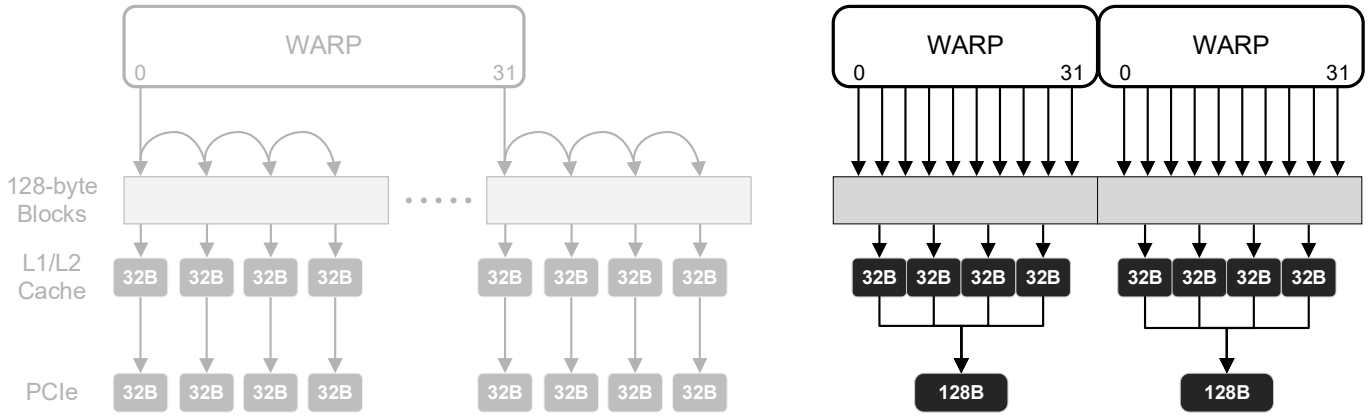
- Theoretical analysis of achievable bandwidth for our test system
  - The maximum # of outstanding PCIe 3.0 requests per specification is N = 256
  - The measured PCIe round-trip-time (RTT) is around 1.0 ~ 1.6 μs

$$PCIe_{BW,th} = \frac{32\ Bytes \times N}{RTT} = \frac{32\ Bytes \times 256}{1.6\ \mu s} \approx 4.76\ GB/s$$

[Min, Mailthody, Qureshi, Xiong, Ebrahimi, and Hwu., EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs, VLDB'21]

Observation: interleaved and aligned assignment results in coalesced 128-byte PCIe packets

- Warp threads access consecutive 32 floats that align with the 128-byte cacheline granularity



(a) Strided (b) Merged and Aligned

$$PCIe_{BW,th} = \frac{128\ Bytes \times 256}{1.6\ \mu s} \approx 19\ GB/s$$

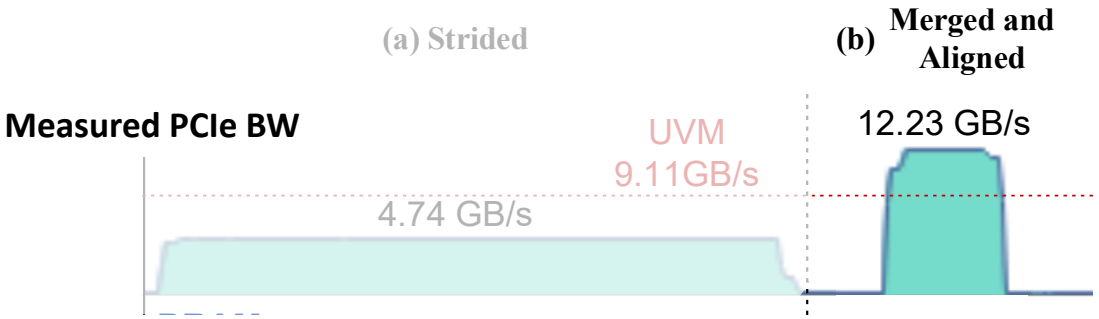
$$PCIe_{BW,spec} = 16\ GB/s$$

Ideally, we can now utilize the full PCIe BW

Considering PCIe package overhead

$$PCIe_{BW,limit} = \frac{128}{128 + 18} \times PCIe_{BW,spec} \approx 14\ GB/s$$

PCIe 3.0 transaction layer package header (minimum 18 bytes)

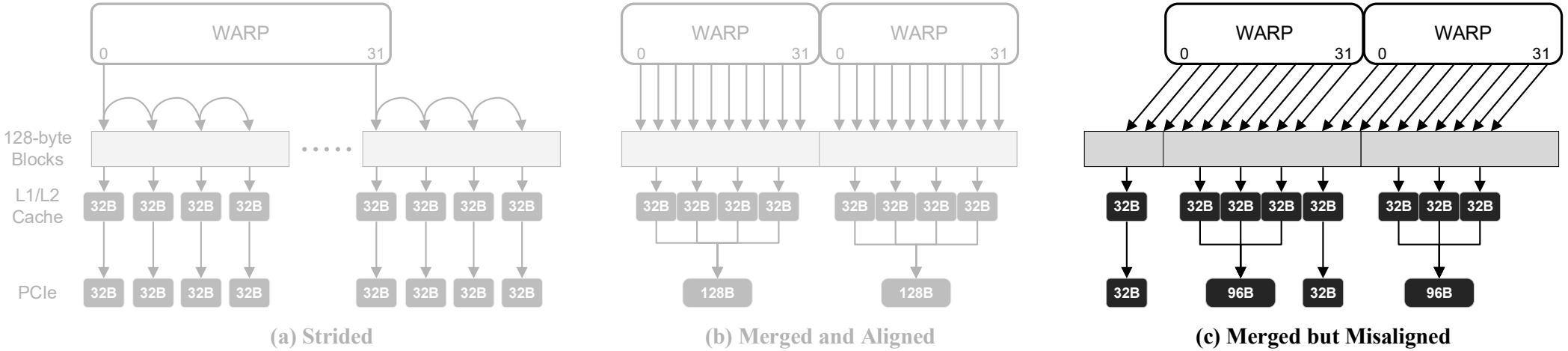


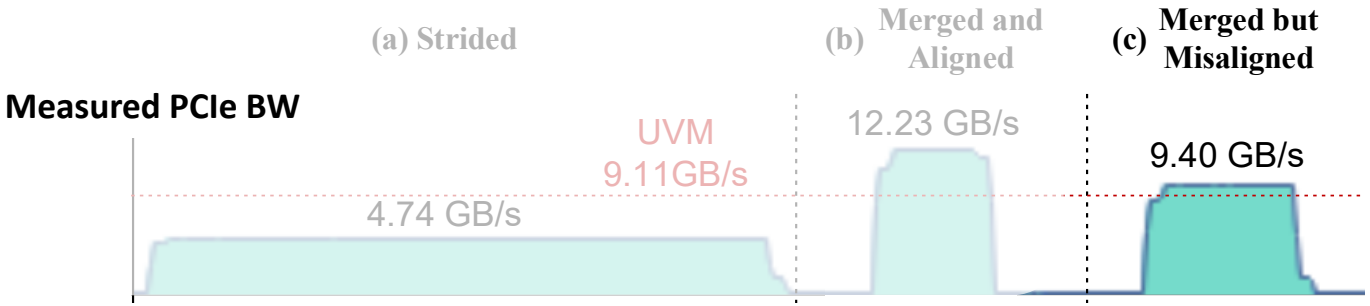
Zero-copy almost saturates the achievable PCIe BW limit

[Min, Mailthody, Qureshi, Xiong, Ebrahimi, and Hwu., EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs, VLDB'21]

Observation: tiled but non-aligned assignment results in suboptimal PCIe packet sizes

- Always guaranteeing 128-byte alignment for any data structure can be difficult



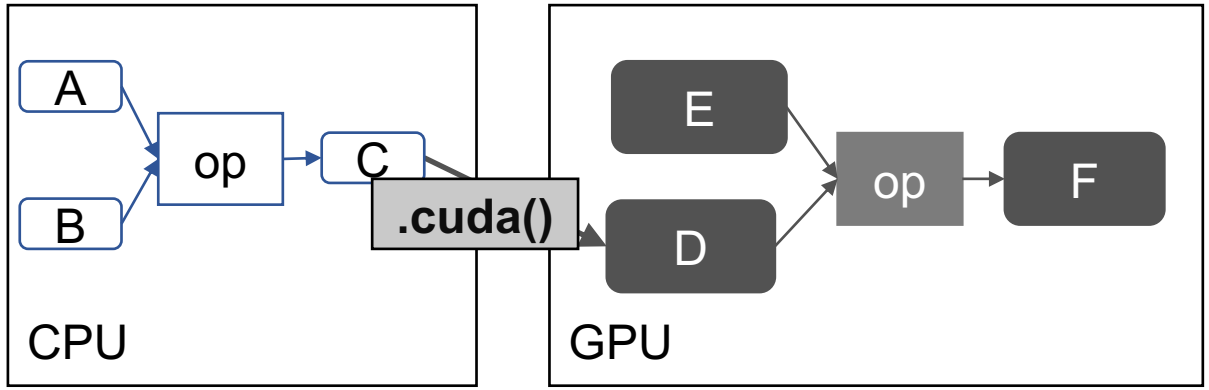


A warp may generate two separate PCIe requests to fetch a single 128-byte cacheline.

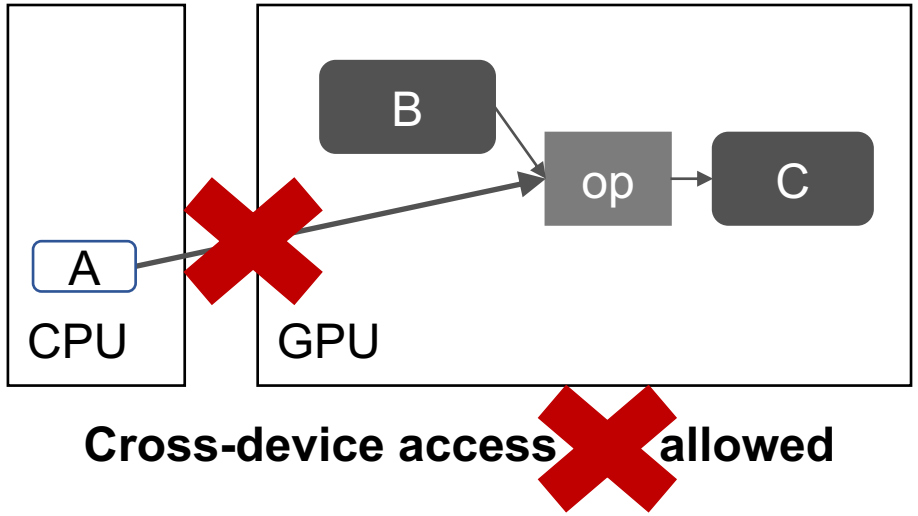
[Min, Mailthody, Qureshi, Xiong, Ebrahimi, and Hwu., EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs, VLDB'21]

Existing GNN frameworks (e.g., PyTorch/DGL) do not support zero-copy

- PyTorch always keeps data right next to the processing unit
  - CPU tensor in host memory and GPU tensor in GPU memory
  - No ways for GPU operators to directly access host memory
- Zero-copy tensor is transient
  - Resulting tensors should be GPU tensors in order to be stored in device memory



Native PyTorch tensors are bound to devices.



Cross-device access allowed

[Min, Wu, Huang, Hidayetoglu, Xiong, Ebrahimi, Chen, and Hwu., Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture, VLDB'21]

PyTorch-Direct: leverage EMOGI to make zero-copy available for all GNN models

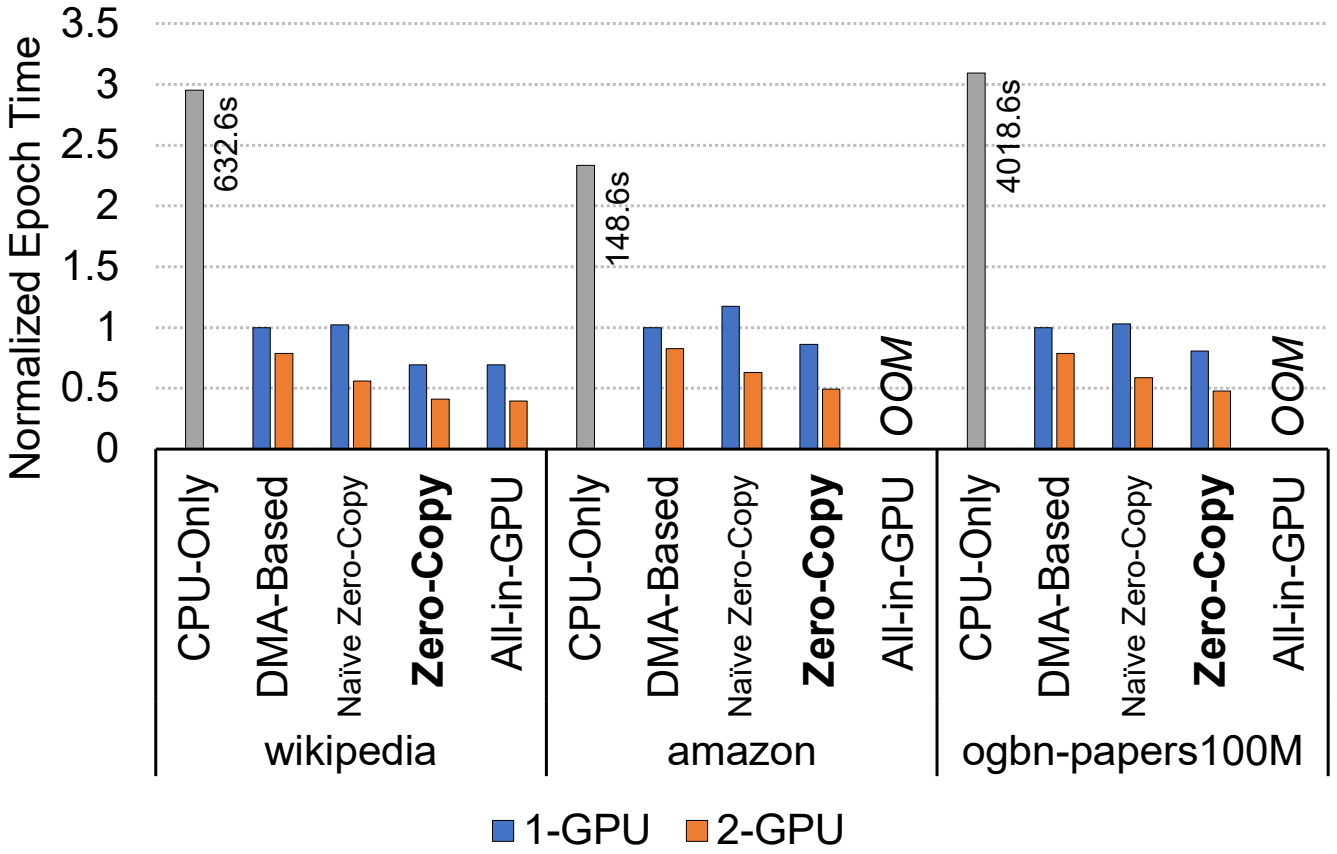
- Introduces new "unified tensor" data types in DGL
  - To enable the zero-copy direct data access paradigm for PyTorch
  - Compact and intuitive APIs for seamlessly integration
- Two affinity modes: GPU-affinitive and CPU-affinitive
  - No actual data movements when the affinity is switched

Tensor type	Executor	Storage
CPU	CPU	CPU Memory
CUDA	GPU	GPU Memory
GPU-Affinitive Unified	GPU	CPU Memory
Host-Affinitive Unified	CPU	CPU Memory

[Min, Wu, Huang, Hidayetoglu, Xiong, Ebrahimi, Chen, and Hwu., Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture, VLDB'21]

PyTorch-Direct improves end-to-end GNN training & even matches all-in-GPU method

Improvements of 16 – 44% for 1 GPU and 65 – 92% for 2 GPUs
Freed up CPU for other tasks (less memory access and less power consumption)

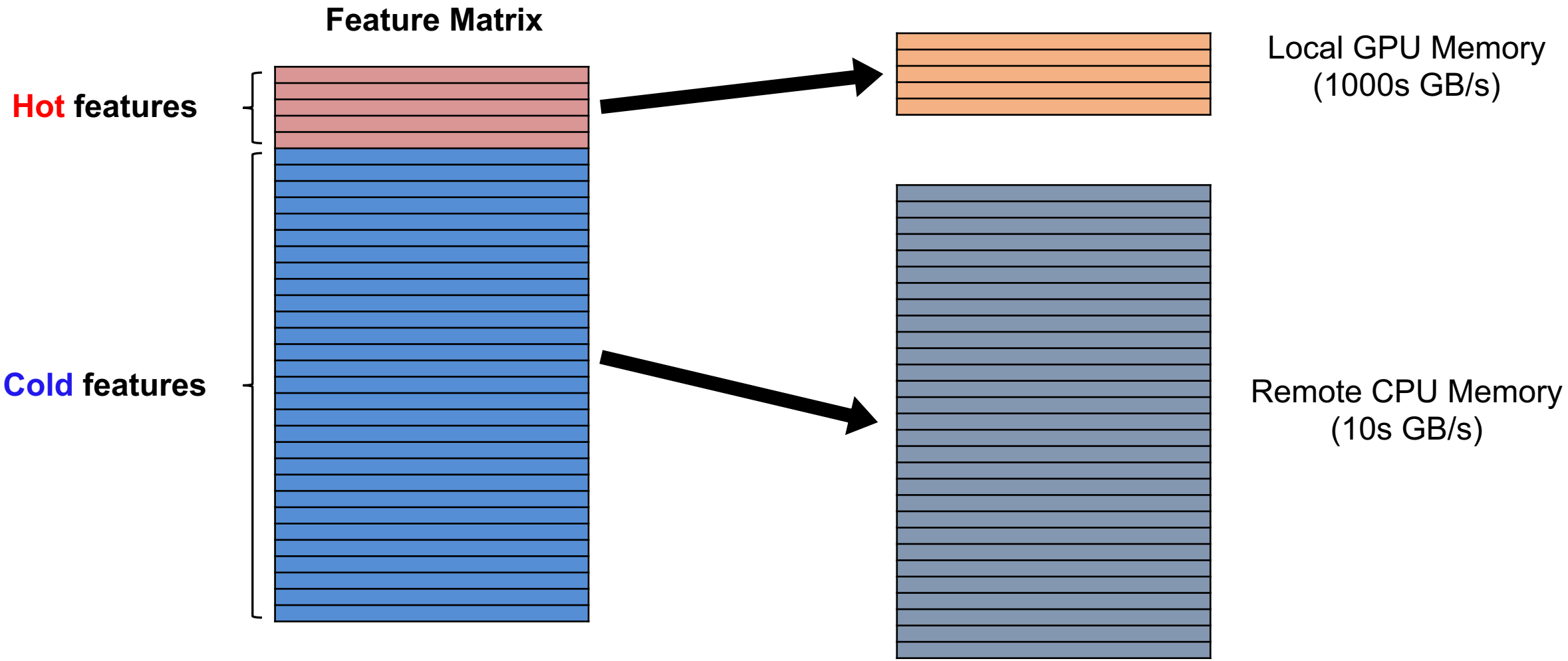


Dataset	# Features	GB Size	# Nodes	# Edges
Wikipediaa	315	17.1	13.6 M	437.2 M
Amazon	578	34.0	14.7 M	64.0 M
ogbn-paper	128	57	111 M	1.6 B

System	
CPU	AMD Threadripper 3960X 24C/48T
GPU	2x Ampere RTX 3090 24 GB
Memory	DDR4 3200 MHz 256 GB
CUDA	11.2
PyTorch	1.8.0-nightly
OS	Ubuntu 20.04.1, Linux Kernel 5.8.0

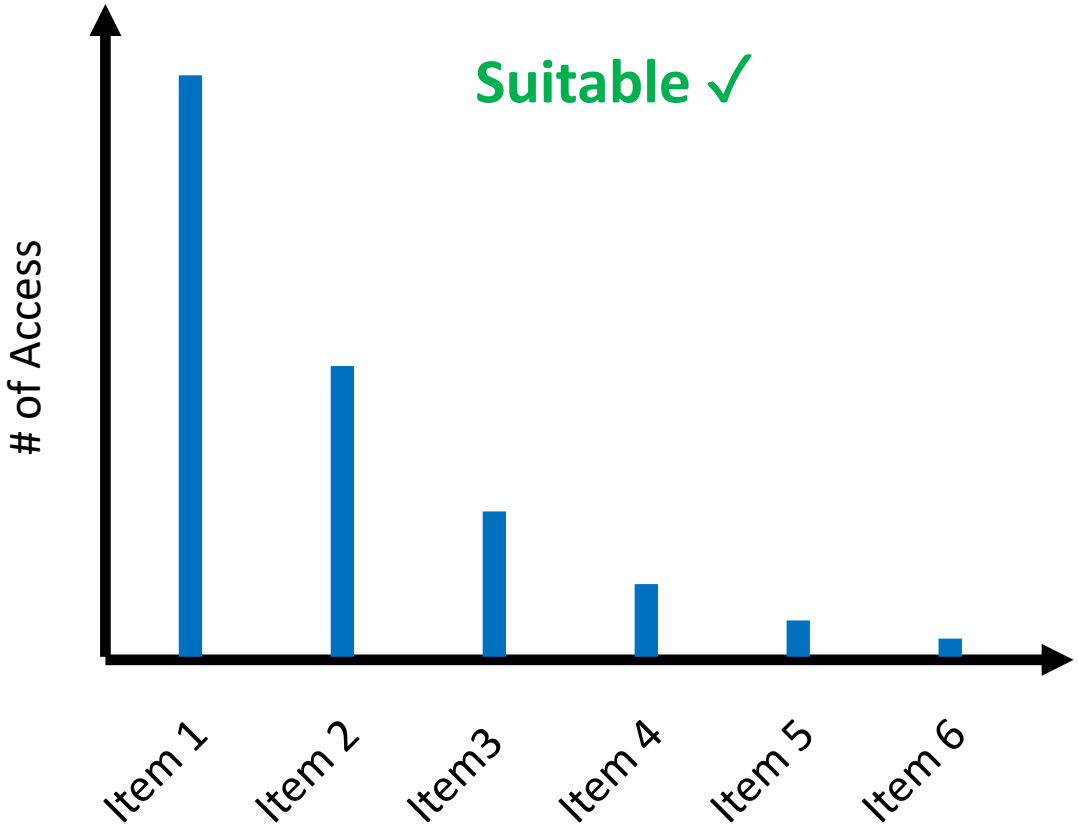
[Min, Wu, Huang, Hidayetoglu, Xiong, Ebrahimi, Chen, and Hwu., Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture, VLDB'21]

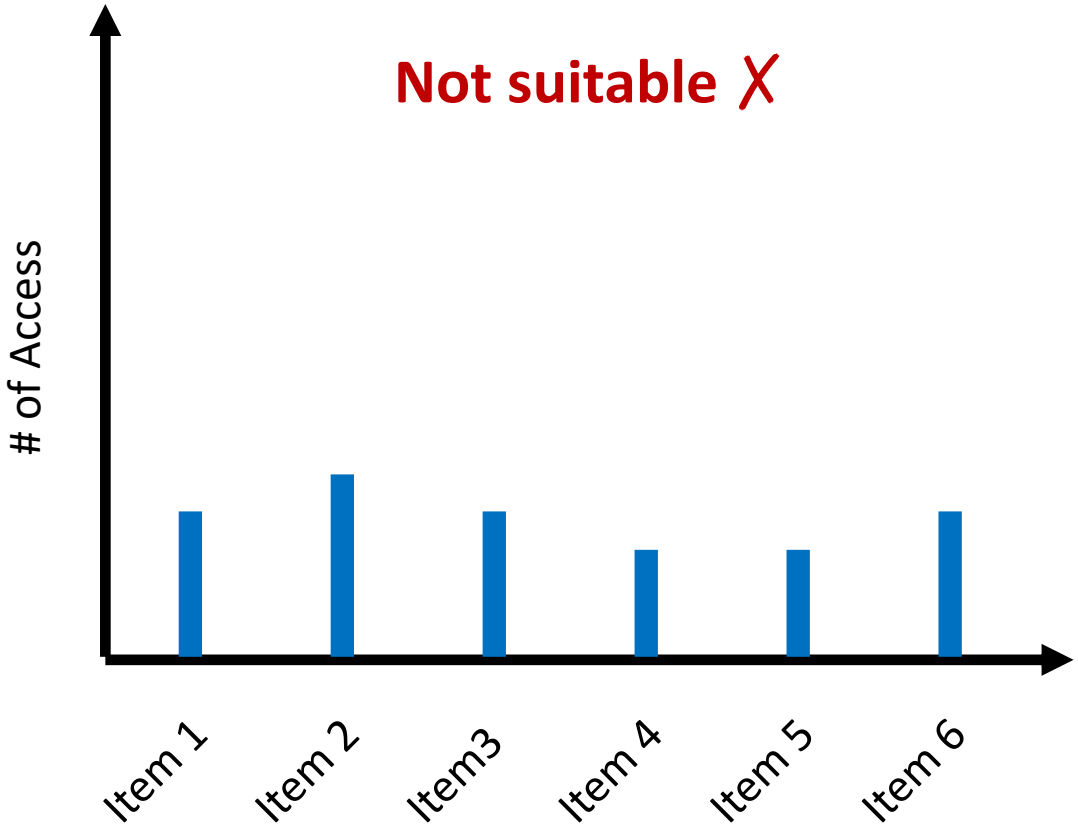
Data-tiering to determine what data should be placed in CPU vs GPU, even with zero-copy



[Min, Wu, Huang, Hidayetoglu, Xiong, Song, and Hwu., Graph Neural Network Training with Data Tiering, KDD'22]

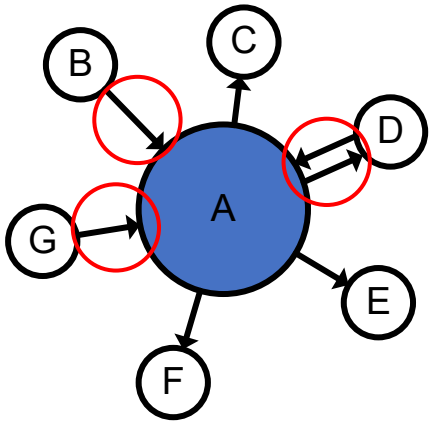
Data-tiering works best with huge imbalanced data access frequencies





[Min, Wu, Huang, Hidayetoglu, Xiong, Song, and Hwu., Graph Neural Network Training with Data Tiering, KDD'22]

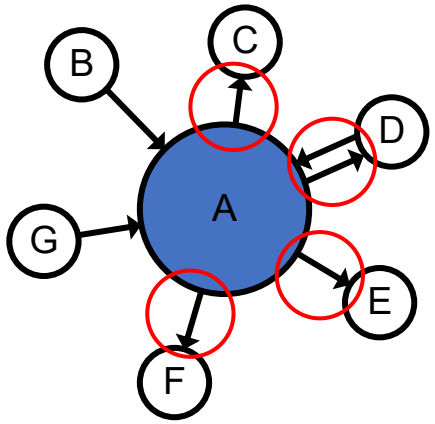
A statistical scoring method to identify more likely accessed nodes for GNN training



Score(A) = Score(B) + Score(D) + Score(G)

Score(A) = Score(A) / 4

(a) Pagerank



Score(A) = Score(C) + Score(D) + Score(E) + Score(F)

Score(A) = Score(A) / 3

(b) Reverse Pagerank

Non-important nodes:

$$\frac{1}{\textit{total number of nodes}}$$

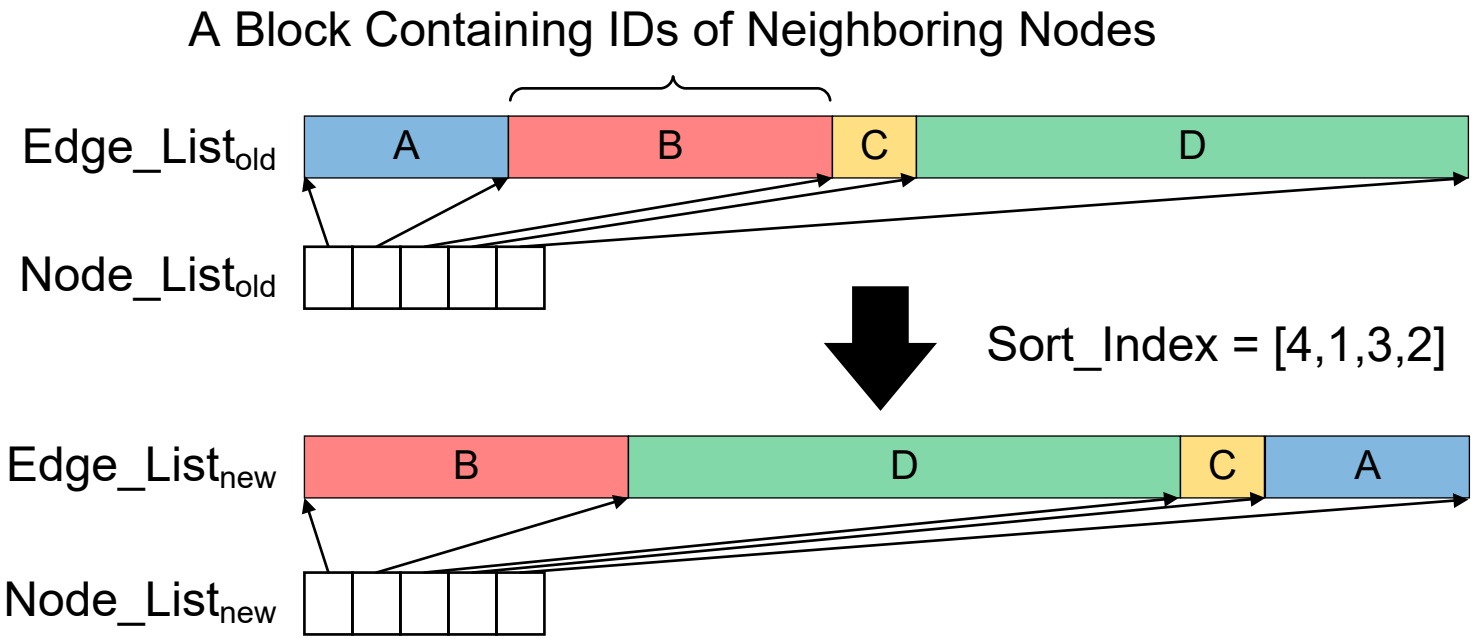
Important nodes (e.g., with labels):

$$\frac{1}{\textit{total number of nodes}} \times \textit{weight}$$

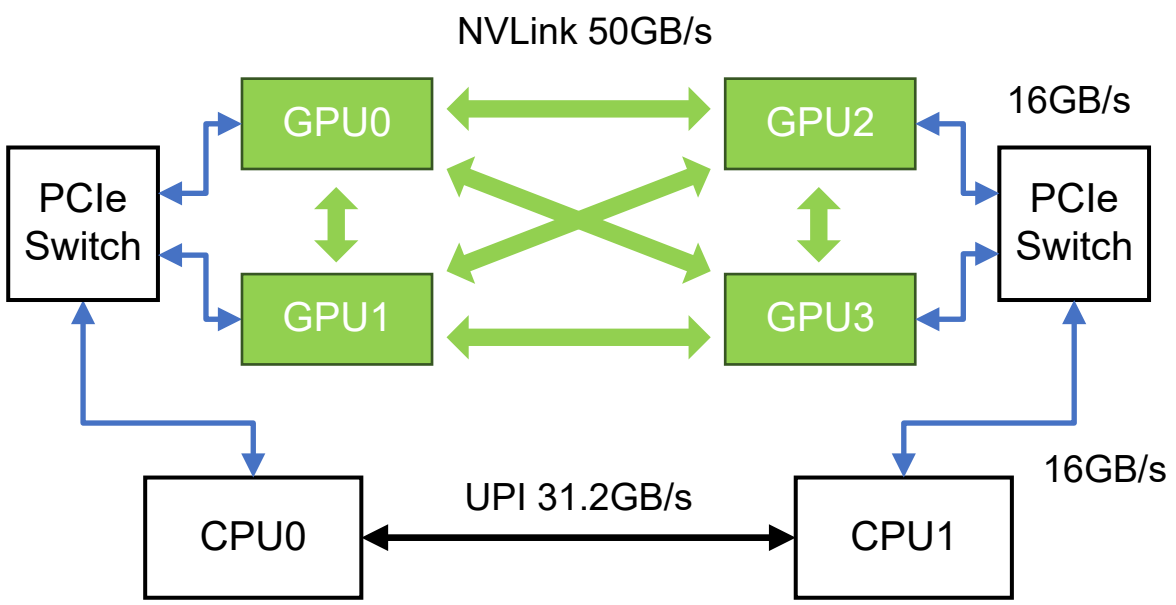
$$\textit{weight} = \frac{\textit{total number of nodes}}{\textit{total number of important nodes}}$$

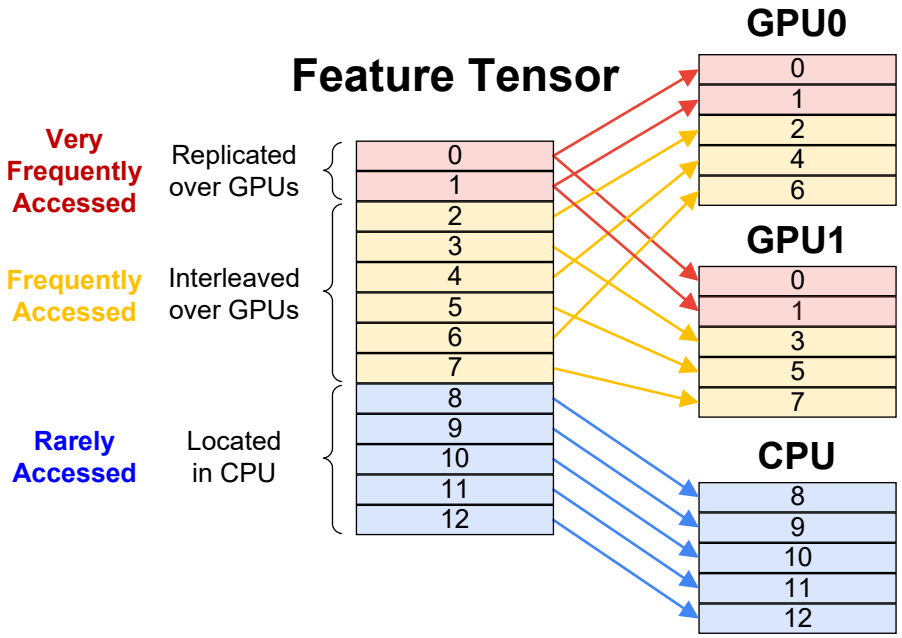
(c) Weighted Reverse Pagerank

Parallel implementation of reordering of adjacency matrix



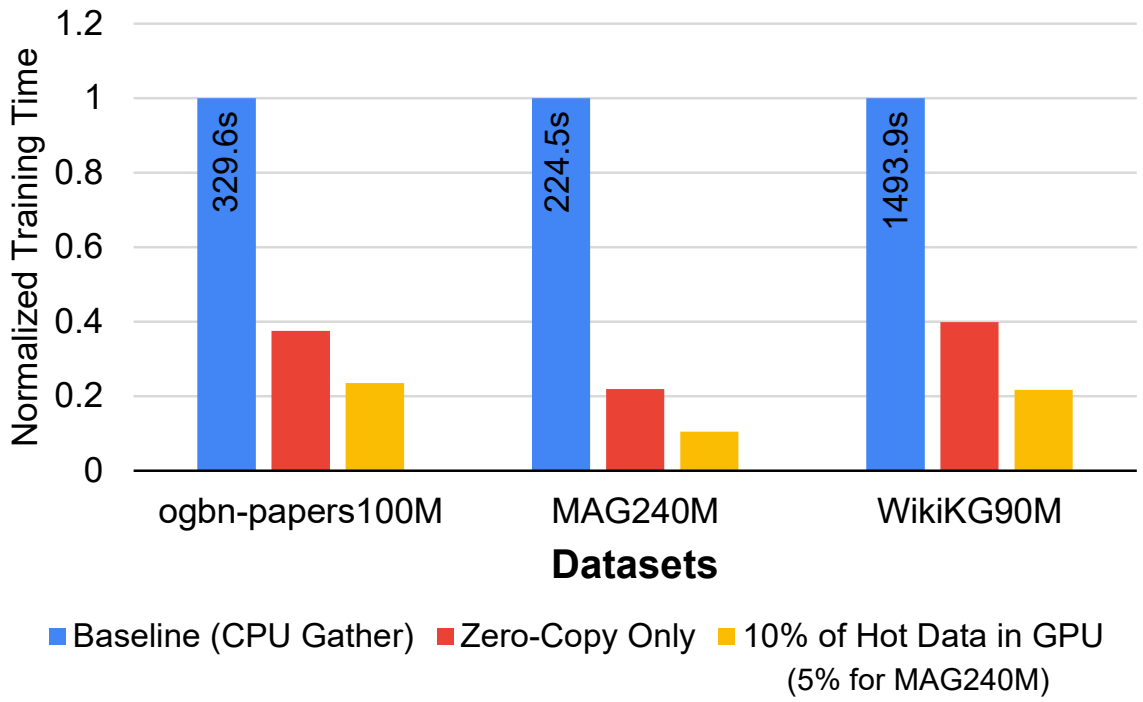
Data-tiering in a multiple GPU environment

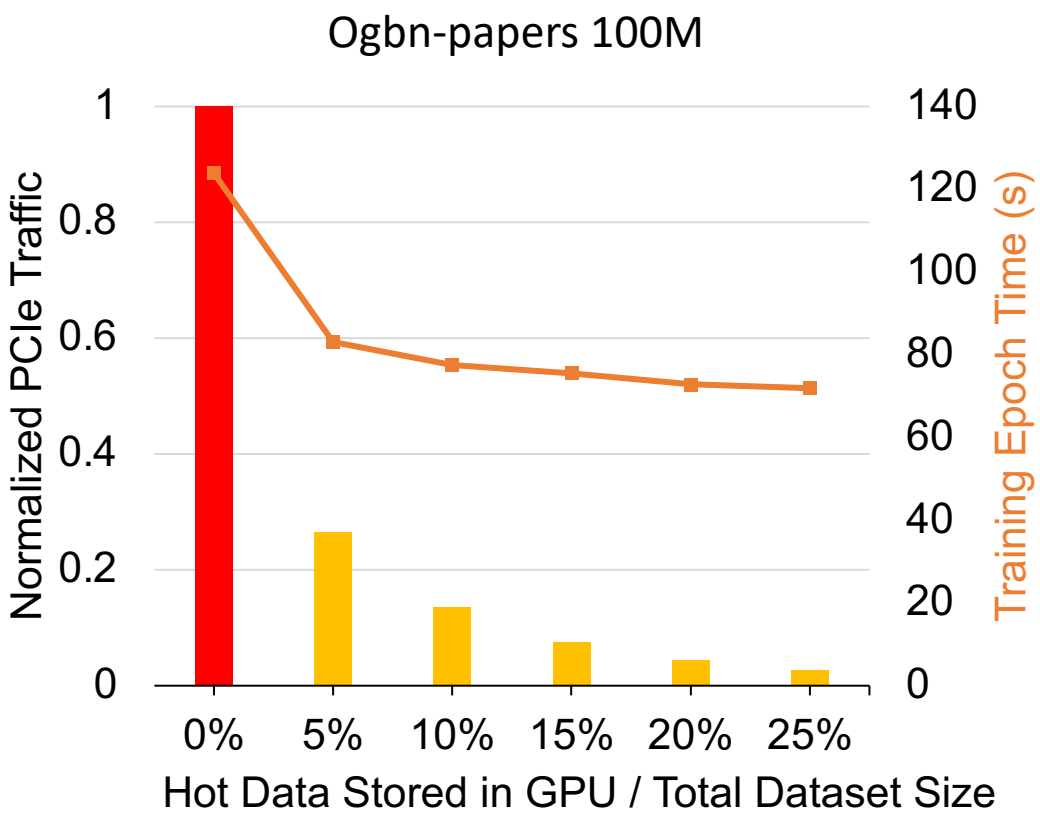




[Min, Wu, Huang, Hidayetoglu, Xiong, Song, and Hwu., Graph Neural Network Training with Data Tiering, KDD'22]

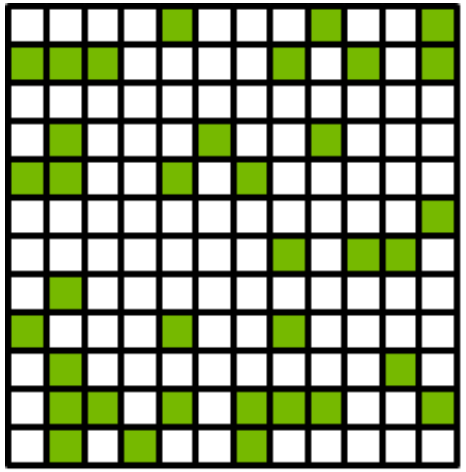
Data-tiering further improves GNN training by 1.6 – 2.1 X with much reduced PCIe traffic



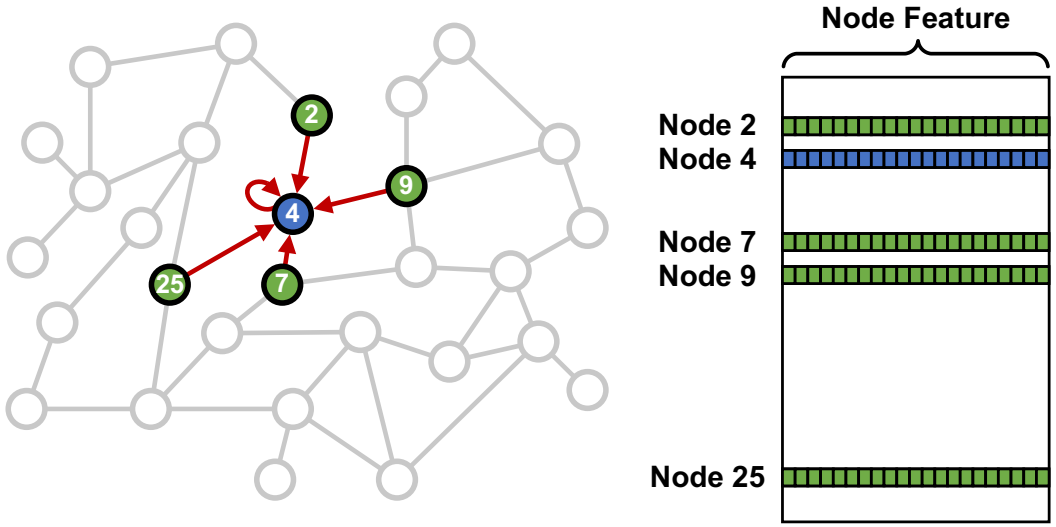


[Min, Wu, Huang, Hidayetoglu, Xiong, Song, and Hwu., Graph Neural Network Training with Data Tiering, KDD'22]

Take-away messages



Sparse computation in the form of SpMM



Sparse data access in the form of graph

- EMOGI: zero-copy to allow GPU-centric sparse data access (VLDB'21)
- PyTorch-Direct: unified tensor with zero-copy for GNN training (VLDB'21)
- Data-tiering: model-driven data placement for further performance (KDD'22)

Enjoy the rest of the tutorial

1. Opening remarks and overview of sparsity in ML – 1 pm
2. Tiled SpMM and its performance model on GPUs – 1:50 pm
3. Sparse deep neural network inference on FPGAs – 2:45 pm
4. Break (30 mins)
5. 2:4 Sparsity on GPU Tensor Cores – 4 pm
6. Future work and closing remarks – 4:45 pm

Tiled SpMM and its Performance Model on GPUs

Mert Hidayetoglu

University of Illinois at Urbana-Champaign

Organizers: Mert Hidayetoglu, Wen-mei Hwu, Jinjun Xiong, Rakesh Nagi,

Jeff Pool, Sitao Huang, Vikram Mailthody

Tutorial | MLSys 2022 | Aug 30, 2022

https://mlsys.org/virtual/2022/tutorial/2199

Slack for discussions: https://bit.ly/3KLQEUN

Outline

1. Opening remarks and overview of sparsity in ML – 1 pm
2. Tiled SpMM and its performance model on GPUs – 1:50 pm
3. Sparse deep neural network inference on FPGAs – 2:45 pm
4. Break (30 mins)
5. 2:4 Sparsity on GPU Tensor Cores – 4 pm
6. Future work and closing remarks – 4:45 pm

Fundamental Matrix Multiplication Operations

Fundamental Matrix Multiplication Operations

DGEMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Dense Dense Dense

Fundamental Matrix Multiplication Operations

DGEMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Dense Dense Dense

SpMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Sparse Dense Dense

Fundamental Matrix Multiplication Operations

DGEMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Dense Dense Dense

SpMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Sparse Dense Dense

SpGEMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Sparse Sparse Sparse

Fundamental Matrix Multiplication Operations

DGEMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Dense Dense Dense

SpMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Sparse Dense Dense

SpGEMM

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$$

Sparse Sparse Sparse

Fundamental Matrix Multiplication Operations

DGEMM

$$A \times B = C$$

Dense Dense Dense

SpMM

$$A \times B = C$$

Sparse Dense Dense

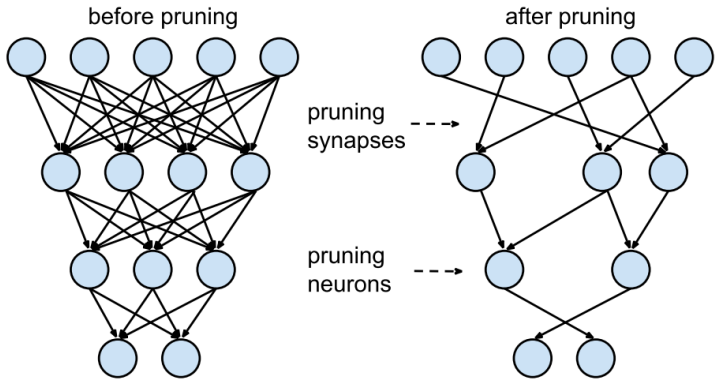
Sparse Connection Layers
Graph Neural Networks
Sparse Attention Layers

SpGEMM

$$A \times B = C$$

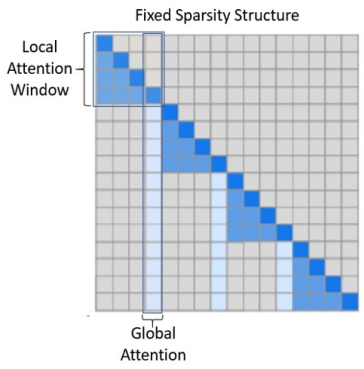
Sparse Sparse Sparse

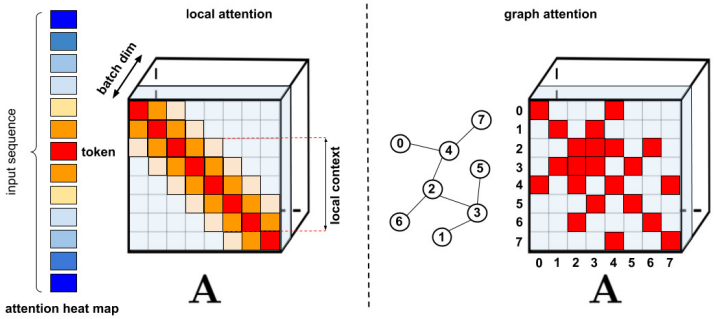
Sparse Neural Networks



[Lecun et al. NIPS'89] [Han et al. NIPS'15]

Sparse Attention Layers



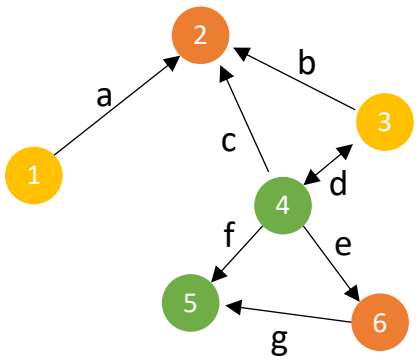


Google ai, "Rethinking Attention with Performers,"

Child, et al., "Generating Long Sequences with Sparse Transformers," Arxiv

Graph Neural Networks

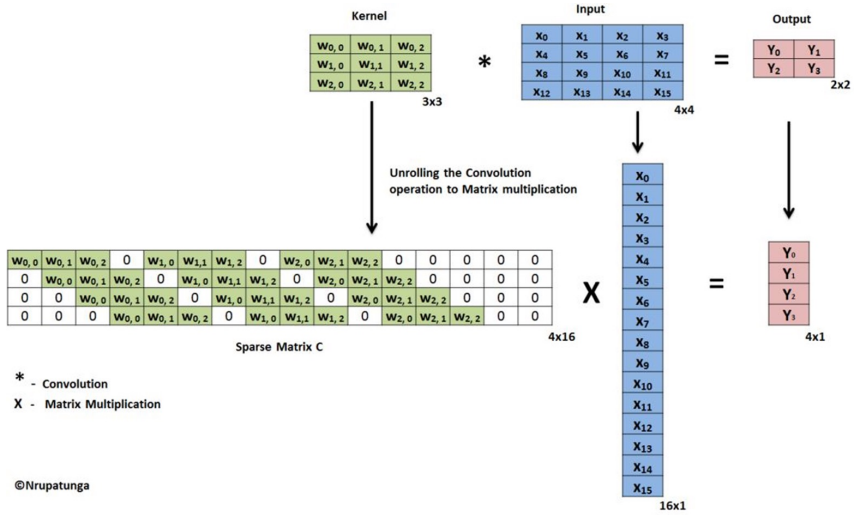
Graph Representation



Matrix Representation

	1	2	3	4	5	6
1		a				
2						
3		b		d		
4		c	d		f	
5						e
6					g	

Convolutional Neural Networks



Sparsity Level ↑

Memory Footprint ↓ 👍

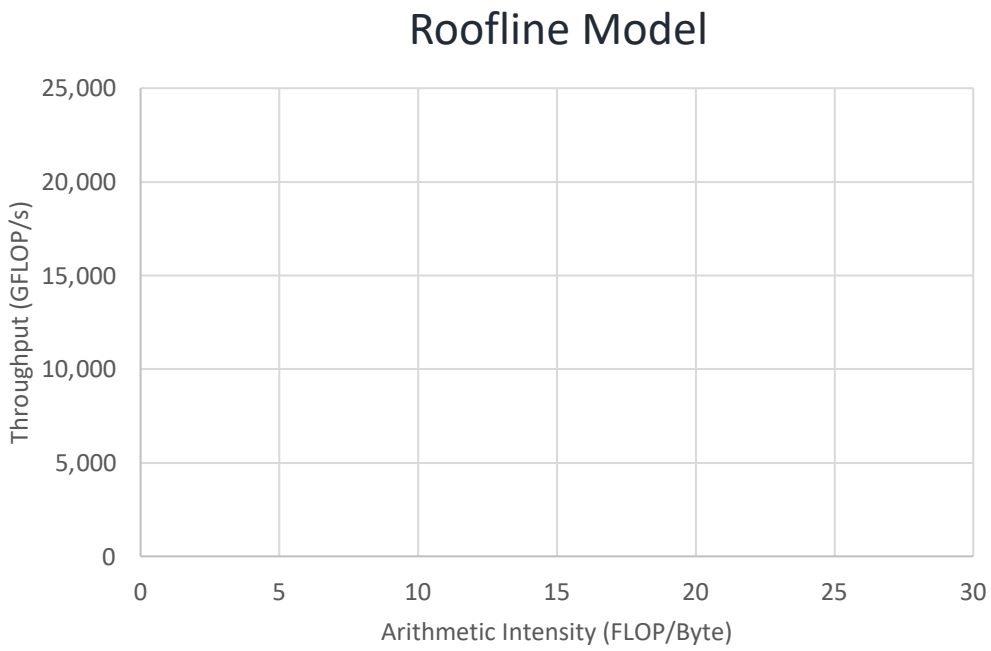
Number of Ops ↓ 👍

Accuracy ↓ 👎

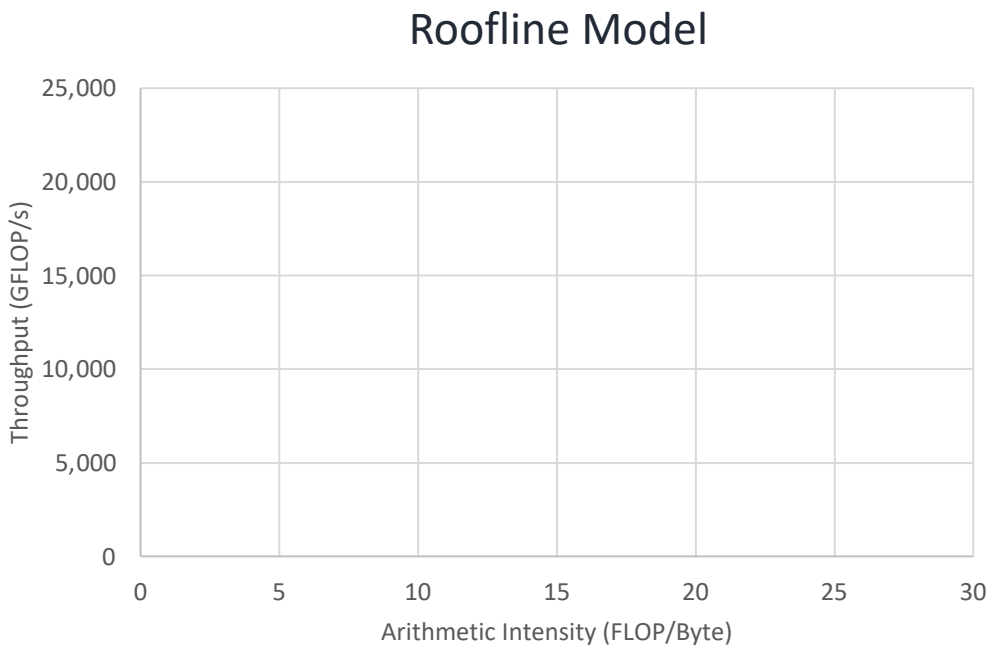
Throughput ?

The Roofline Model

The Roofline Model



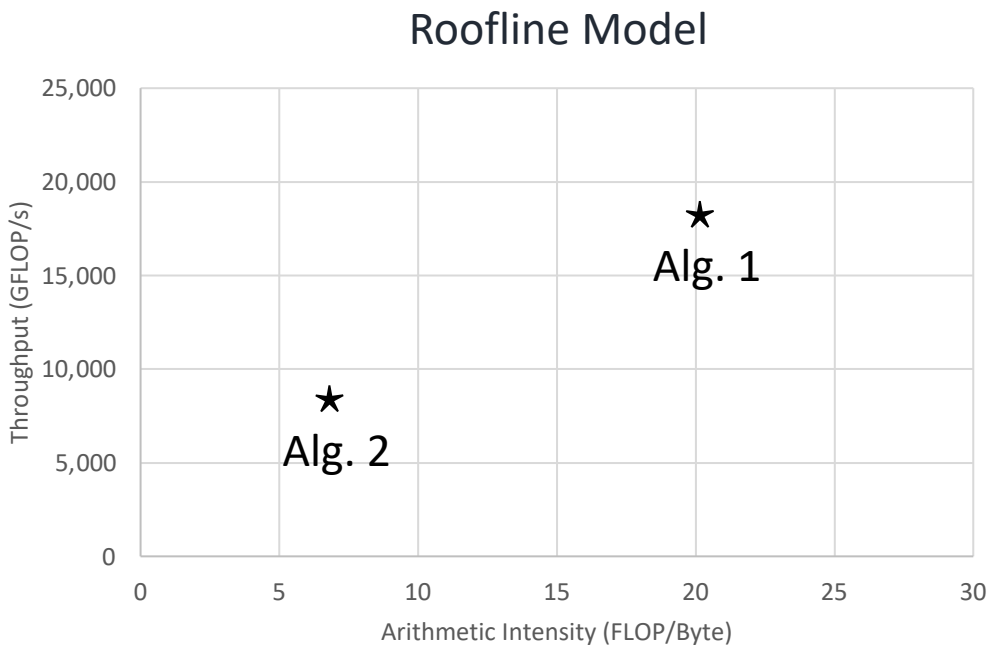
The Roofline Model



Arithmetic Intensity

$$I = \frac{\sum \text{Floating Point Operations (GFLOP)}}{\sum \text{Memory Accesses (GB)}}$$

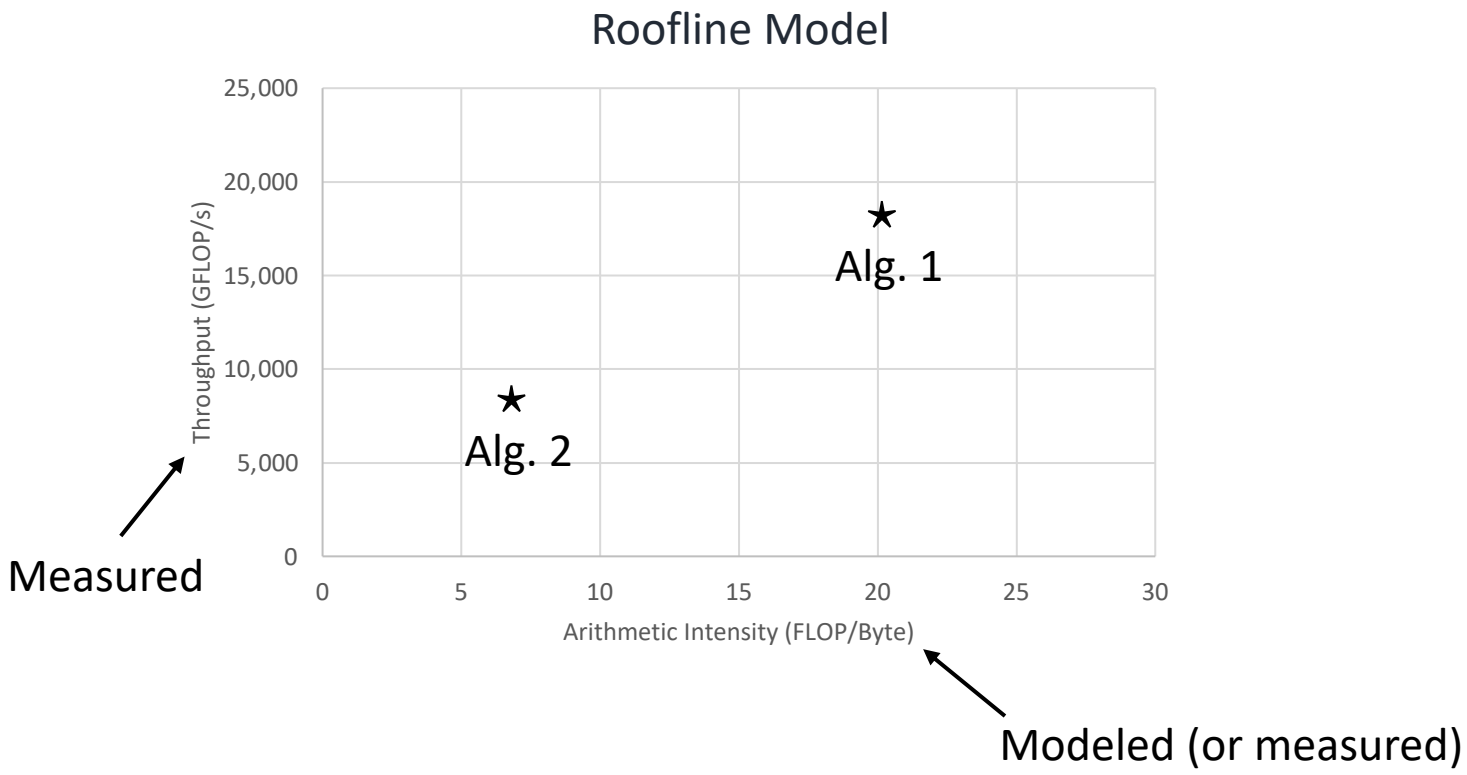
The Roofline Model



Arithmetic Intensity

$$I = \frac{\sum \text{Floating Point Operations (GFLOP)}}{\sum \text{Memory Accesses (GB)}}$$

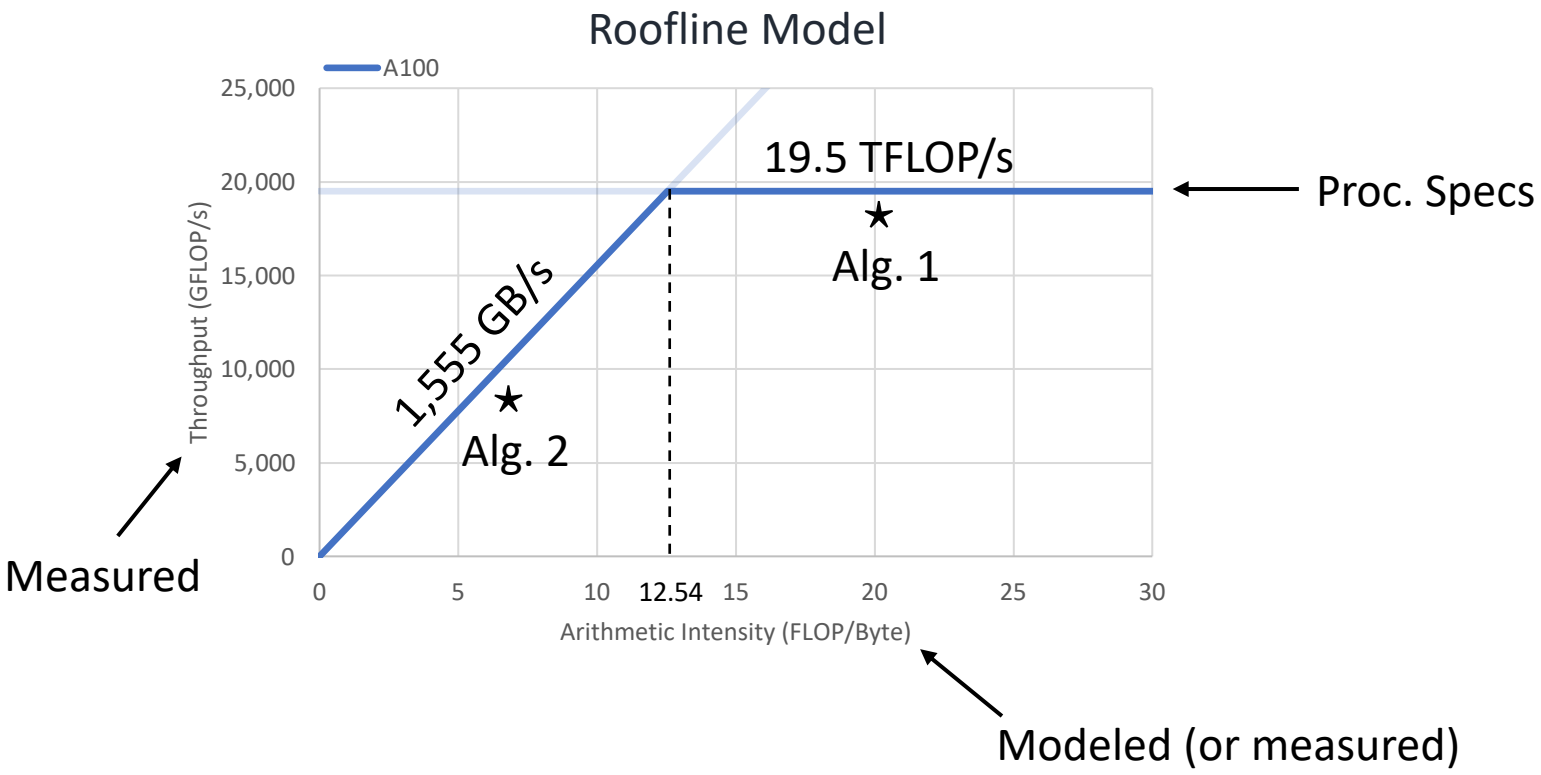
The Roofline Model



Arithmetic Intensity

$$I = \frac{\sum \text{Floating Point Operations (GFLOP)}}{\sum \text{Memory Accesses (GB)}}$$

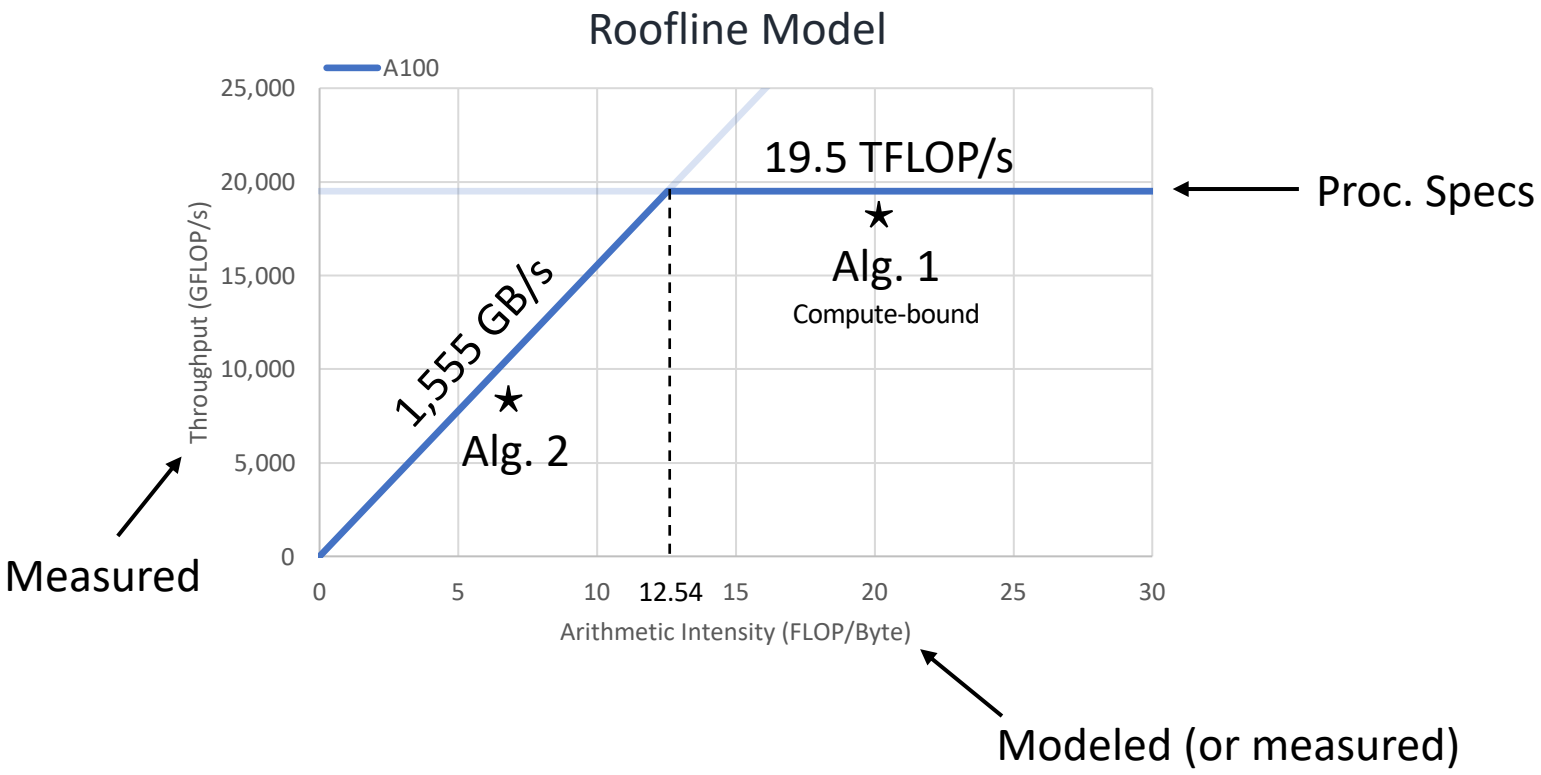
The Roofline Model



Arithmetic Intensity

$$I = \frac{\sum \text{Floating Point Operations (GFLOP)}}{\sum \text{Memory Accesses (GB)}}$$

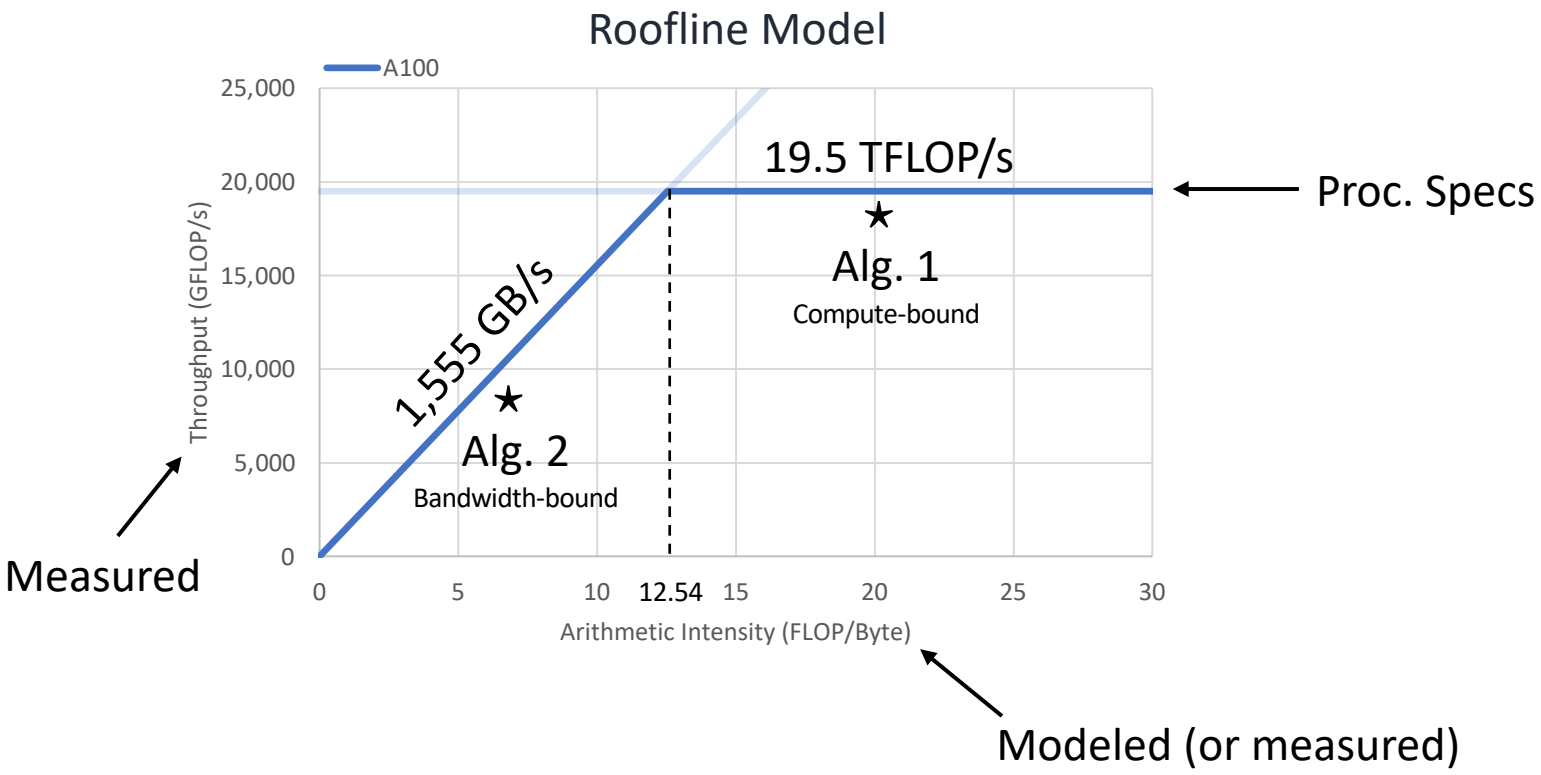
The Roofline Model



Arithmetic Intensity

$$I = \frac{\sum \text{Floating Point Operations (GFLOP)}}{\sum \text{Memory Accesses (GB)}}$$

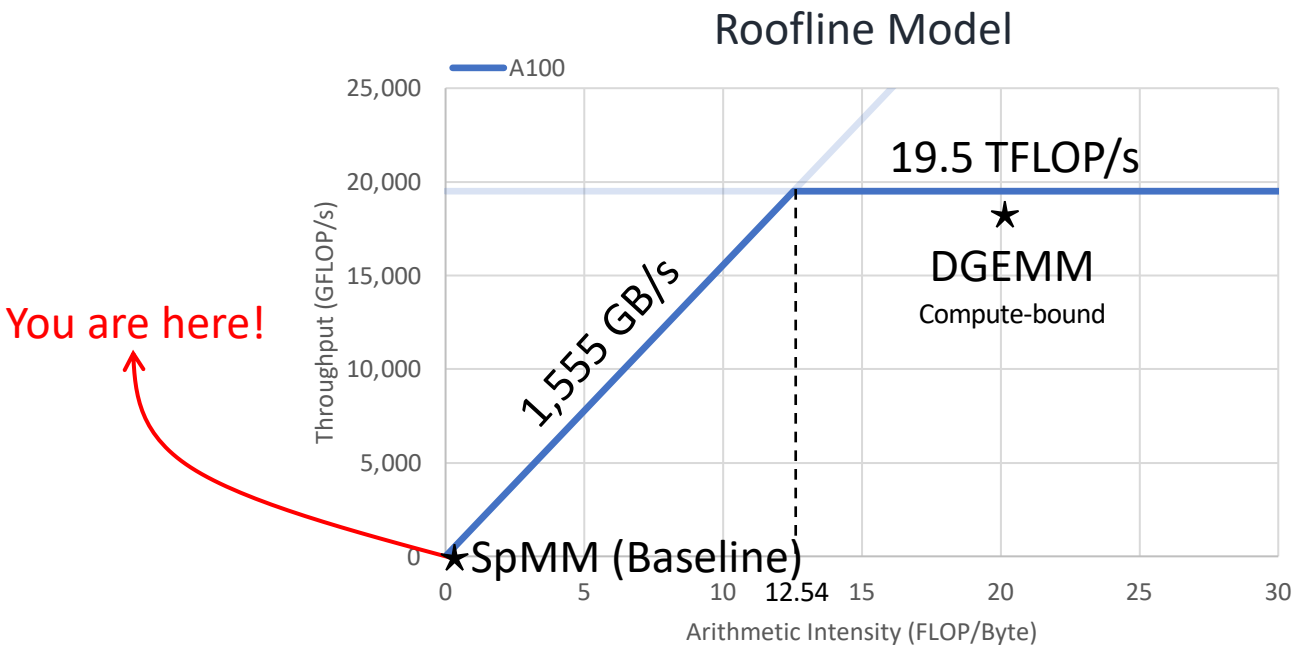
The Roofline Model



Arithmetic Intensity

$$I = \frac{\sum \text{Floating Point Operations (GFLOP)}}{\sum \text{Memory Accesses (GB)}}$$

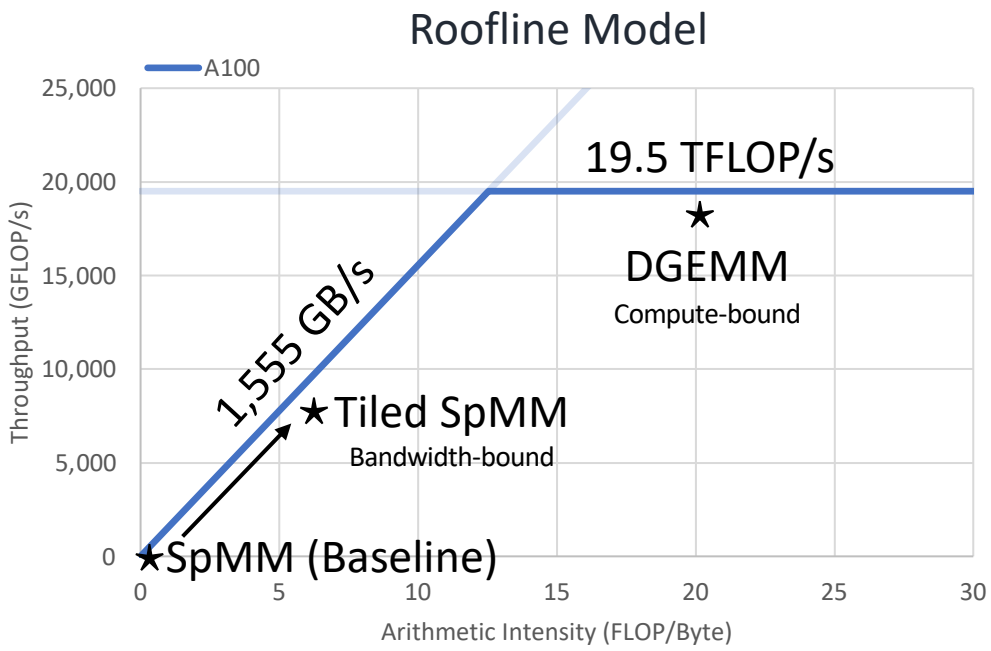
The Roofline Model



Arithmetic Intensity

$$I = 0.17 \text{ FLOP/Byte}$$

The Roofline Model



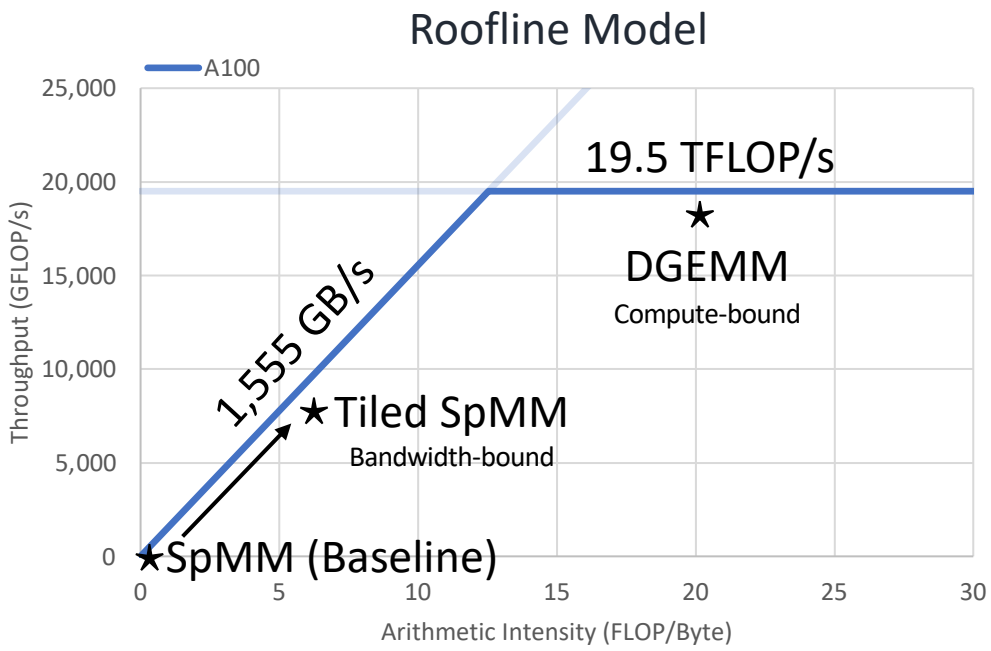
Tiled SpMM

Applies Loop Transformations

Utilizes scratchpad and registers

Improves arithmetic intensity

The Roofline Model



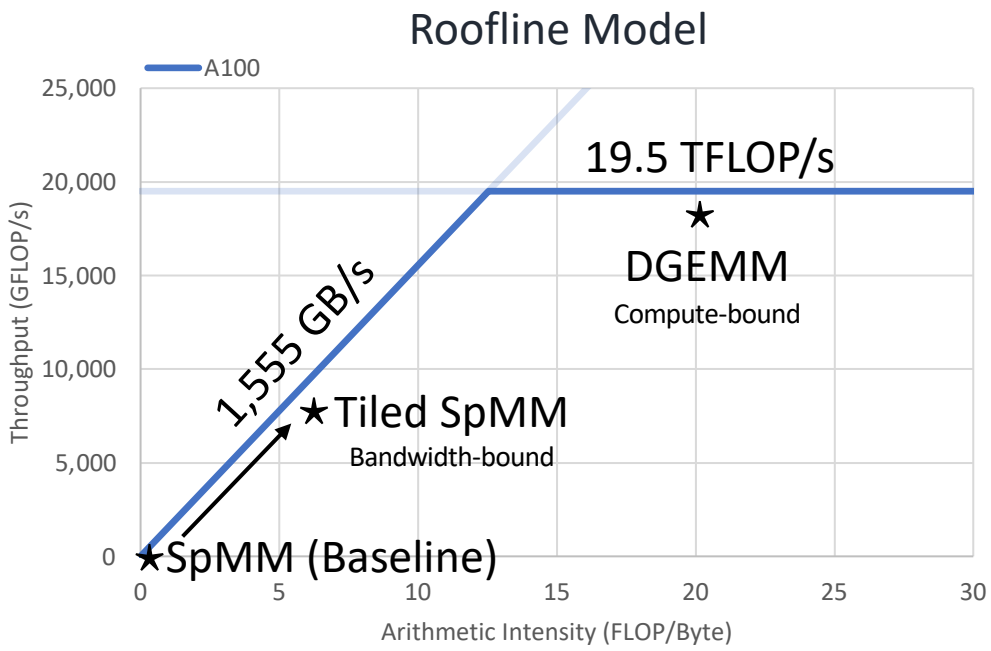
Tiled SpMM

Applies Loop Transformations { Reordering, Tiling, Fusion }

Utilizes scratchpad and registers

Improves arithmetic intensity

The Roofline Model



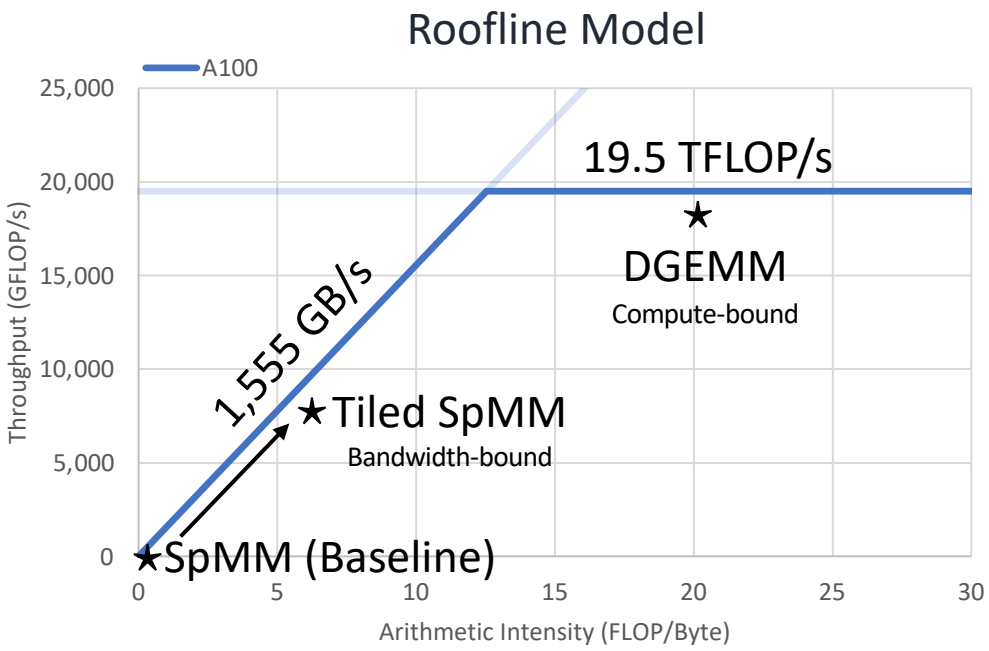
Tiled SpMM

Applies Loop Transformations

Utilizes scratchpad and registers
- Provide on-chip data reuse
- Reduce DRAM memory traffic

Improves arithmetic intensity

The Roofline Model



Tiled SpMM

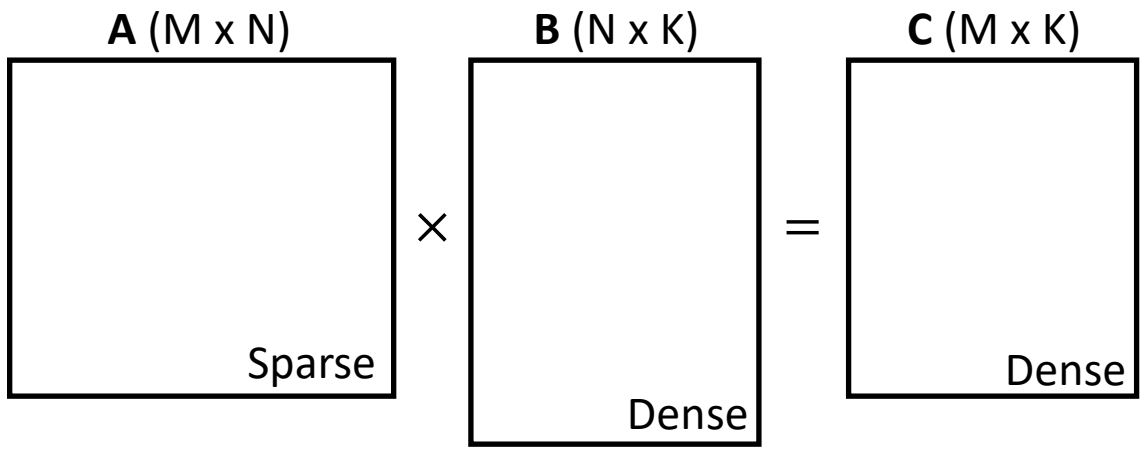
Applies Loop Transformations

Utilizes scratchpad and registers

Improves arithmetic intensity

Modeling Arithmetic Intensity of Baseline SpMM

Arithmetic Intensity: Algorithmic Model



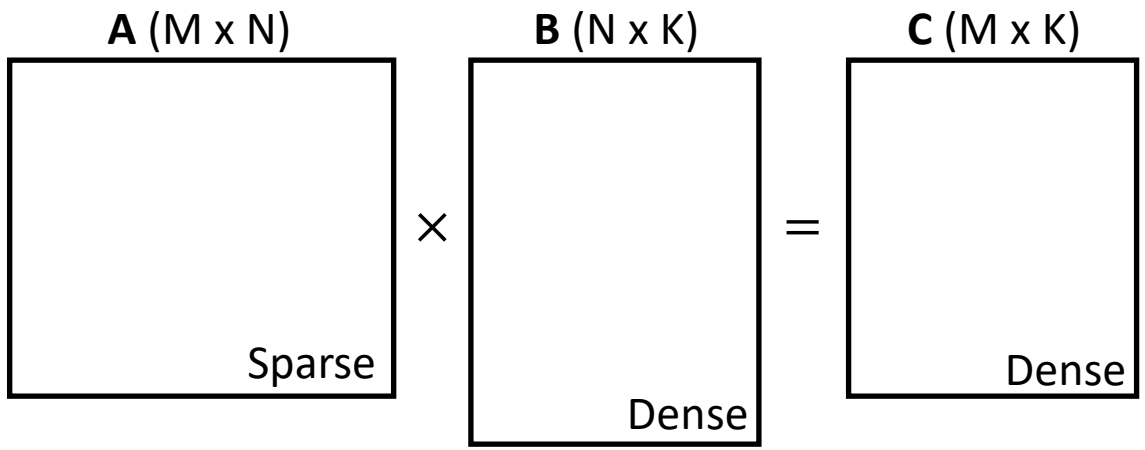
CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

For K columns

For M Rows

Arithmetic Intensity: Algorithmic Model

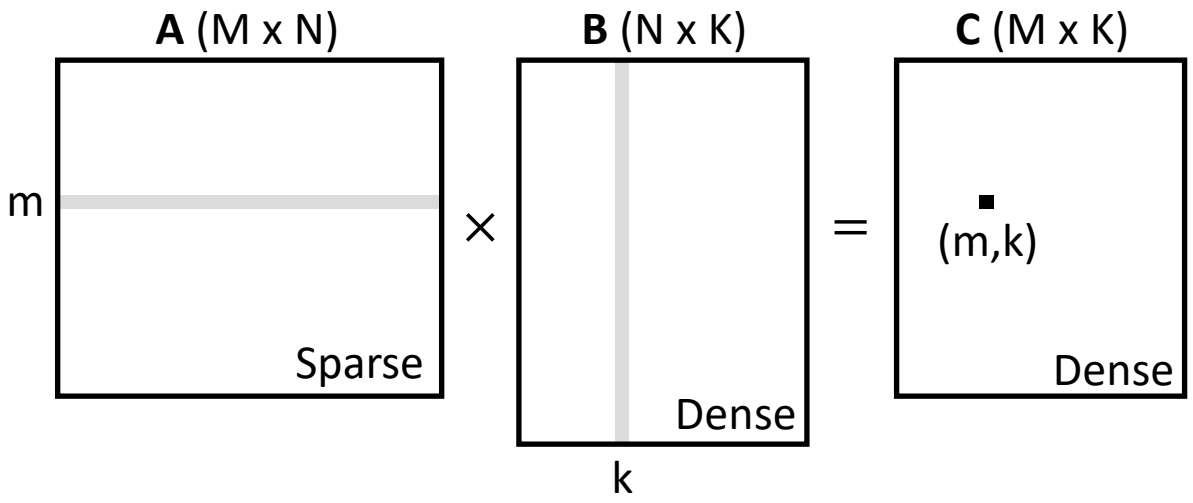


CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Arithmetic Intensity: Algorithmic Model



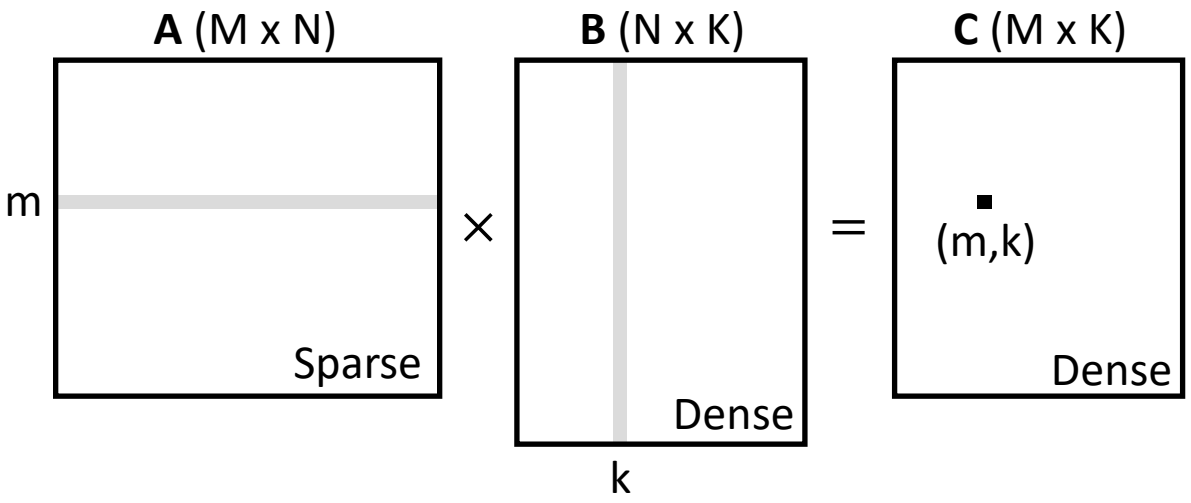
CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

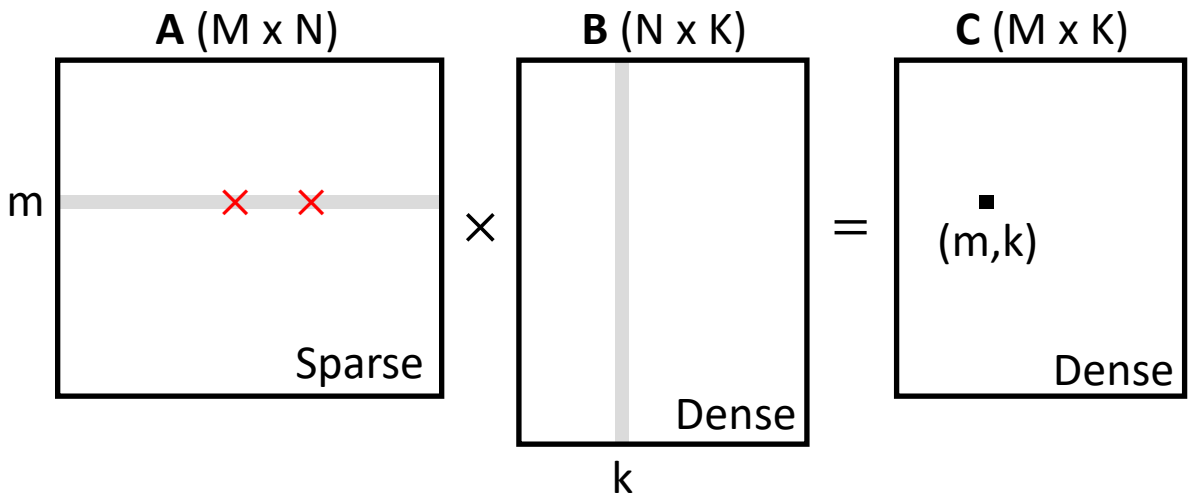
M x K Threads

Each Thread

$$I = \frac{2\mu}{(B_T + B_I)\mu + B_T\mu + B_T + \delta}$$

Algorithmic Intensity

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

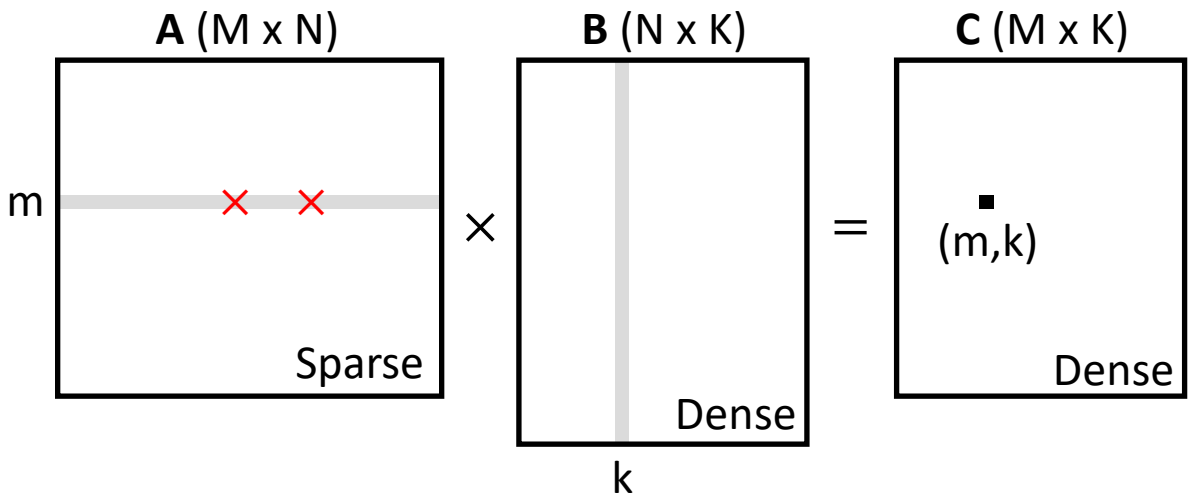
Legend

μ: # of nonzeroes per row

$$I = \frac{2\mu}{(B_T + B_I)\mu + B_T\mu + B_T + \delta}$$

Algorithmic Intensity

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

Fused Multiply-Add (FMA): 2 FLOPs

FLOPs per nnz

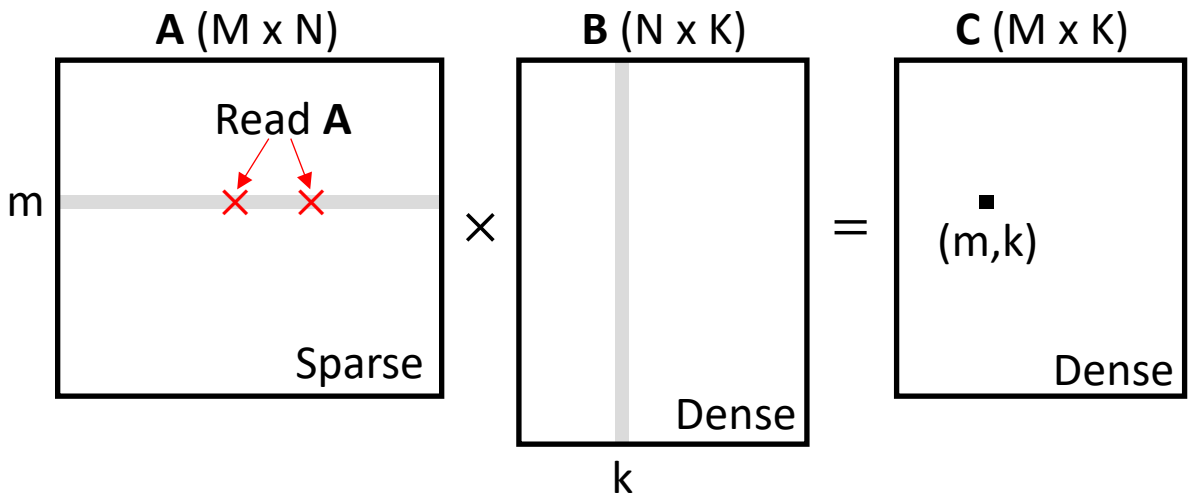
$$I = \frac{2\mu}{(B_T + B_I)\mu + B_T\mu + B_T + \delta}$$

Algorithmic Intensity

Legend

μ: # of nonzeroes per row

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

Read **A**

Fused Multiply-Add (FMA): 2 FLOPs

$$I = \frac{\overbrace{2\mu}^{\text{FLOPs per nnz}}}{\underbrace{(B_T + B_I)\mu}_{\text{Read } \mathbf{A}} + B_T\mu + B_T + \delta}$$

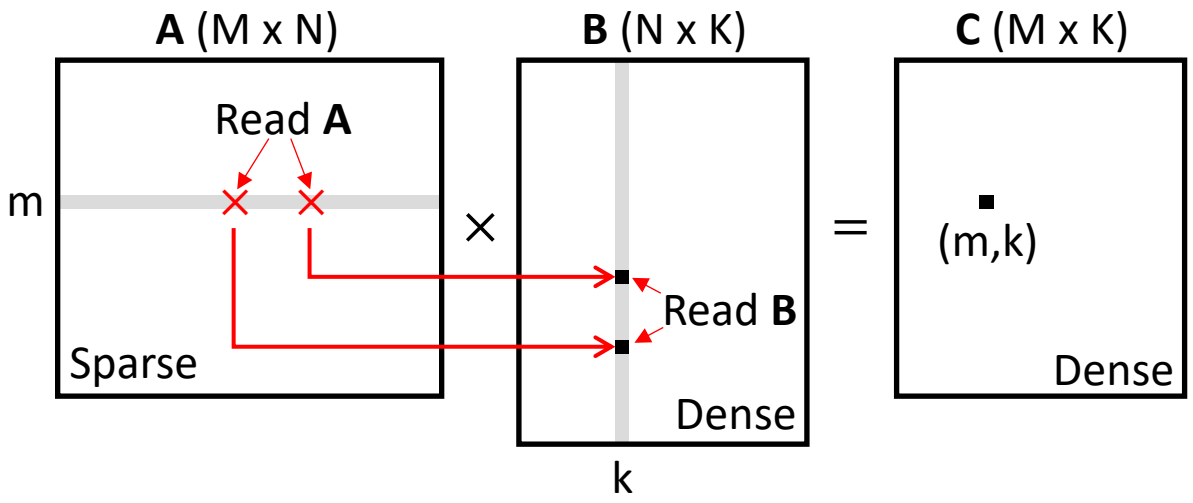
Legend

μ: # of nonzeroes per row

B_T: Bytes per value

B_I: Bytes per index

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

Read **A**

Read **B**

Fused Multiply-Add (FMA): 2 FLOPs

$$I = \frac{\overbrace{2\mu}^{\text{FLOPs per nnz}}}{\underbrace{(B_T + B_I)\mu}_{\text{Read A}} + \underbrace{B_T\mu}_{\text{Read B}} + B_T + \delta}$$

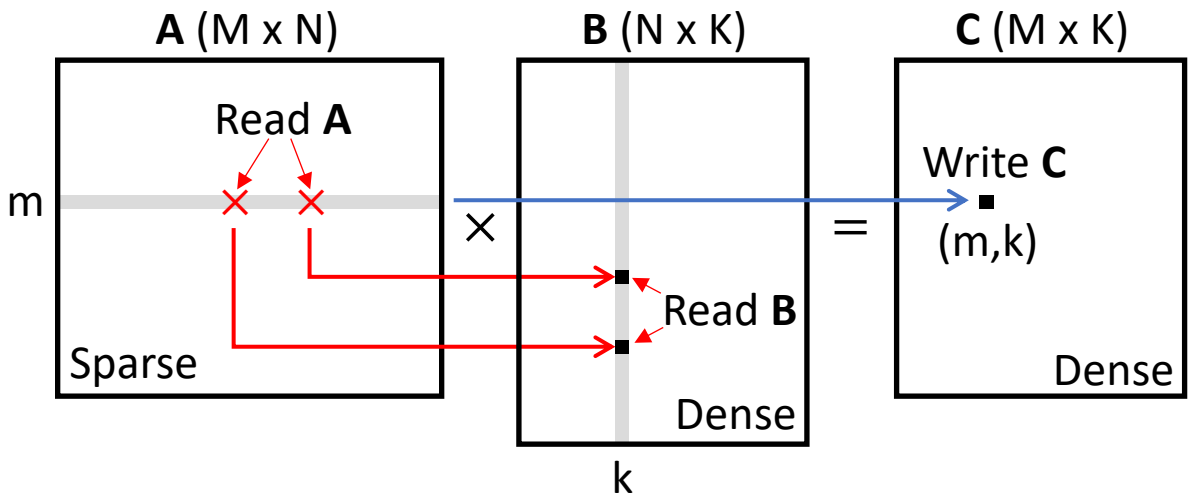
Legend

μ: # of nonzeroes per row

B_T: Bytes per value

B_I: Bytes per index

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

Read **A**

Read **B**

Fused Multiply-Add (FMA): 2 FLOPs

Write **C**

$$I = \frac{2\mu}{\underbrace{(B_T + B_I)\mu}_{\text{Read } \mathbf{A}} + \underbrace{B_T\mu}_{\text{Read } \mathbf{B}} + \underbrace{B_T}_{\text{Write } \mathbf{C}} + \delta}$$

FLOPs per nnz

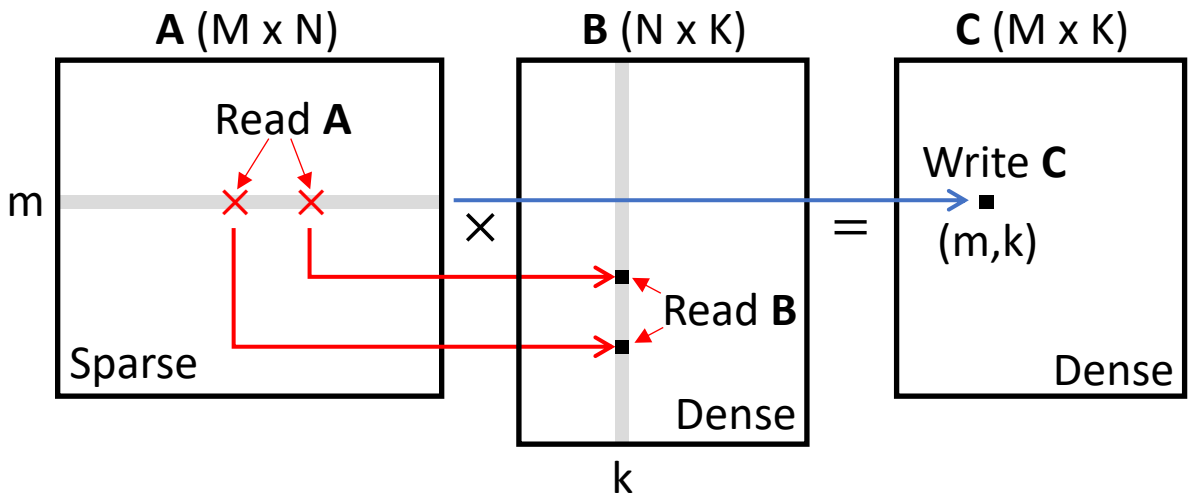
Legend

μ: # of nonzeroes per row

B_T: Bytes per value

B_I: Bytes per index

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

Metadata

μ times

Read A

Read B

Fused Multiply-Add (FMA): 2 FLOPs

Write C

$$I = \frac{2\mu}{\underbrace{(B_T + B_I)\mu}_{\text{Read A}} + \underbrace{B_T\mu}_{\text{Read B}} + \underbrace{B_T}_{\text{Write C}} + \delta}$$

FLOPs per nnz

Read Metadata

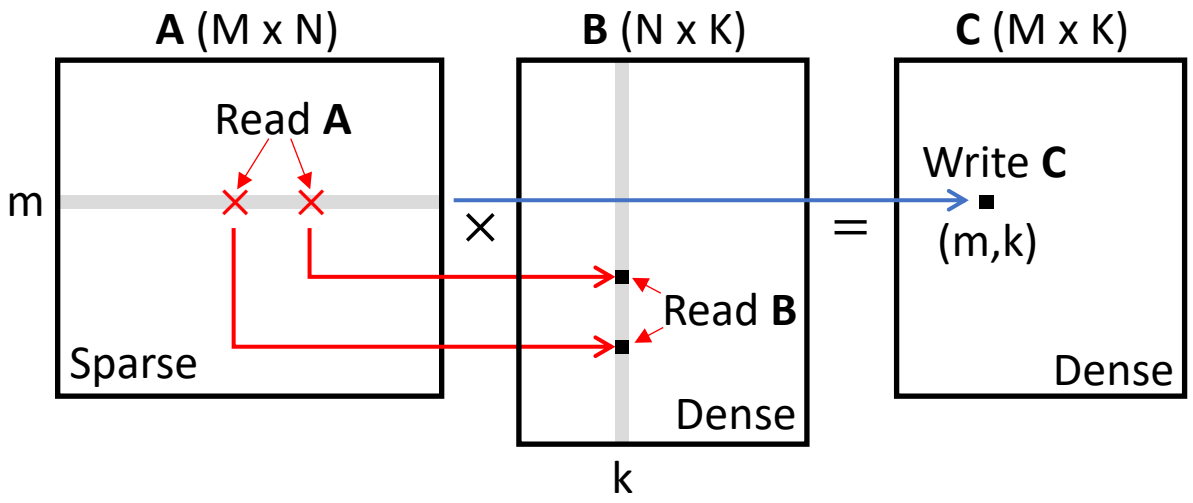
Legend

μ: # of nonzeroes per row

B_T: Bytes per value

B_I: Bytes per index

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

Metadata

μ times

Read A

Read B

Fused Multiply-Add (FMA): 2 FLOPs

Write C

$$I = \frac{2\mu}{\underbrace{(B_T + B_I)\mu}_{\text{Read A}} + \underbrace{B_T\mu}_{\text{Read B}} + \underbrace{B_T}_{\text{Write C}} + \delta}$$

FLOPs per nnz

Read Metadata

$\frac{\text{FLOPs}}{\text{byte}}$

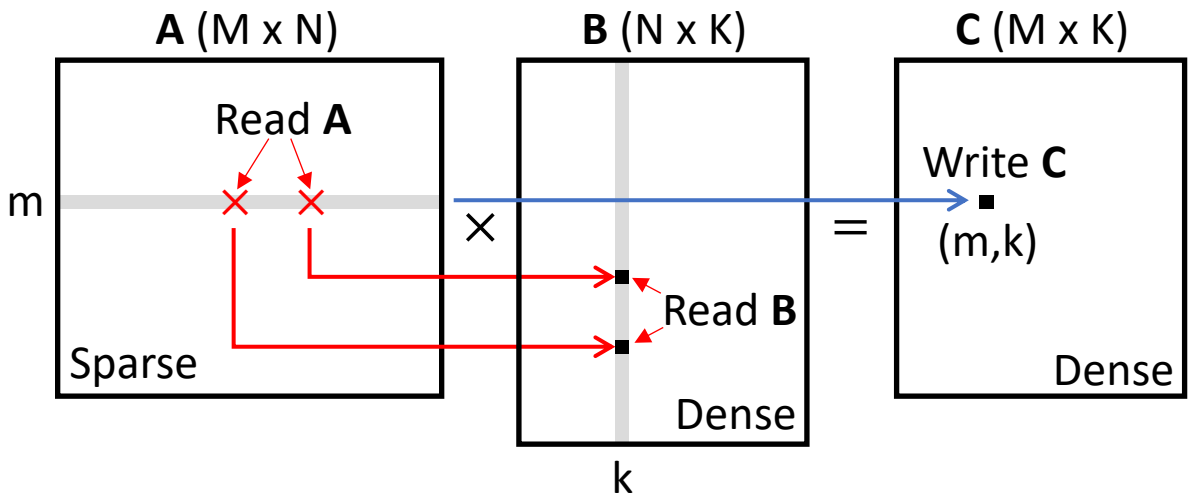
Legend

μ: # of nonzeroes per row

B_T: Bytes per value

B_I: Bytes per index

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

Metadata

μ times

Read A

Read B

Fused Multiply-Add (FMA): 2 FLOPs

Write C

$$I\beta = \frac{2\beta\mu}{(B_T + B_I)\mu + B_T\mu + B_T + \delta}$$

FLOPs per nnz

1,555 GB/s

GFLOPs / second

Read A

Read B

Write C

Read Metadata

Legend

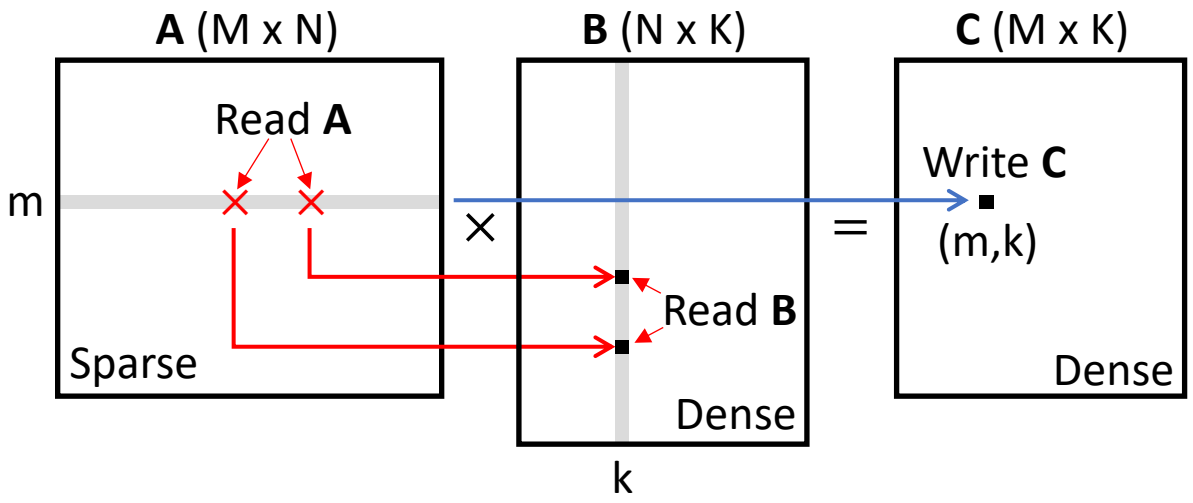
μ: # of nonzeroes per row

B_T: Bytes per value

B_I: Bytes per index

β: Memory B/W

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

Read **A**

Read **B**

Write **C**

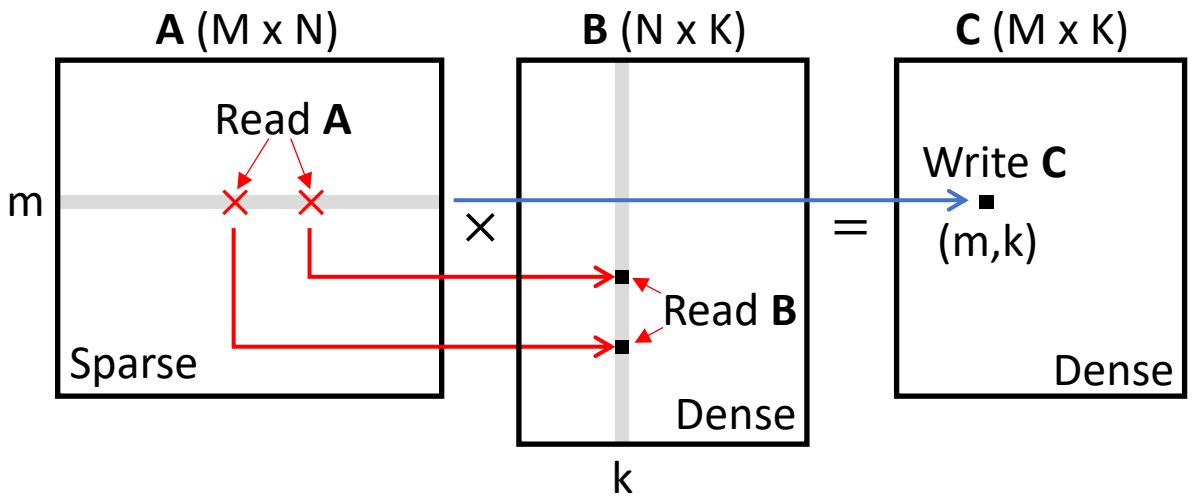
$$\underbrace{130 \text{ GFLOPS}}_{\substack{\mu = 1 \\ \text{more sparsity} \\ \text{e.g, diagonal matrix}}} < I\beta = \frac{2\beta\mu}{(B_T + B_I)\mu + B_T\mu + B_T + \delta} < \underbrace{260 \text{ GFLOPS}}_{\substack{\mu \gg B_T \\ \text{less sparsity}}}$$

1,555 GB/s

Four bytes

Eight bytes

Arithmetic Intensity: Algorithmic Model



CSR SpMM (Baseline)

```
for (int k = 0; k < K; k++)
  for (int m = 0; m < M; m++)
  {
    float acc = 0;
    for (int l = rowPtr[m]; l < rowPtr[m + 1]; l++)
    {
      int idx = index[l];
      float val = value[l];
      acc += B[idx][k] * val;
    }
    C[m][k] = acc;
  }
```

M x K Threads

Each Thread

μ times

Read **A**

Read **B**

Write **C**

$$\underbrace{130\text{ GFLOPS}}_{\substack{\mu = 1 \\ \text{more sparsity} \\ \text{e.g, diagonal matrix}}} < I\beta = \frac{2\beta\mu}{(B_T + B_I)\mu + B_T\mu + B_T + \delta} < \underbrace{260\text{ GFLOPS}}_{\substack{\mu \gg B_T \\ \text{less sparsity}}}$$

1,555 GB/s

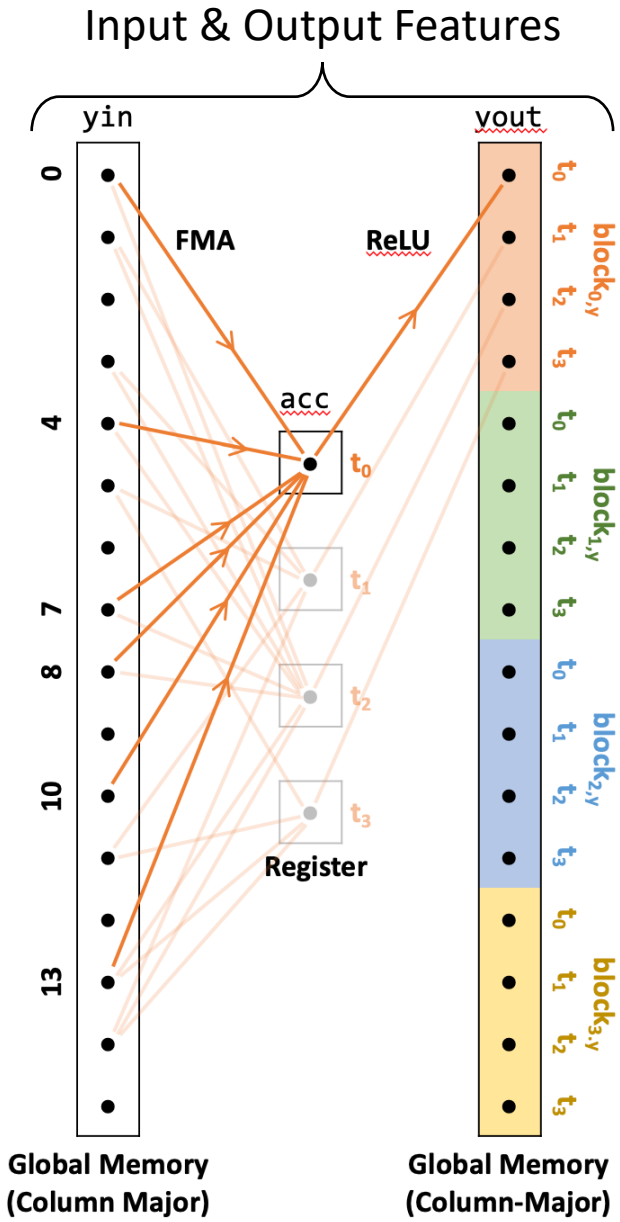
Four bytes

Eight bytes

1.3% of the peak compute throughput!

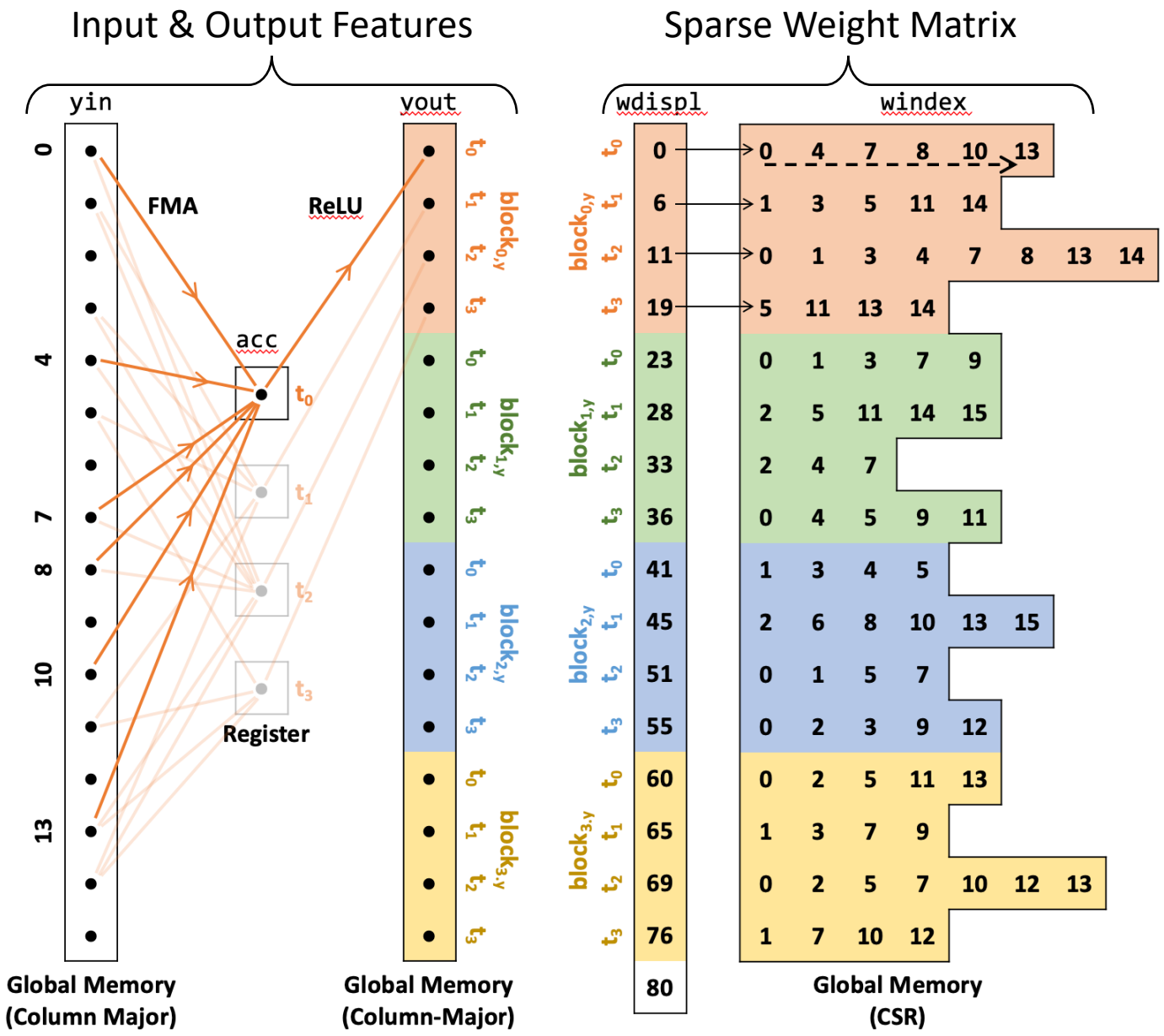
Baseline Sparse Layer Implementation

- Activation features: dense (col.-major)
- Gather approach
- Each thread computes a single output



Baseline Sparse Layer Implementation

- Activation features: dense (col.-major)
- Gather approach
- Each thread computes a single output
- Sparse weights: CSR
- **ReLU is fused with SpMM**

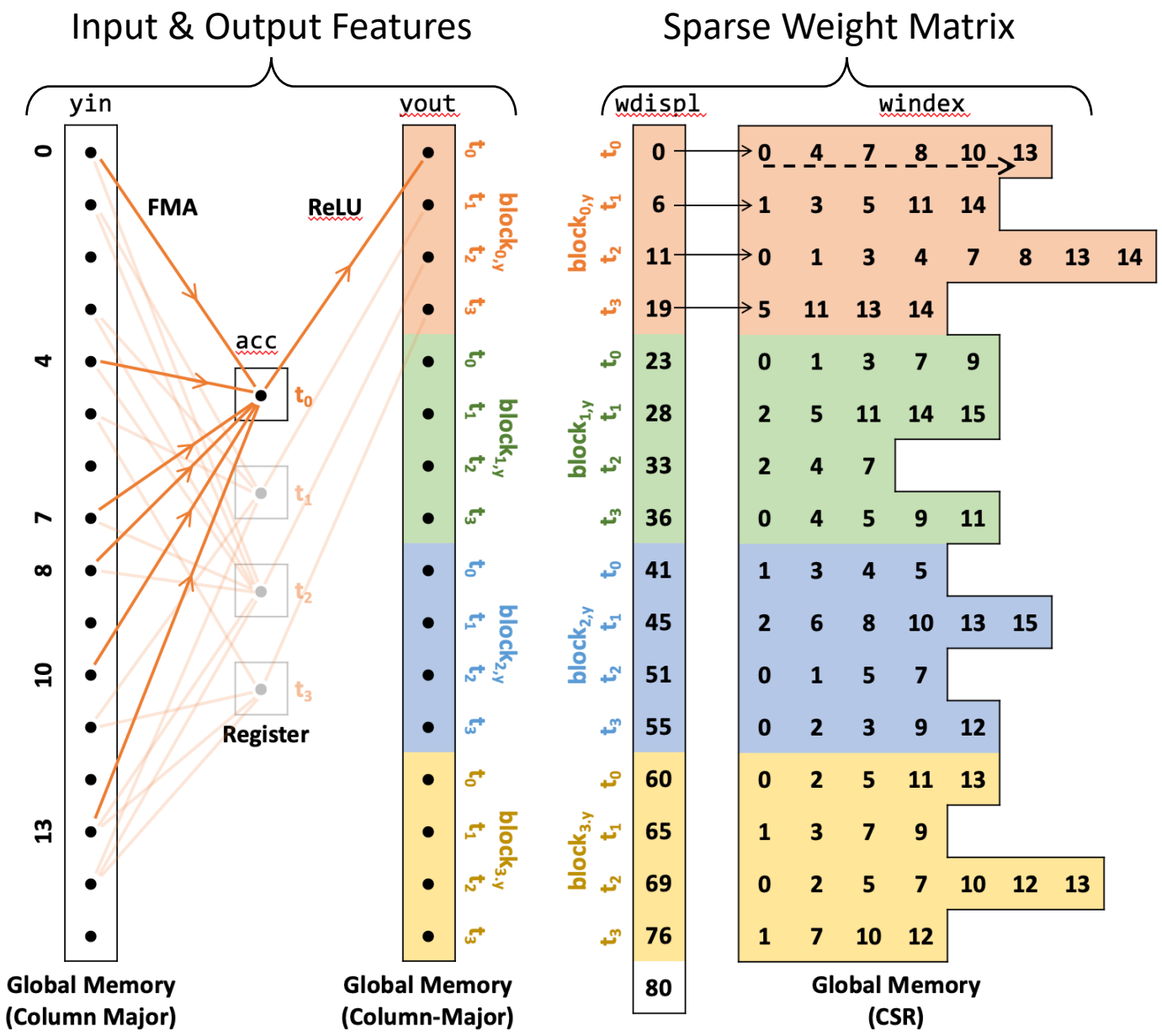


Baseline Sparse Layer Implementation

- Activation features: dense (col.-major)
- Gather approach
- Each thread computes a single output
- Sparse weights: CSR
- **ReLU is fused with SpMM**

Data Access Redundancies

- Weight matrix by all output features
- Input features by different threads



Baseline Sparse Layer Implementation

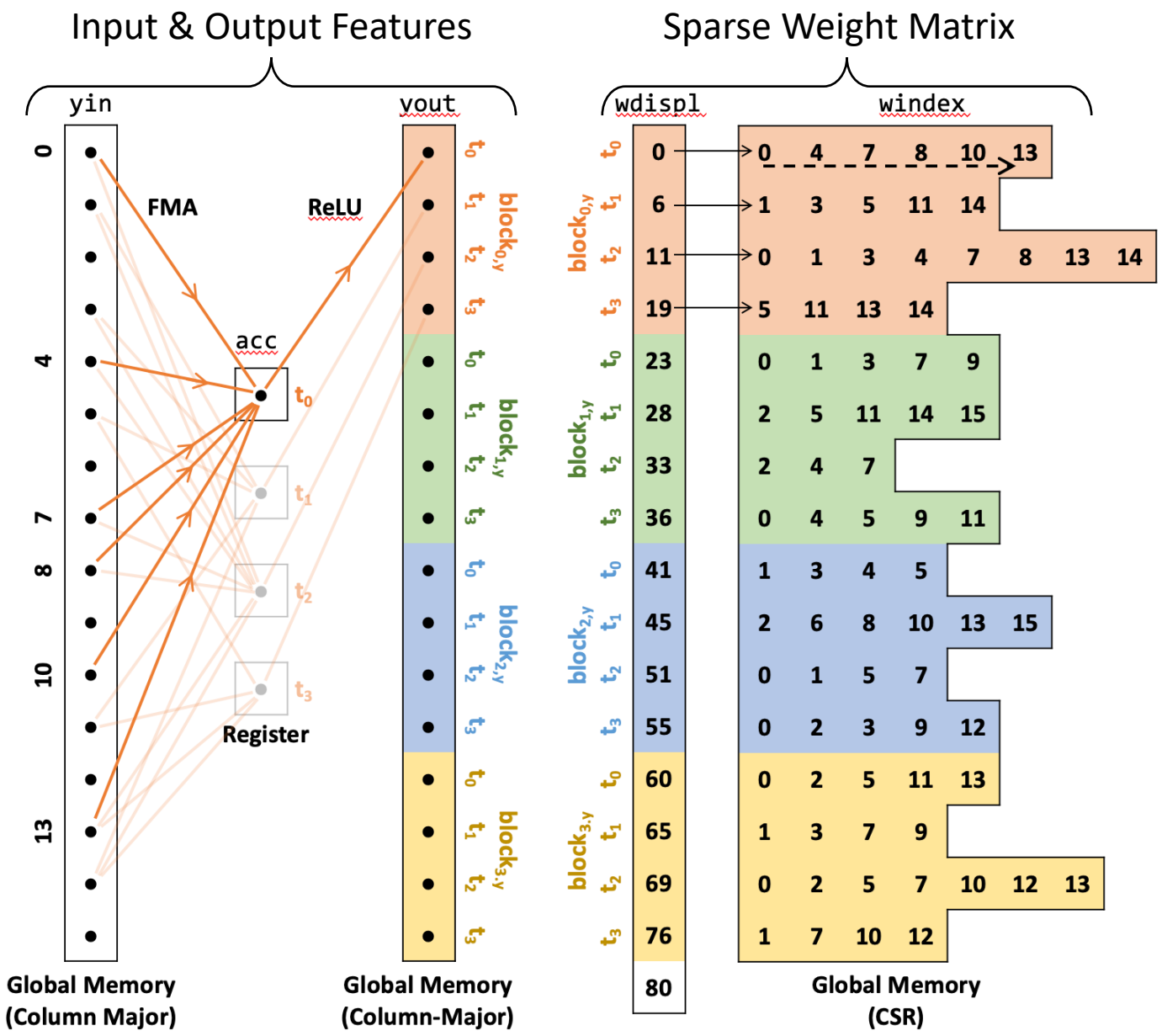
- Activation features: dense (col.-major)
- Gather approach
- Each thread computes a single output
- Sparse weights: CSR
- **ReLU is fused with SpMM**

Data Access Redundancies

- Weight matrix by all output features
- Input features by different threads

Data Access Latencies

- Irregular access to input features
- Uncoalesced access to weight matrix



SpMM Optimization for ML

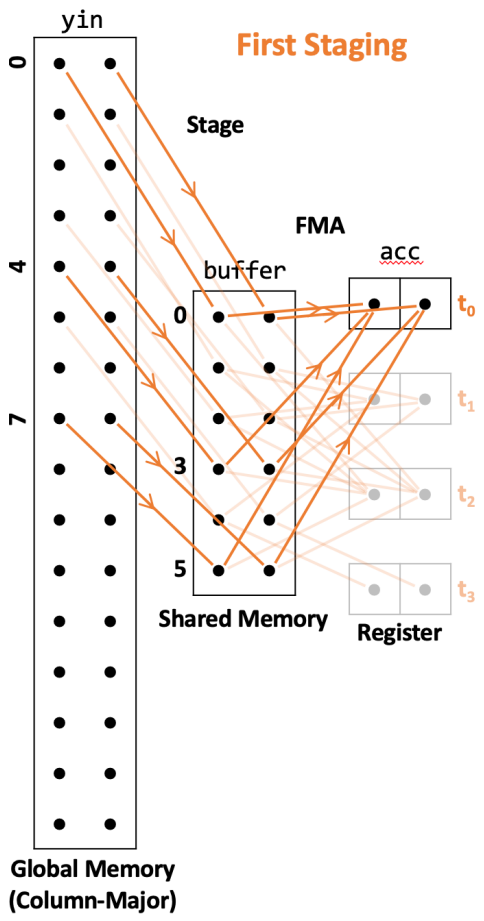
Register Tiling

Scratchpad Tiling

Fusing Activation

Tiled SpMM for Sparse Deep Neural Networks

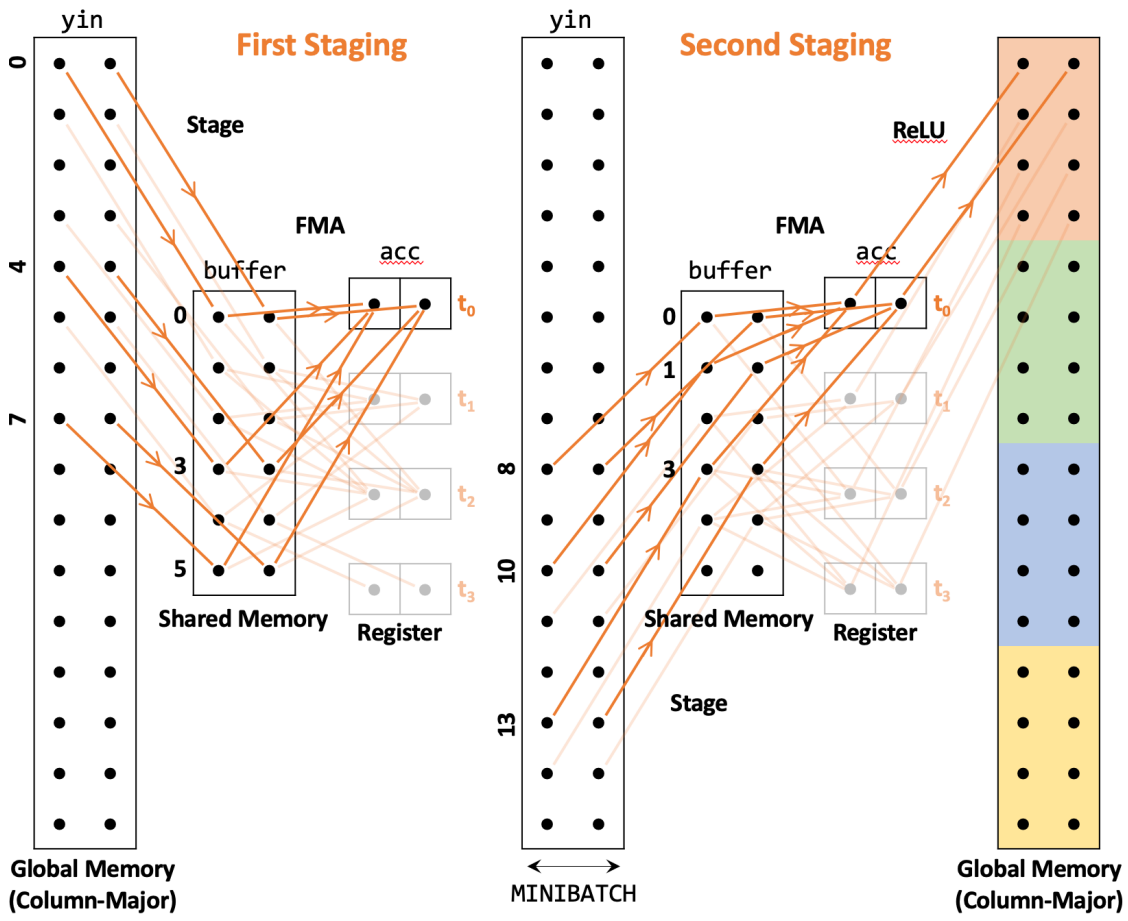
Multi-Level Input Buffering



Hidayetoglu, et al., “At-Scale Deep Neural Network Inference with Efficient GPU Implementation,” Arxiv and HPEC’20.

Tiled SpMM for Sparse Deep Neural Networks

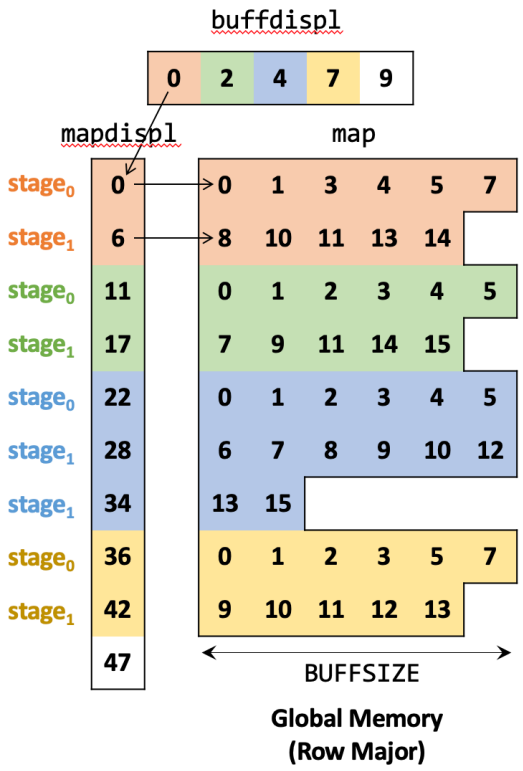
Multi-Level Input Buffering



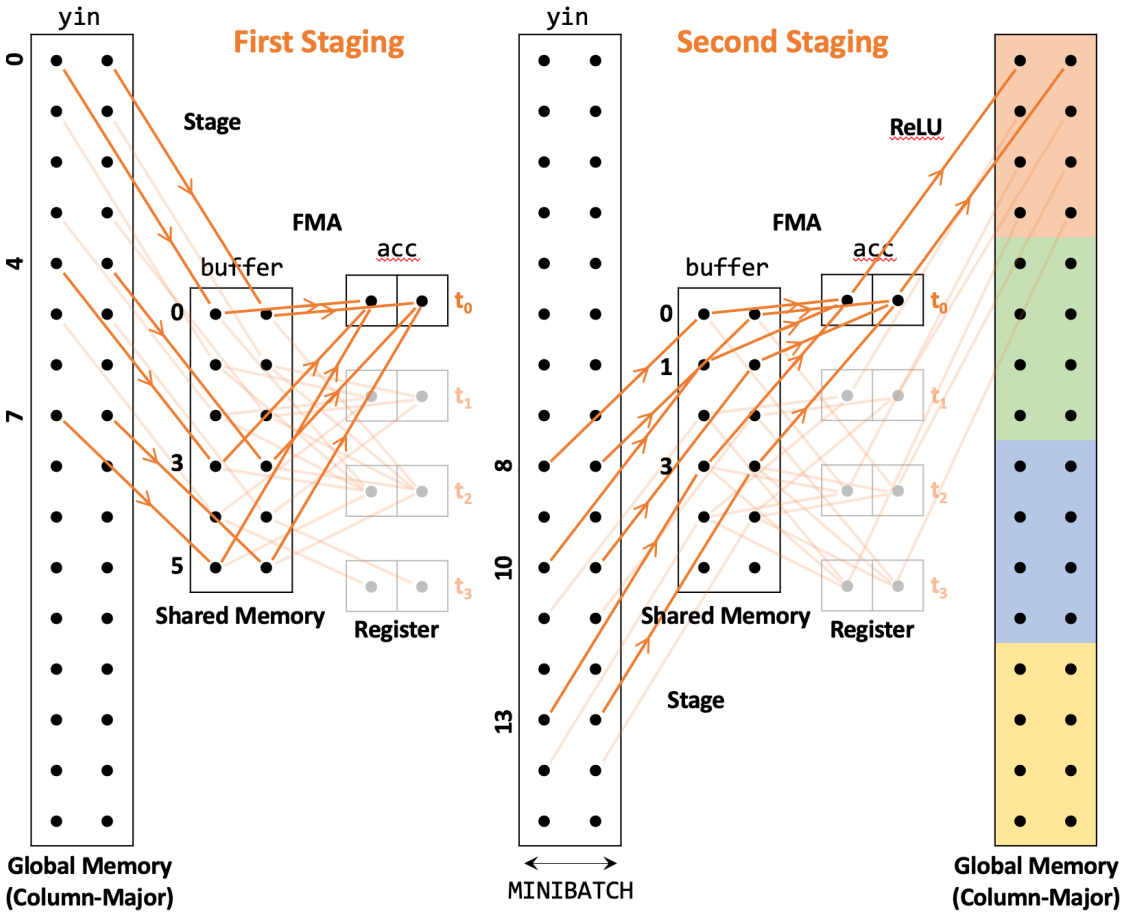
Hidayetoglu, et al., “At-Scale Deep Neural Network Inference with Efficient GPU Implementation,” Arxiv and HPEC’20.

Tiled SpMM for Sparse Deep Neural Networks

Intermediate Data Structures



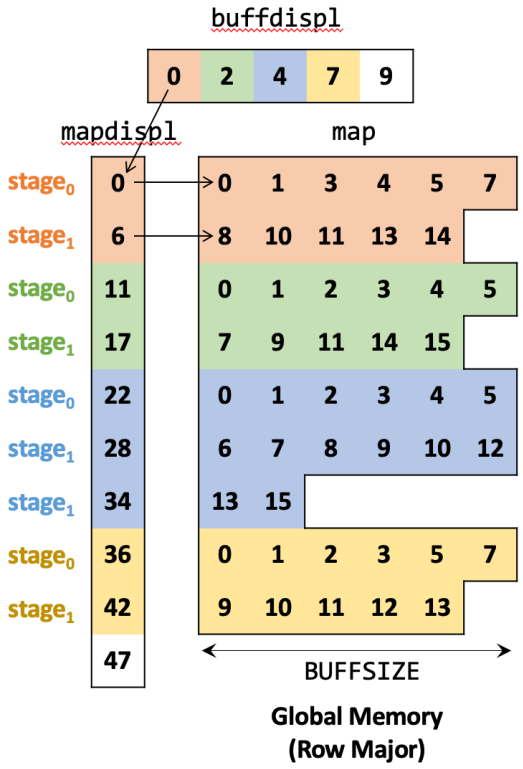
Multi-Level Input Buffering



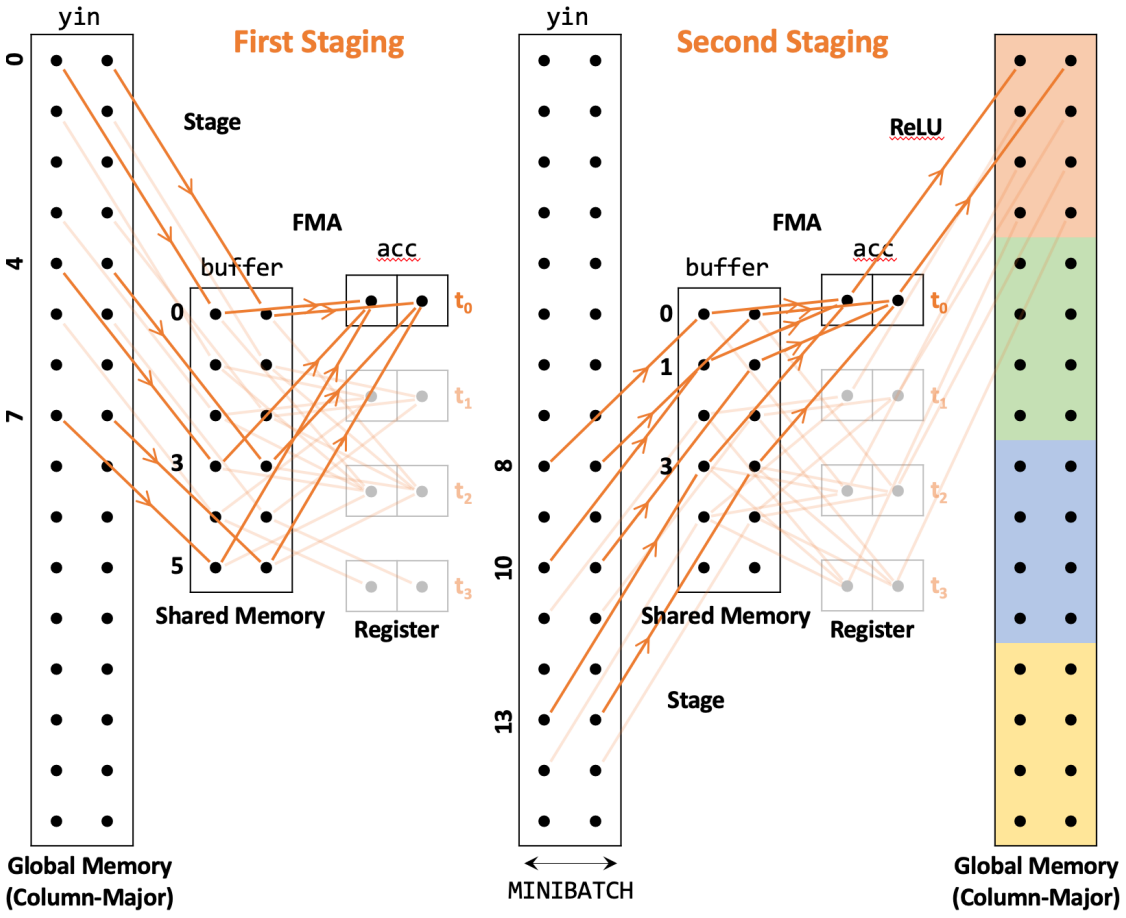
Hidayetoglu, et al., “At-Scale Deep Neural Network Inference with Efficient GPU Implementation,” Arxiv and HPEC’20.

Tiled SpMM for Sparse Deep Neural Networks

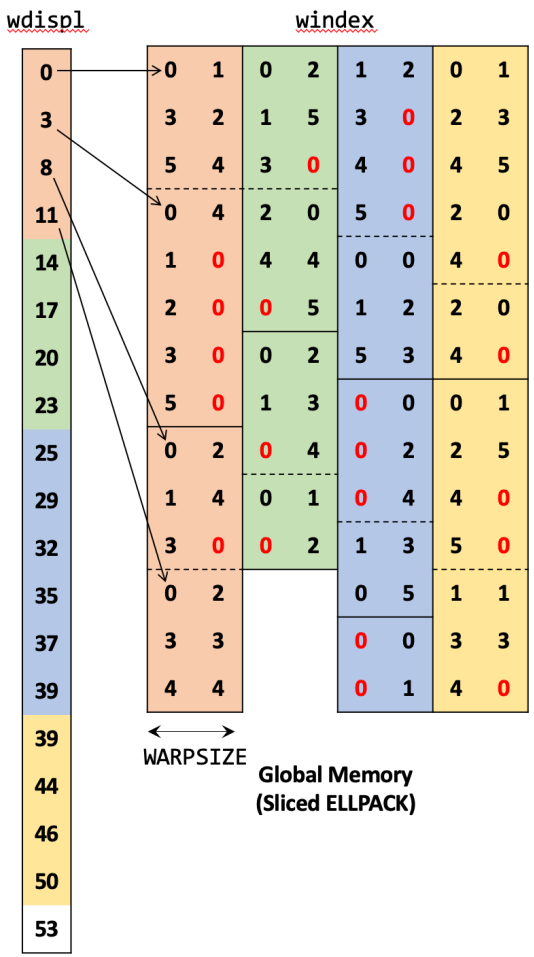
Intermediate Data Structures



Multi-Level Input Buffering



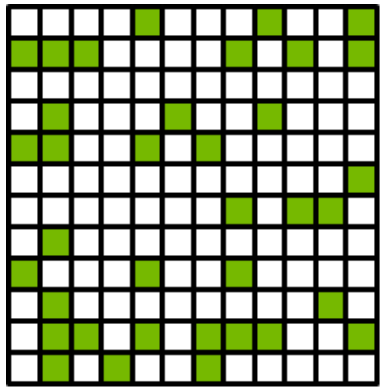
Weights Sliced ELLPACK

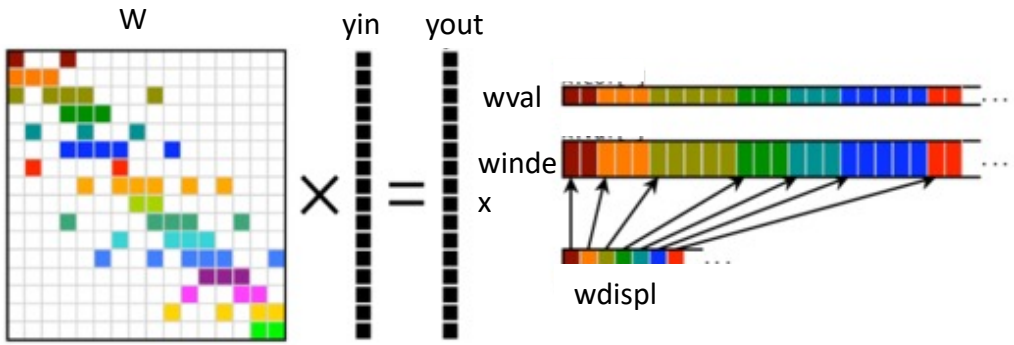


Hidayetoglu, et al., “At-Scale Deep Neural Network Inference with Efficient GPU Implementation,” Arxiv and HPEC’20.

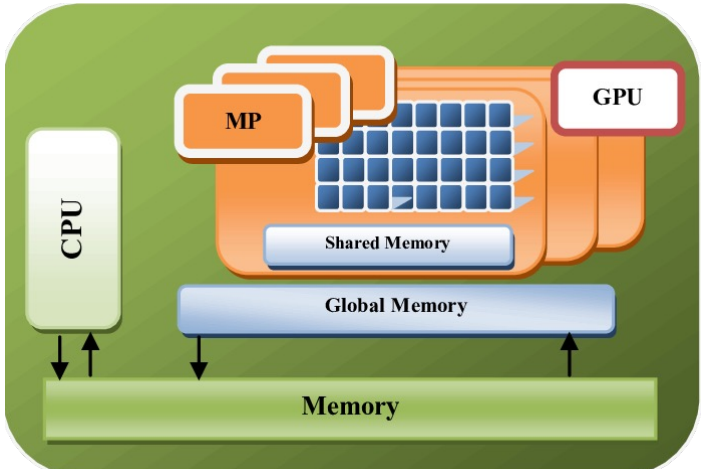
Tiled SpMM acceleration on GPUs – talk by Mert @ 1:50 pm

1. Optimize SpMM data structures and algorithms for generic hardware features

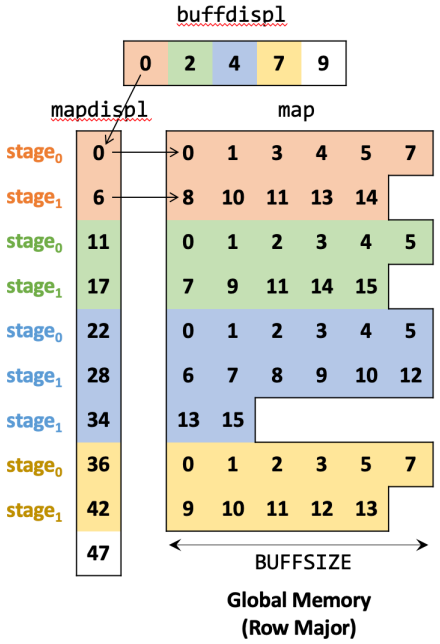




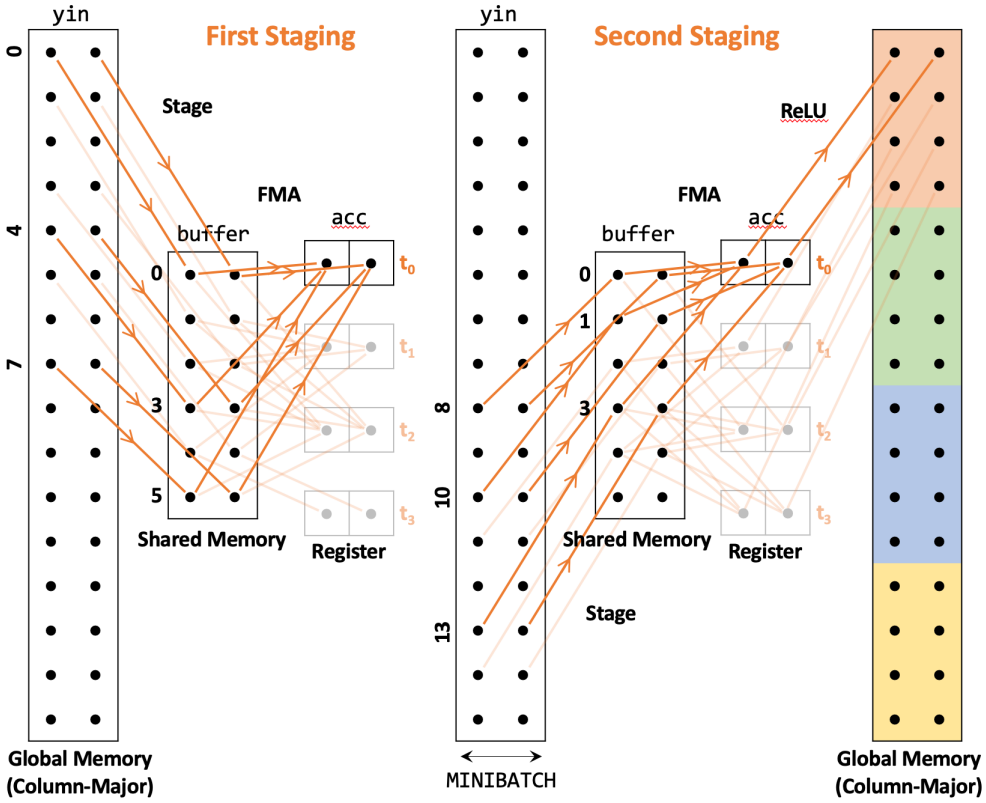
2. Co-optimize SpMM implementation for underlying hardware features



Intermediate Data Structures



Multi-Level Input Buffering

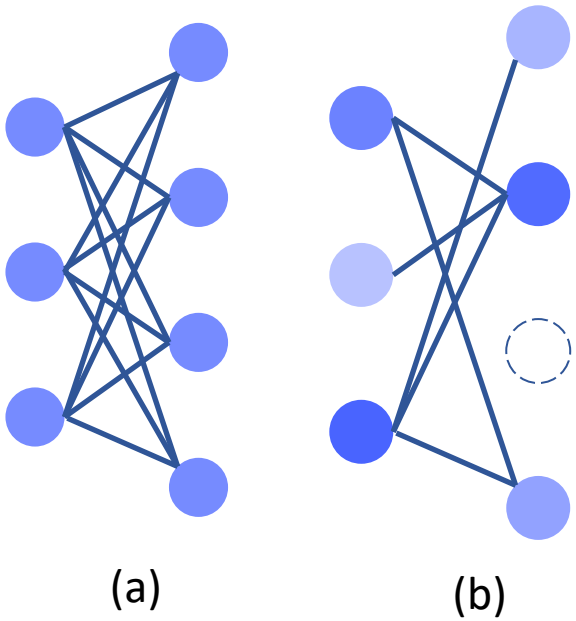


[Hidayetoglu et al: "At-Scale Sparse Deep Neural Network Inference With Efficient GPU Implementation," IEEE GraphChallenge 2020 (Champion)]

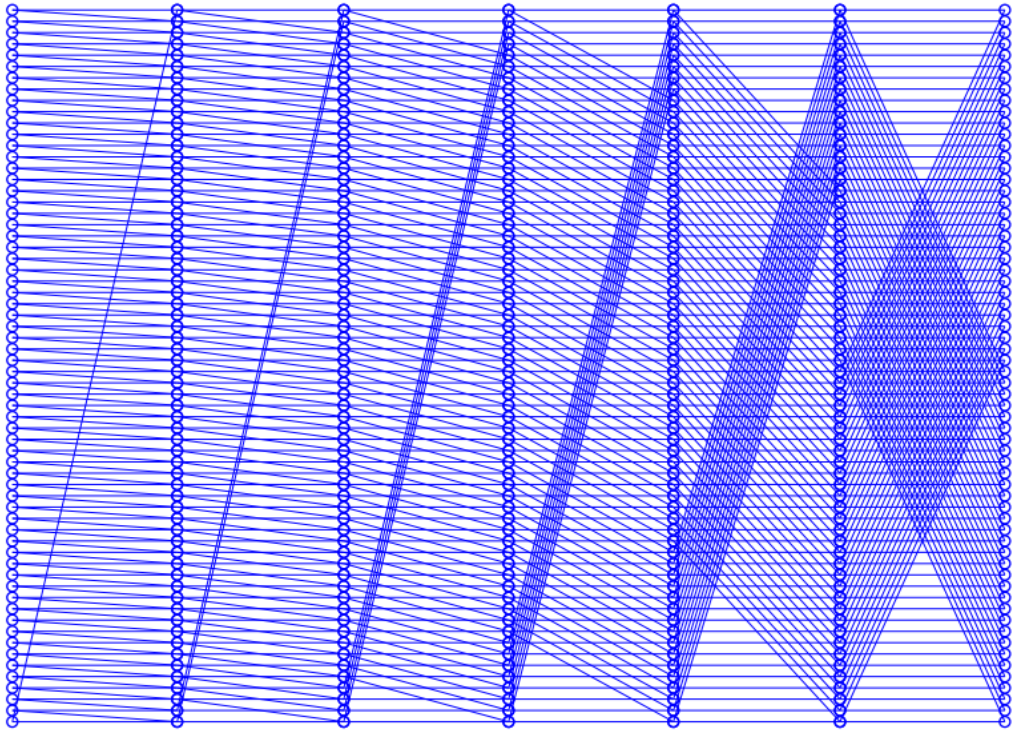
Tiled SpMM for Sparse Deep Neural Networks

MIT/Amazon/IEEE Graph Challenge

grapchallenge.mit.edu



(a) Fully connected layer;
(b) Sparse layer (after pruning)



A sparse DNN with 6 layers, 64 neurons per layer (synthetic, 2 connections per neuron) [1]

[1] J. Kepner et al. Sparse Deep Neural Network Graph Challenge. HPEC 2019.

Tiled SpMM for Sparse Deep Neural Networks

MIT/Amazon/IEEE Graph Challenge

grapchallenge.mit.edu

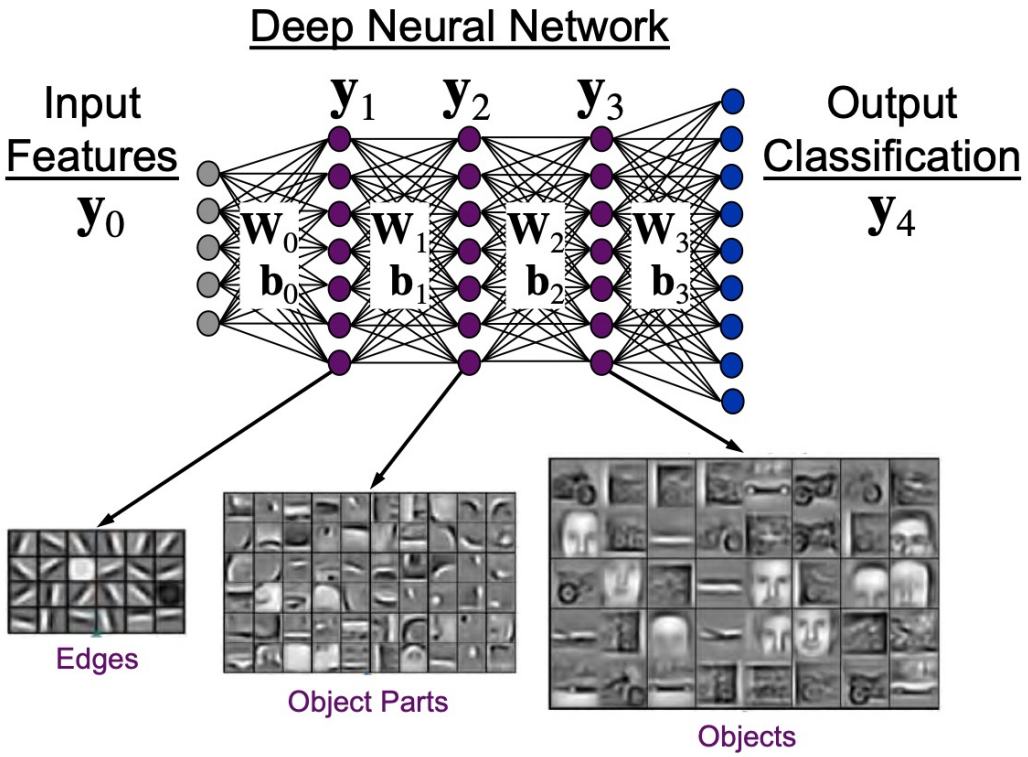
Challenges

- Irregular: Frequent random accesses to memory
- Various Scales: from 120 layers, 1024 neurons/layer to 1920 layers 65536 neurons/layer (16.3 GB)

Computation

- Input and parameters are in graph format (sparse)
- CPU performs data preprocessing
- GPU performs DNN inference: ℓ-th layer:

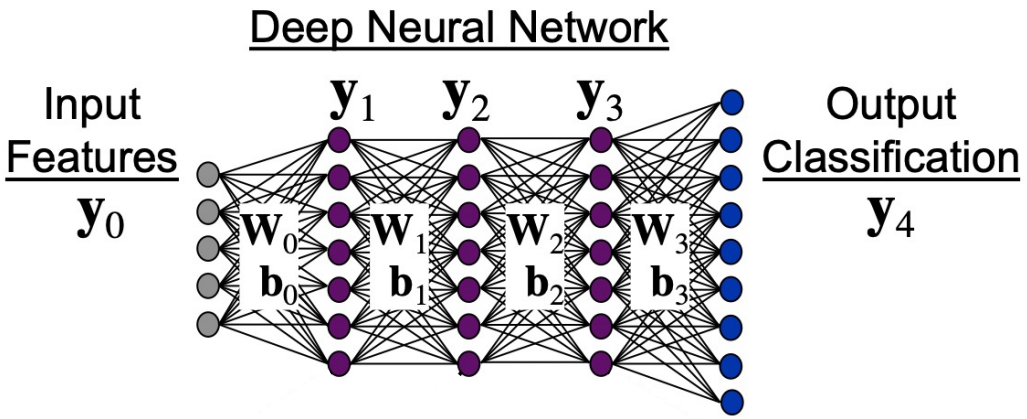
$$\boldsymbol{Y}_{\ell+1} = h(\boldsymbol{W}_\ell \boldsymbol{Y}_\ell + \boldsymbol{B}_\ell)$$



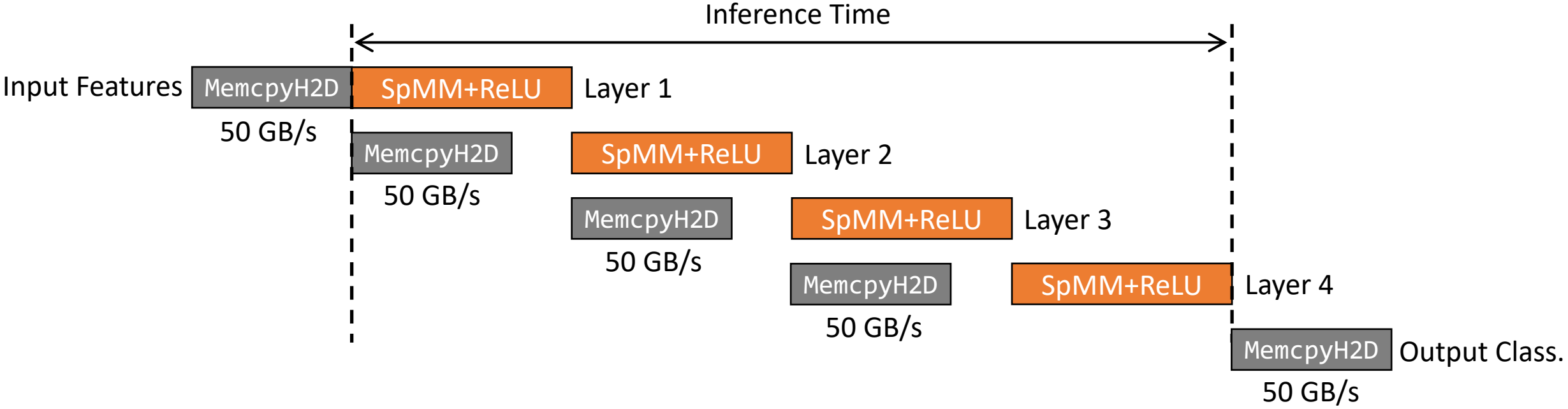
Tiled SpMM for Sparse Deep Neural Networks

Memory Optimizations

- `short` indices
- `half` weights (not used for Graph Challenge)
- Compact `struct` (not used for Graph Challenge)
- Batching for features
- Out-of-core streaming for weights



Out-of-core Streaming



Tiled SpMM for Sparse Deep Neural Networks

MIT/Amazon/IEEE Graph Challenge

grapchallenge.mit.edu

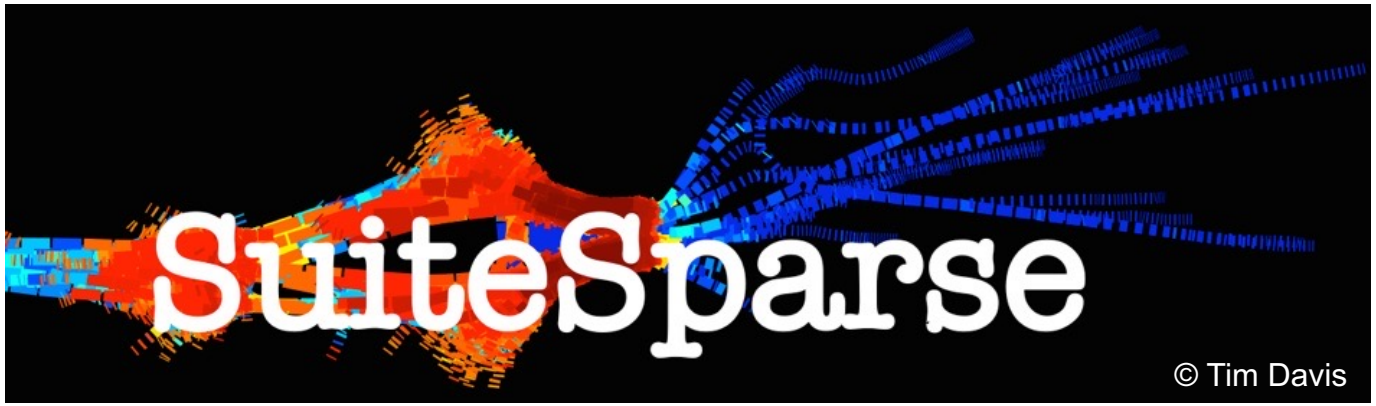
HPEC'20 Champion

Benchmark Throughput (# edges per second)

		This Work	Bisson & Fatica [18] 2019 Champion		Davis et al. [20] 2019 Champion		Ellis & Rajamanickam [21] 2019 Innovation		Wang et al. [22] 2019 Student Innov.		Wang et al. [23] 2019 Finalist	
Neurons	**Layers**	**Throughput**	**Throughput**	**Speedup**	**Throughput**	**Speedup**	**Throughput**	**Speedup**	**Throughput**	**Speedup**	**Throughput**	**Speedup**
1024	**120**	2.917E+13	4.517E+12	6.46	1.533E+11	190.28	2.760E+11	105.69	1.407E+11	207.32	8.434E+10	345.88
	480	2.930E+13	7.703E+12	3.80	2.935E+11	99.83	2.800E+11	104.64	1.781E+11	164.51	9.643E+10	303.84
	1920	2.883E+13	8.878E+12	3.25	2.754E+11	104.68	2.800E+11	102.96	1.896E+11	152.06	9.600E+10	300.30
4096	**120**	8.220E+13	6.541E+12	12.57	1.388E+11	592.22	2.120E+11	387.74	1.943E+11	423.06	6.506E+10	1,263.52
	480	8.222E+13	1.231E+13	6.68	1.743E+11	471.72	2.160E+11	380.65	2.141E+11	384.03	6.679E+10	1,230.99
	1920	8.232E+13	1.483E+13	5.55	1.863E+11	441.87	2.160E+11	381.11	2.197E+11	374.69	6.617E+10	1,244.02
16384	**120**	1.469E+14	1.008E+13	14.57	1.048E+11	1,401.53	1.270E+11	1,156.54	1.966E+11	747.10	3.797E+10	3,867.84
	480	1.394E+14	1.500E+13	9.29	1.156E+11	1,206.23	1.280E+11	1,089.38	2.060E+11	676.89	3.747E+10	3,721.66
	1920	1.464E+14	1.670E+13	8.77	1.203E+11	1,216.96	1.310E+11	1,117.56	1.964E+11	745.52	3.750E+10	3,903.72
65536	**120**	1.796E+14	9.388E+12	19.13	1.050E+11	1710.29	9.110E+10	1971.24	1.892E11	949.15	–	–
	480	1.703E+14	1.638E+13	10.40	1.091E+11	1,560.59	8.580E+10	1,984.38	1.799E+11	946.41	–	–
	1920	1.714E+14	1.787E+13	9.59	1.127E+11	1,520.59	8.430E+10	2,032.86	–	–	–	–

Hidayetoglu, et al., "At-Scale Deep Neural Network Inference with Efficient GPU Implementation," Arxiv and HPEC'20.

Generalization of Tiled SpMM



2,893 Sparse Matrices

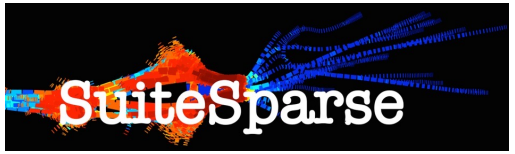
Picked the largest 200 samples in terms of nonzeroes

The matrices in the Collection cover a wide spectrum of domains, including

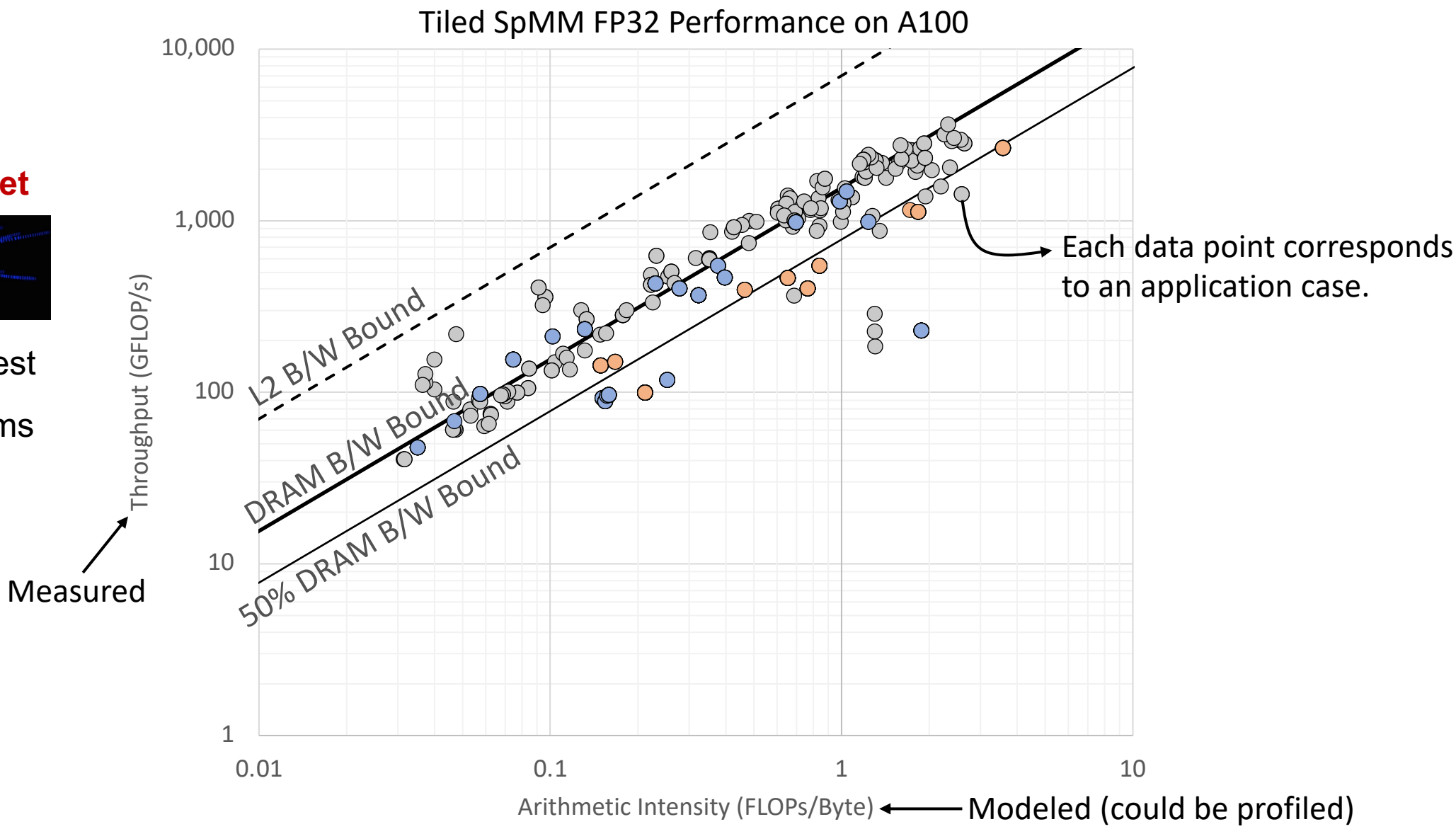
- 2D or 3D Geometric Domains
  - Structural engineering
  - Computational fluid dynamics
  - Model reduction
  - Electromagnetics
  - Semiconductor devices
  - Thermodynamics
  - Material science
  - Acoustics
  - Computer graphics and vision
  - Robotics and kinematics
  - Other spatial discretizations
- Non-Geometric Domains
  - Optimization
  - Economic and financial modeling
  - Circuit simulation
  - Theoretical and quantum chemistry
  - Chemical process simulation
  - Mathematics and statistics
  - Power networks
  - Other networks and graph-structured data

Tiled SpMM on Roofline Plot

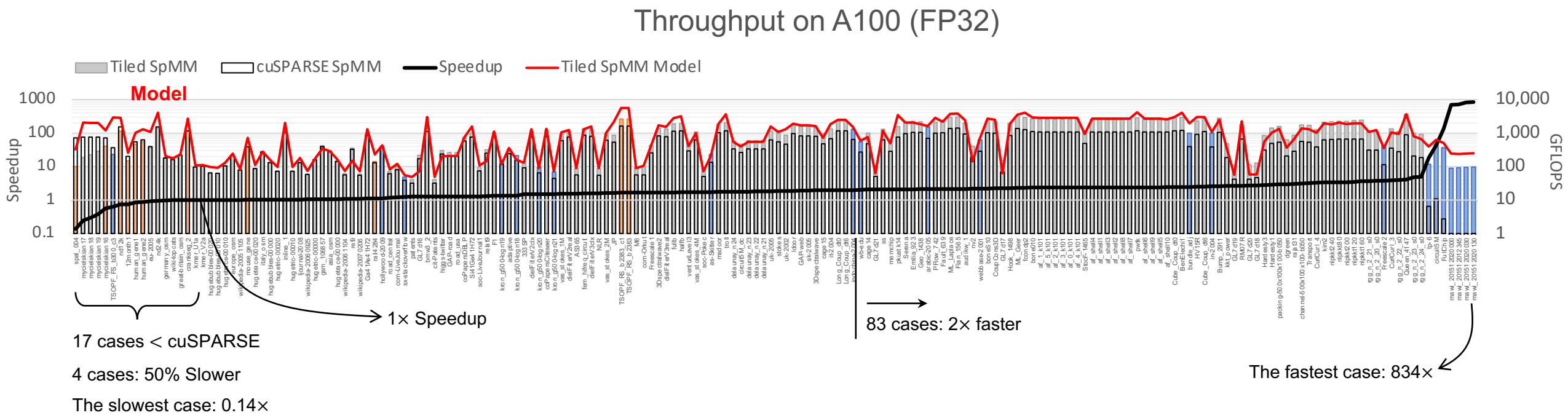
Benchmark Dataset



We picked the largest 200 samples in terms of nonzeroes.



Tiled SpMM vs. cuSPARSE



Speedup over cuSPARSE

(vendor-provided HPC Library)

Average: 19.2×

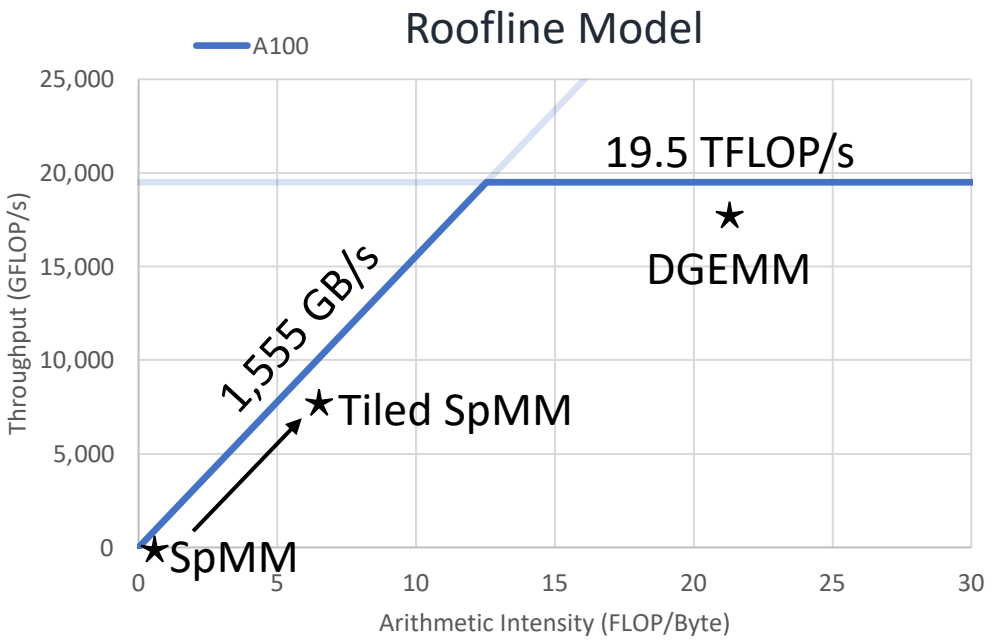
Geo-Mean: 2.03×

Har-Mean: 1.48×

Conclusion

Conclusion

Observation: the bottleneck of sparse computations is data movement over the slowest interconnect, rather than arithmetic operations.

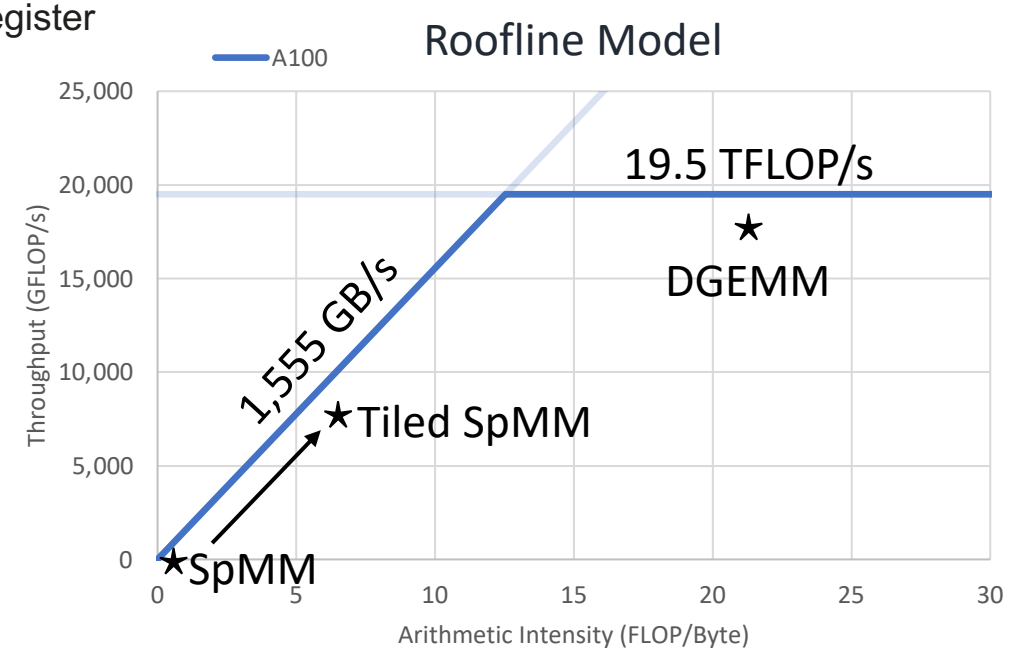


Conclusion

Observation: the bottleneck of sparse computations is data movement over the slowest interconnect, rather than arithmetic operations.

This Work: With the proposed tiling techniques and a careful orchestration of data movement over the memory hierarchy, we can accelerate SpMM on GPUs to a great extent.

DRAM->L2$->L1$/scratchpad->register



Conclusion

Observation: the bottleneck of sparse computations is data movement over the slowest interconnect, rather than arithmetic operations.

This Work: With the proposed tiling techniques and a careful orchestration of data movement over the memory hierarchy, we can accelerate SpMM on GPUs to a great extent.

We developed $\overbrace{\text{Tiled SpMM}}^{\textbf{Work in progress}}$ and …

- Benchmarked over 200 sparse matrices from a wide-spectrum of applications
- Verified performance bounds with the **proposed** analytical model
- Obtained 16x speedup over cuSPARSE

Product: Tiled SpMM is open source for other application developers.

https://github.com/merthidayetoglu/SpDNN_Challenge2020

Accelerating Sparse Deep Neural Network Inference on FPGAs

Sitao Huang

Assistant Professor, EECS, University of California Irvine

Organizers: Mert Hidayetoğlu, Wen-mei Hwu, Jinjun Xiong, Rakesh Nagi, Jeff Pool, Sitao Huang, Vikram Mailthody

Tutorial | MLSys 2022 | Aug 30, 2022

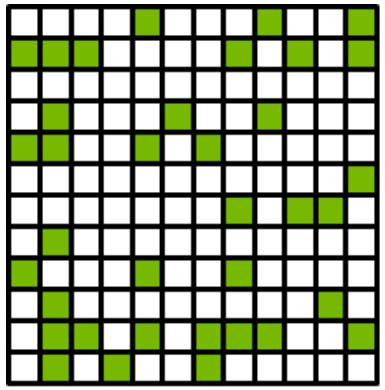
https://mlsys.org/virtual/2022/tutorial/2199

Tutorial Outline

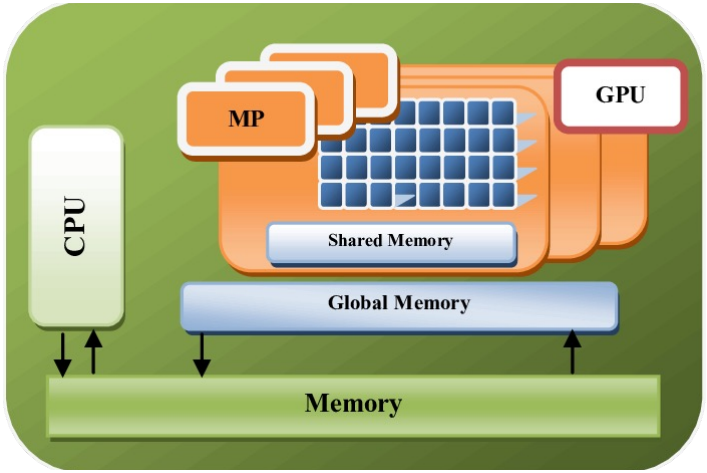
1. Opening remarks and overview of sparsity in ML – 1 pm
2. Tiled SpMM and its performance model on GPUs – 1:50 pm
3. **Sparse deep neural network inference on FPGAs – 2:45 pm**
4. Break (30 mins)
5. 2:4 Sparsity on GPU Tensor Cores – 4 pm
6. Future work and closing remarks – 4:45 pm

Three levels of optimization opportunities for SpMM on heterogeneous systems

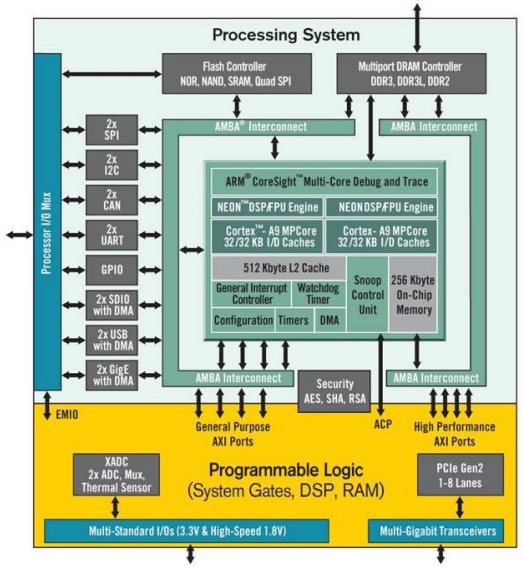
1. Optimize SpMM data structures and algorithms for generic hardware features



2. Co-optimize SpMM implementation for underlying hardware features



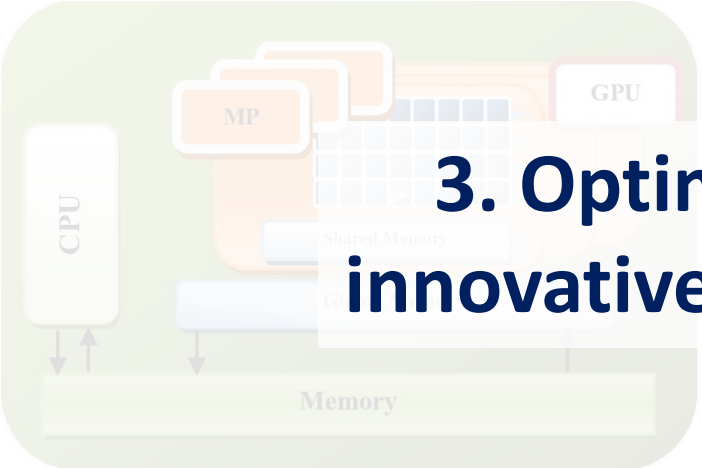
3. Optimize Hardware with innovative architecture features



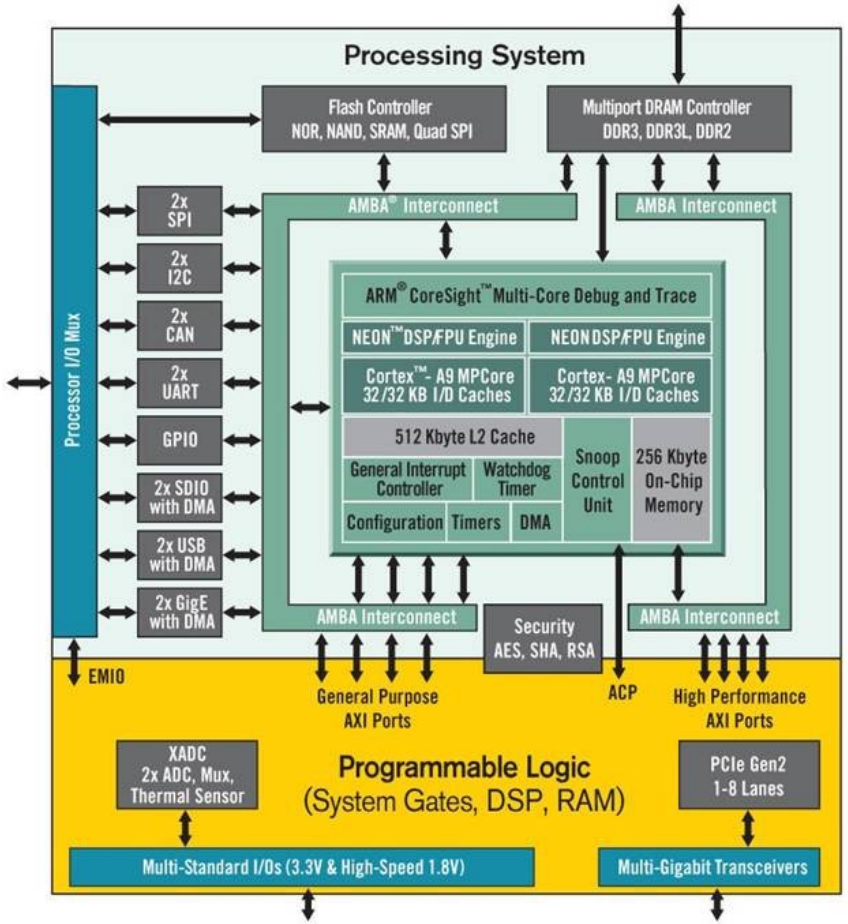
Three levels of optimization opportunities for SpMM on heterogeneous systems

1. Optimize SpMM data structures and algorithms for generic hardware features

2. Co-optimize SpMM implementation for underlying hardware features



3. Optimize Hardware with innovative architecture features



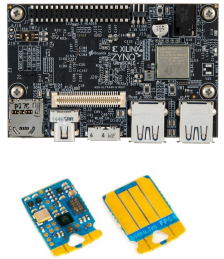
In This Talk

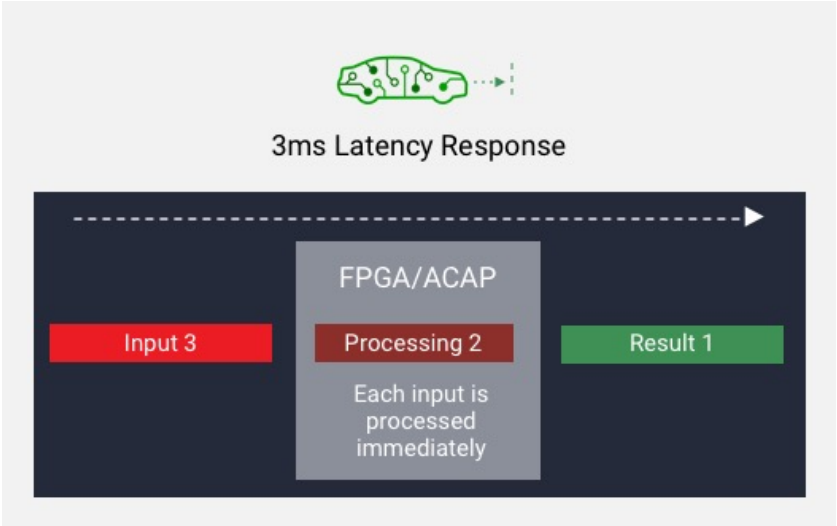
- Why FPGA?
- A Sparse DNN Inference Accelerator on FPGA
- PyLog: Python-based FPGA Design Flow
- Custom Sparse DNN Accelerators
- Future Topics

Why FPGAs?

FPGA: **F**ield **P**rogrammable **G**ate **A**rray

- Customizable, reconfigurable logic
- Low-latency, energy-efficient solutions
- Available in many different sizes
- Cloud computing, HPC, smart phone, edge devices, etc.
- Wide applications: medical, video processing, telecom, space, etc.



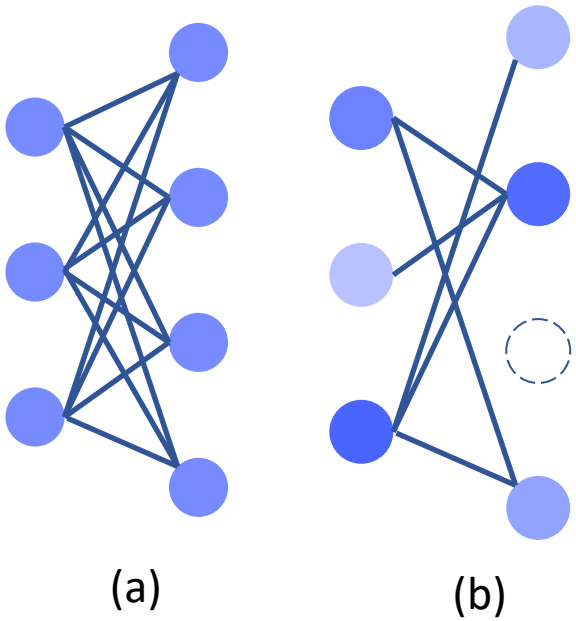


(Source: Xilinx)

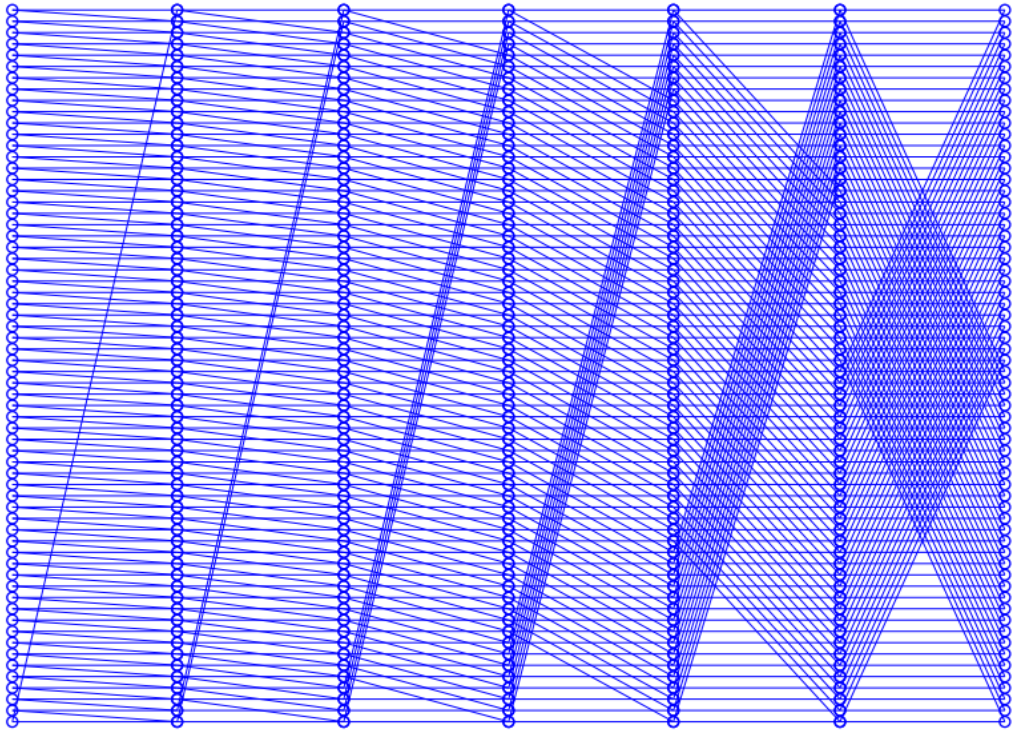
(Source: Microsoft)

A Sparse DNN Inference Accelerator on FPGA

- Accelerating *sparse* large-scale DNNs on FPGAs



(a) Fully connected layer;
(b) Sparse layer (after pruning)



A sparse DNN with 6 layers, 64 neurons per layer (synthetic, 2 connections per neuron) [1]

[1] J. Kepner et al. Sparse Deep Neural Network Graph Challenge. HPEC 2019.

A Sparse DNN Inference Accelerator on FPGA

Challenges

- Irregular: Frequent random accesses to memory
- Large-scale: from 120 layers, 1024 neurons/layer to 1920 layers 65536 neurons/layer (16.3 GB)

Computation

- Input and parameters are in graph format (sparse)
- Host CPU performs data preprocessing
- FPGA performs DNN inference: ℓ-th layer: $\mathbf{Y}_{\ell+1} = h(\mathbf{Y}_\ell \mathbf{W}_\ell + \mathbf{B}_\ell)$

N_a -> N_b: edge in graph

input.tsv:

N_a	N_b	*weight*
1323	10	1
55292	12	1
44323	14	1
1323	41	1
1323	42	1
...	...	...

param-L1.tsv:

N_a	N_b	*weight*
1	1	6.250000e-02
2	1	6.250000e-02
65	1	6.250000e-02
66	1	6.250000e-02
129	1	6.250000e-02
...	...	...

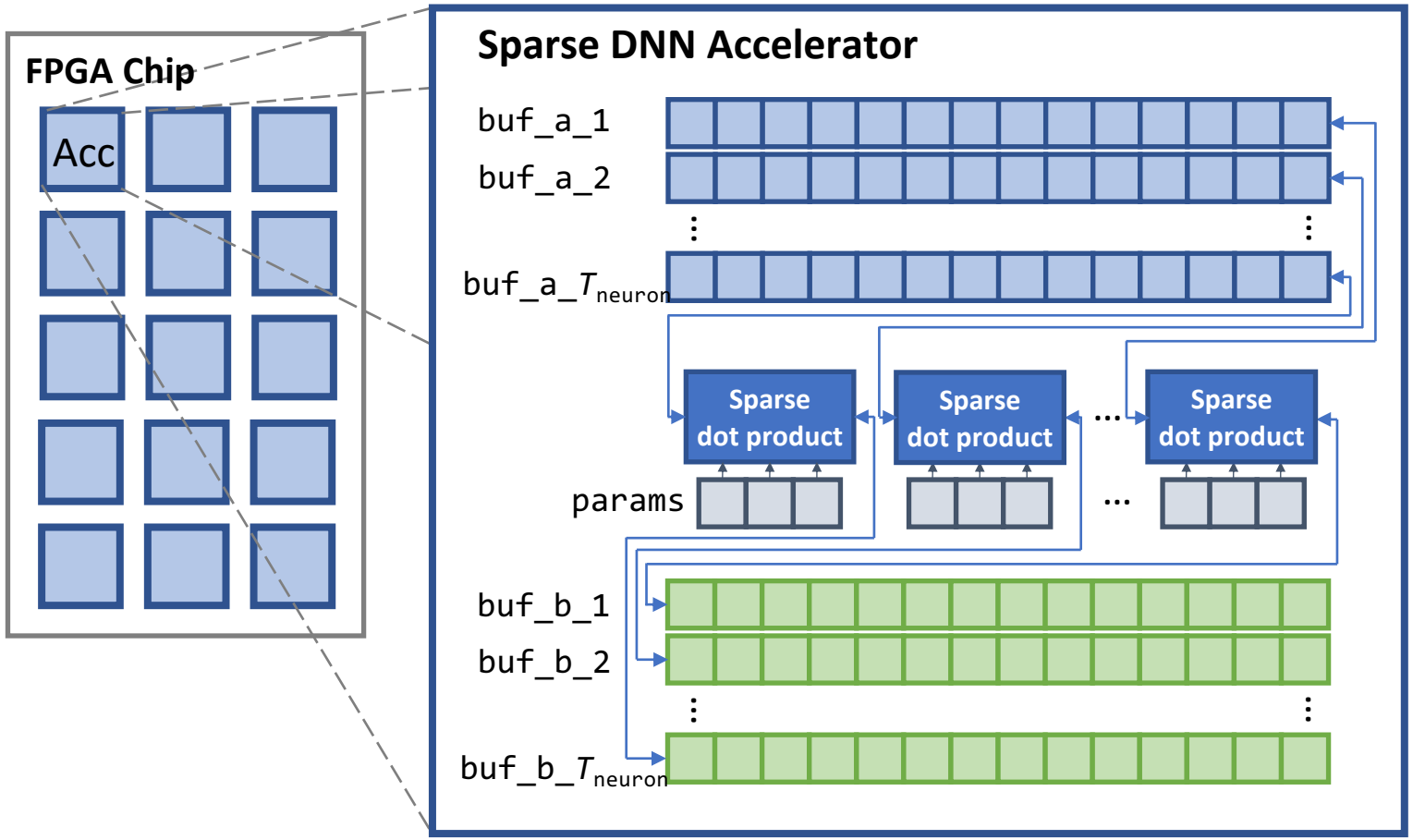
FPGA Accelerator Design

FPGA Design Challenges

- Limited on-chip resources (memory size, ports, DSP, etc.)
- Limited memory bandwidth
- Programmability

Our Solution

- Coarse-grained asynchronous accelerator pool
- Sparse dot product units with buffers
- Fully pipelined datapath
- Design created with C/C++ **high-level synthesis (HLS)**

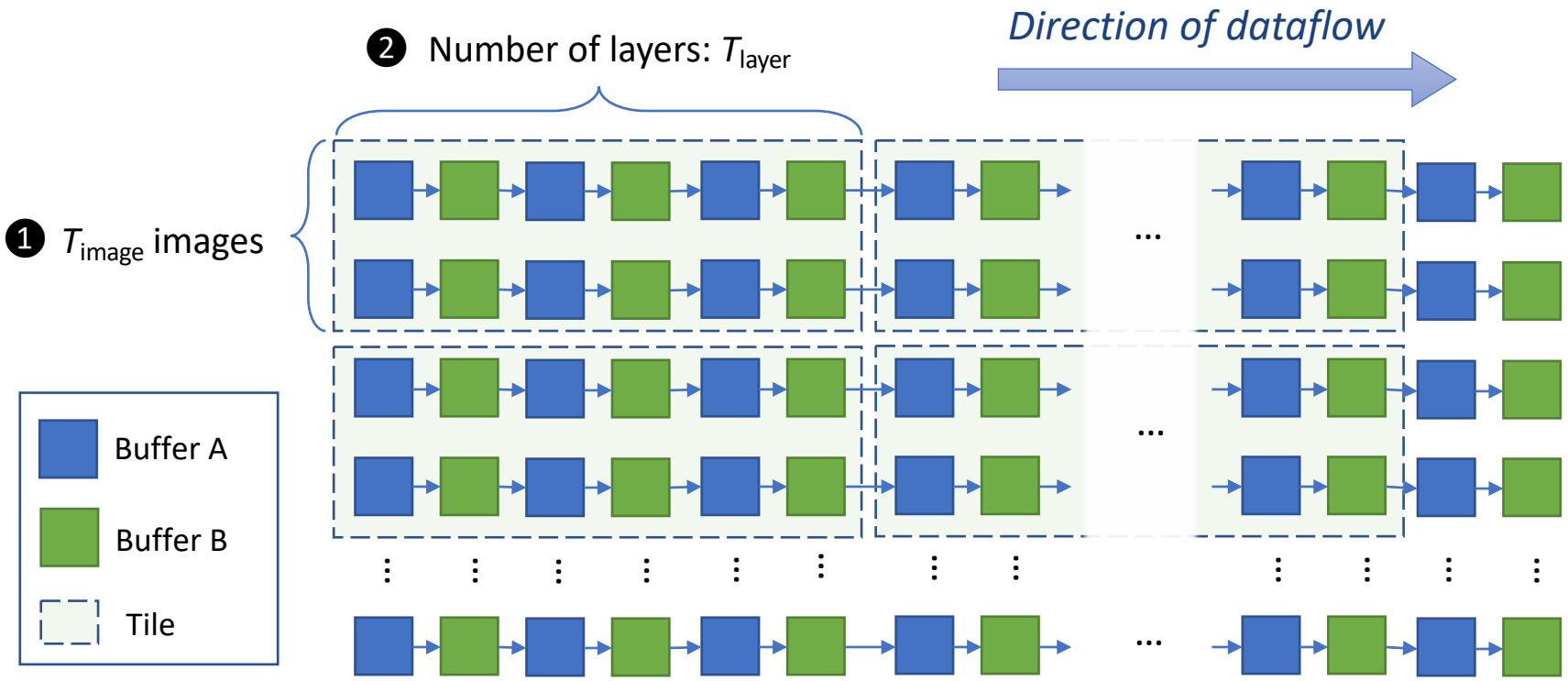


Sitao Huang, Carl Pearson, Rakesh Nagi, Jinjun Xiong, Deming Chen, and Wen-mei Hwu. ***Accelerating Sparse Deep Neural Network on FPGA*.** HPEC 2019.

Design Optimizations

Grid Representation

- Rows: input samples (images), *independent*
- Columns: DNN layers, *dependent*

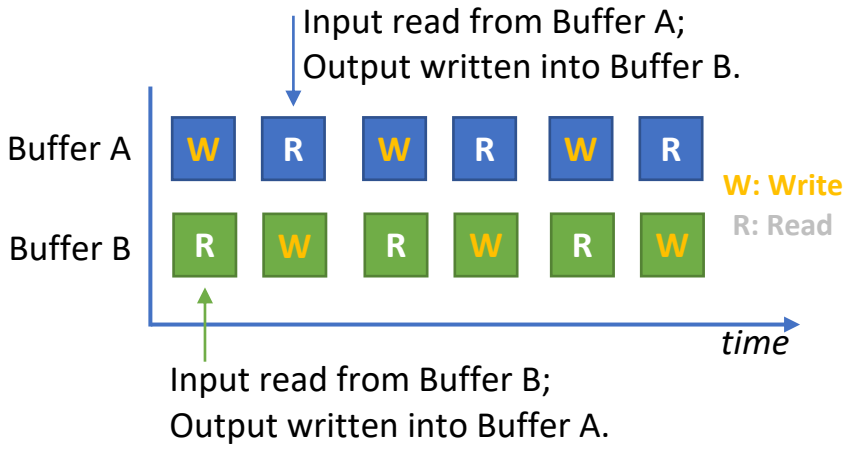


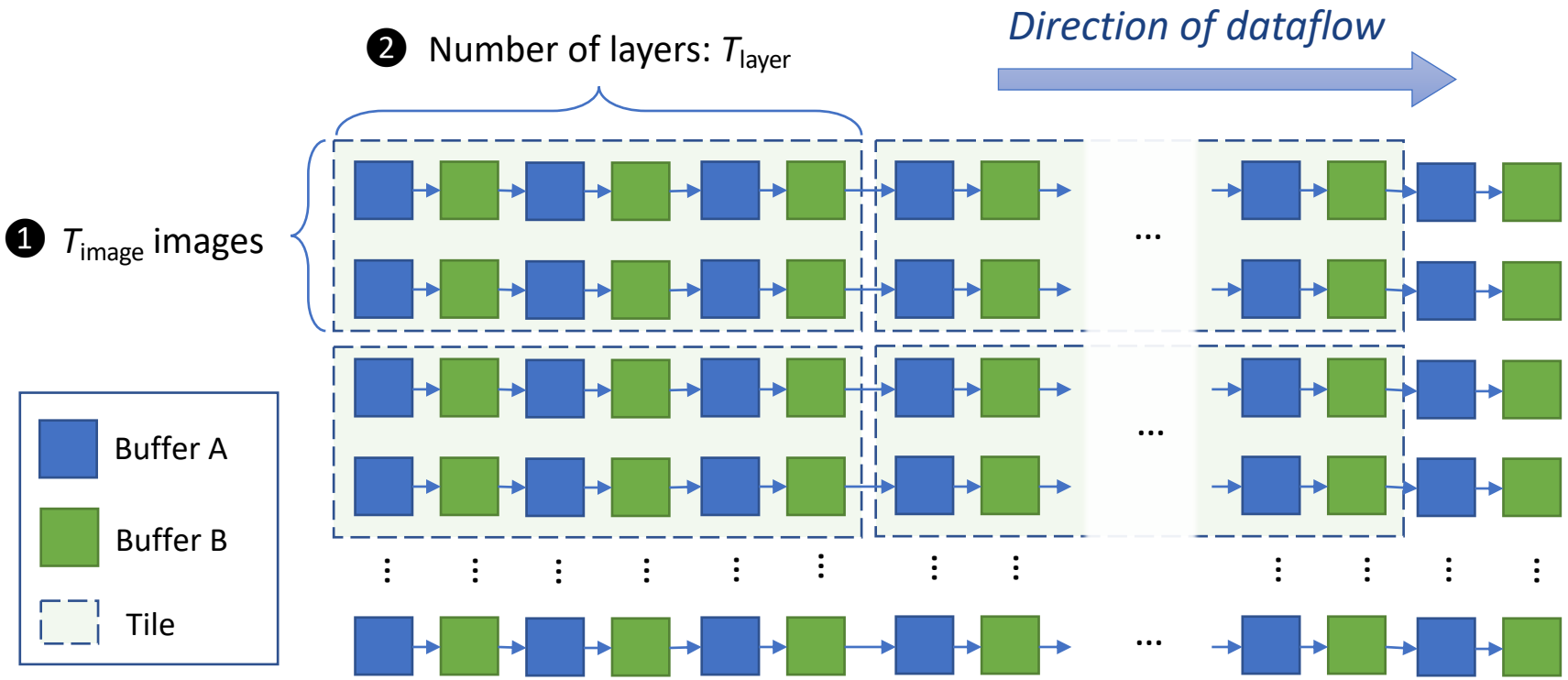
Computation illustrated with a 2D grid

Design Optimizations

Ping-Pong Buffering (Double Buffering)

- For both ***data loading*** and ***computation***
- Alternate *load* and *use* buffers (Buffer A and B)





Computation illustrated with a 2D grid

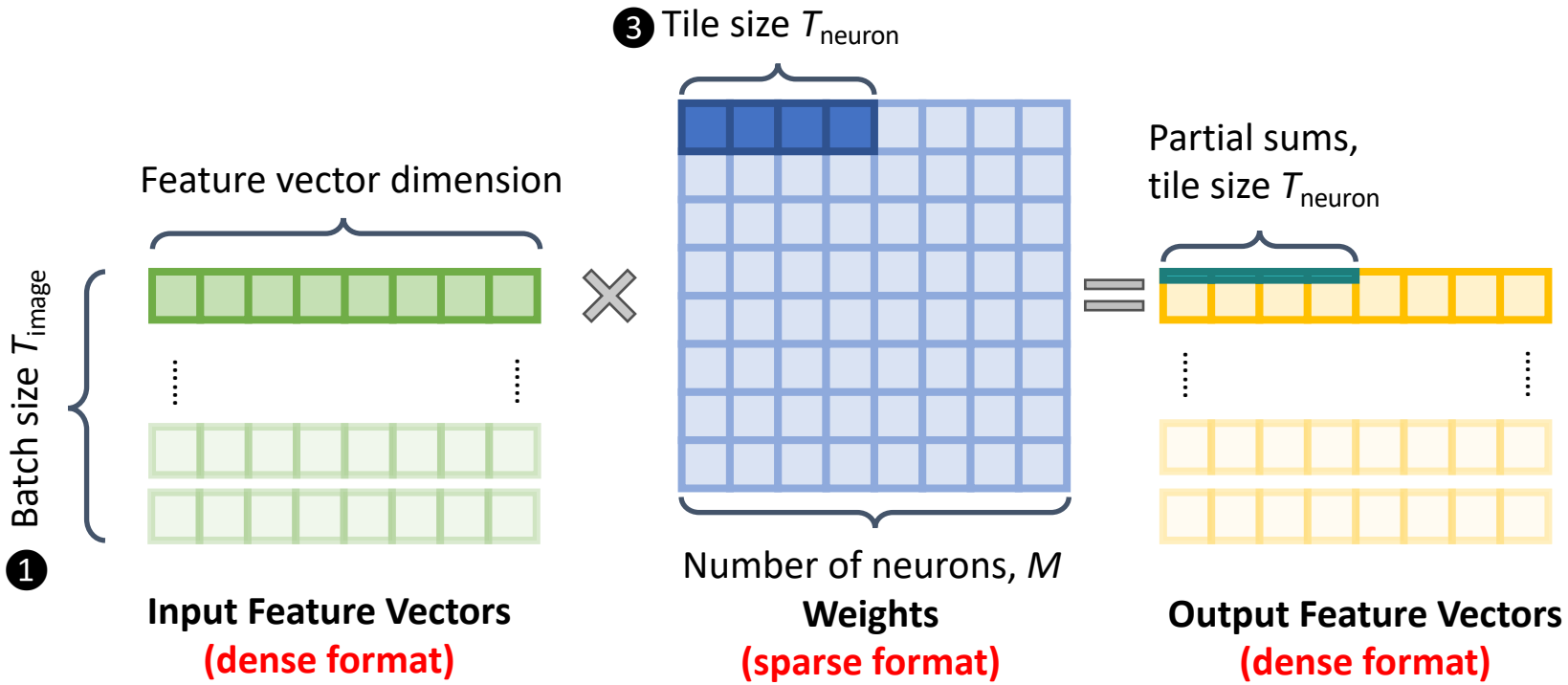
Other buffering optimizations in DNN accelerators:

- CNNs: fine-grained cross-layer pipelining
- Memory block partitioning to increase # of ports

Design Optimizations

Feature vectors and parameters: use sparse or dense format?

- Treating both as sparse: SpMV becomes set intersection
- Access pattern in parameters is regular while pattern in feature vectors is less regular
- Design decision: Feature vectors: dense format; parameters: sparse format

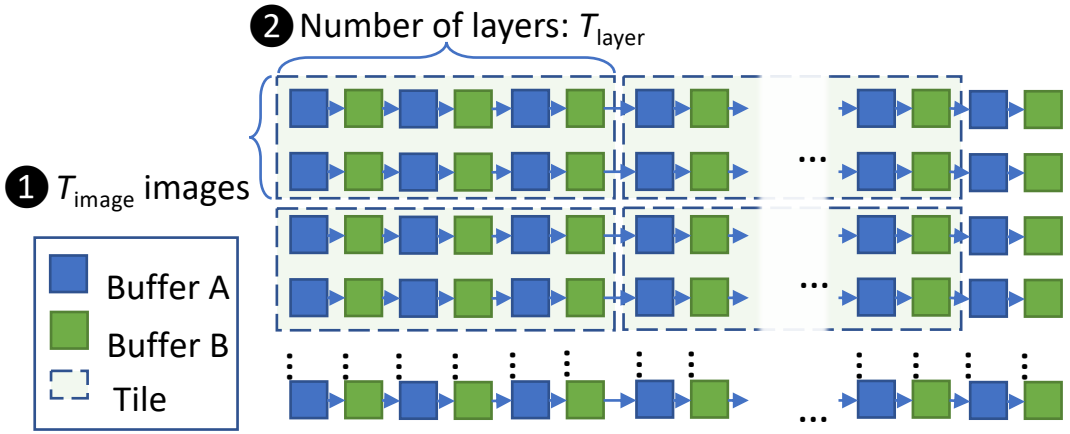


Tiled sparse vector-matrix multiplication

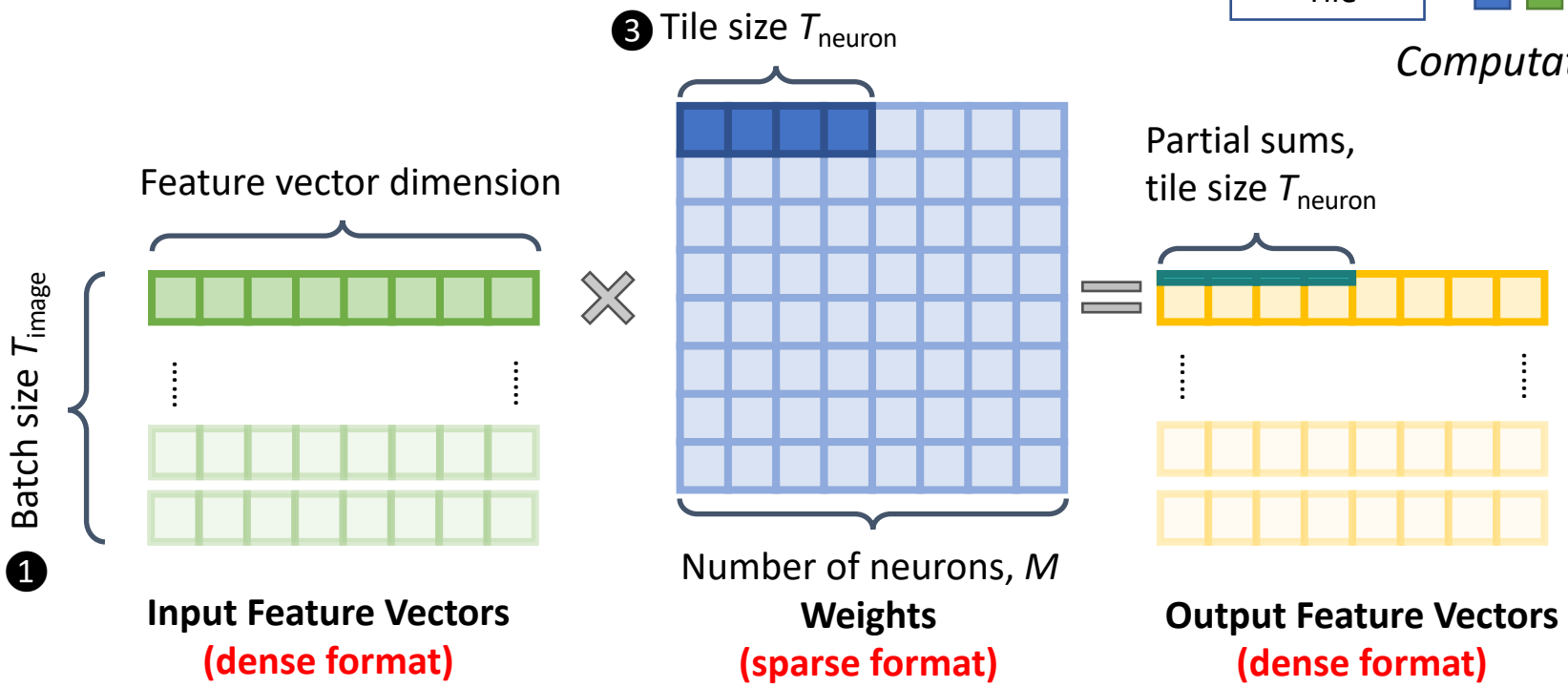
Design Optimizations

Multi-level tiling

- ❶ across input samples (input batch)
- ❷ Fused layers (inter-layer)
- ❸ within a layer, across neurons (intra-layer)



Computation illustrated with a 2D grid

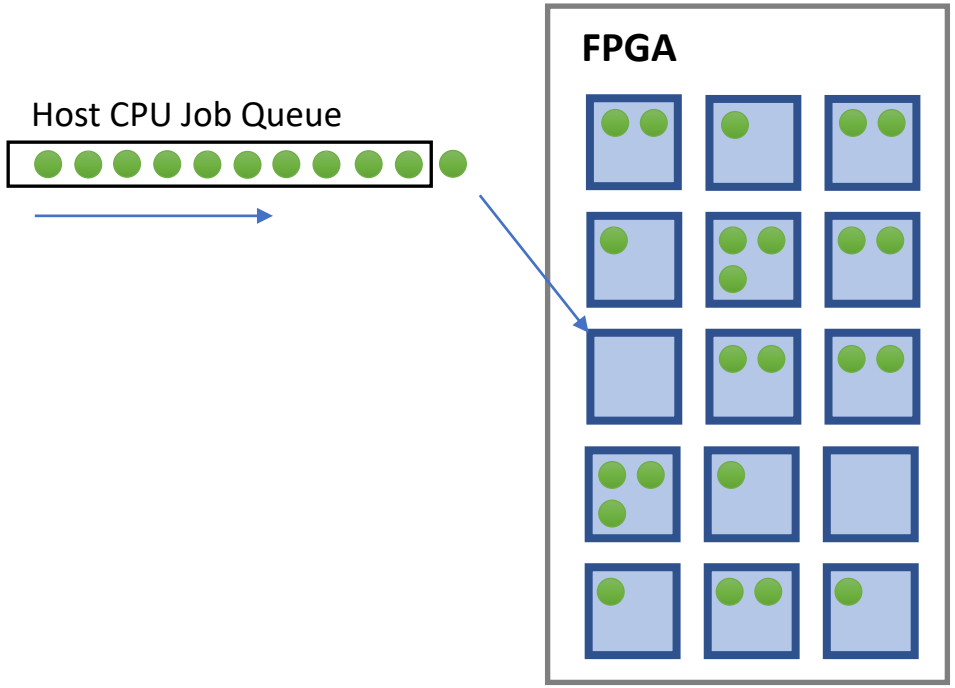


Tiled sparse vector-matrix multiplication

Design Optimizations

Dynamic workload balancing

- Multiple accelerators instantiated (at block design level)
- CPU dynamically assigns jobs to idle accelerators and collect returned results



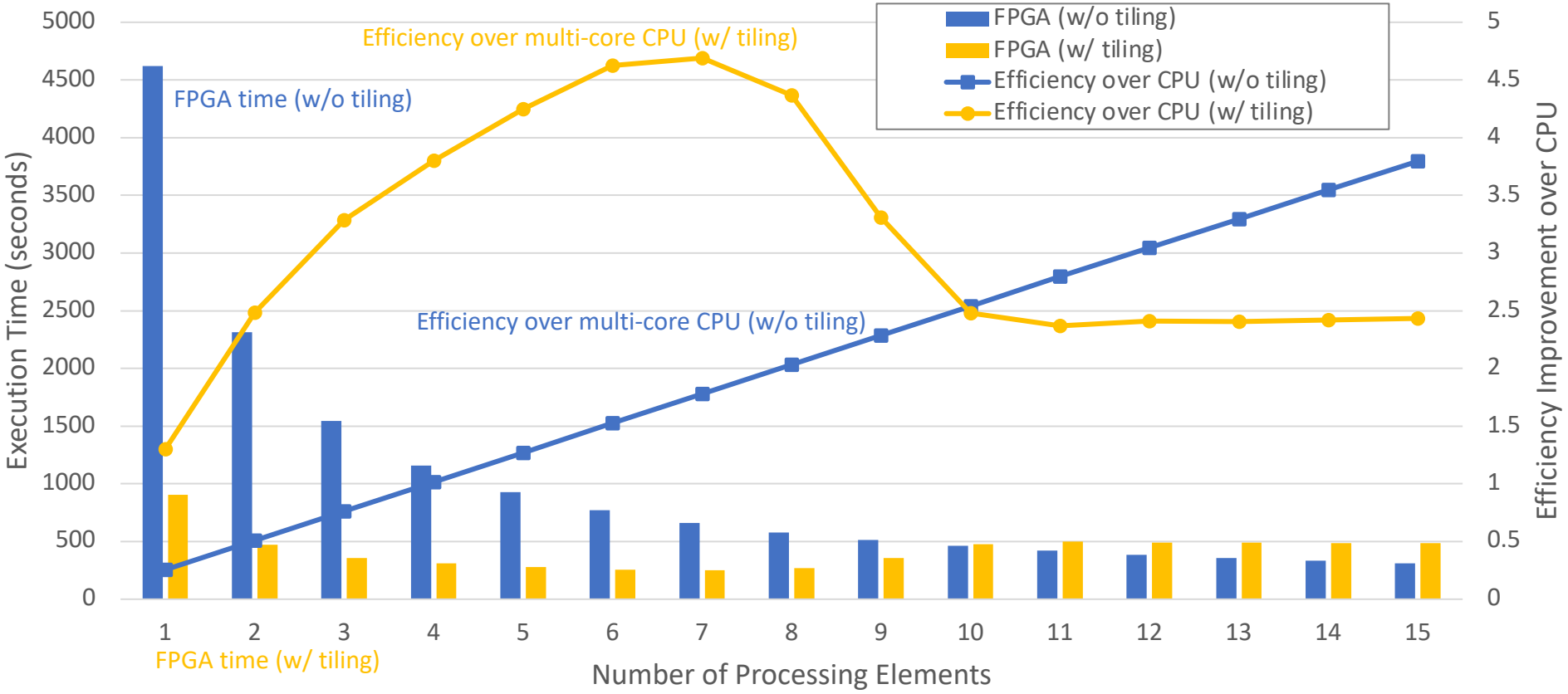
Algorithm 1 Dynamic Workload Assignment

Input: Number of input samples N, pack size S, accelerator pool `P` = {`P[0]`, ..., `P[m-1]`}.

```
curr_img ← 0, acc_ptr ← 0
size ← MIN(S, N − curr_img)
while curr_img < N do
    if P[acc_ptr].ISIDLE() then
        ASSIGN(curr_img, size, P[acc_ptr])
        curr_img ← curr_img + size
    else if P[acc_ptr].ISDONE() then
        COLLECTRESULTS(P[acc_ptr])
        ASSIGN(curr_img, size, P[acc_ptr])
        curr_img ← curr_img+size
    end if
    acc_ptr ← (acc_ptr+1)%m
end while
```

FPGA Accelerator Evaluation

- FPGA: Xilinx Virtex-7 VC709 board (250MHz)
- CPU baseline: AMD Operon 6272 Processors (64 cores), MATLAB Sparse Matrix library
- Dataset: 1024 neurons per layer x 120 layers (32x32 image classification)



High-Level Synthesis

- **High-level synthesis (HLS)**, also referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavior synthesis
- HLS takes an abstract behavioral specification of a digital system, and finds a register-transfer level structure that realizes the given behavior

INPUT:

- A high-level, algorithmic description
  - Control structures (if/else, loop, subroutines)
  - Concurrent and sequential semantics
  - Abstract data types
  - Logical and arithmetic operators
- A set of constraints
  - Speed, power, area, interconnect style
  - A library of pre-specified components

OUTPUT:

- A register-transfer level description for further synthesis and optimization

Reference: Register-Transfer Level (RTL), Abstract Data Type (ADT)

High-Level Synthesis Process

Resource Allocation:

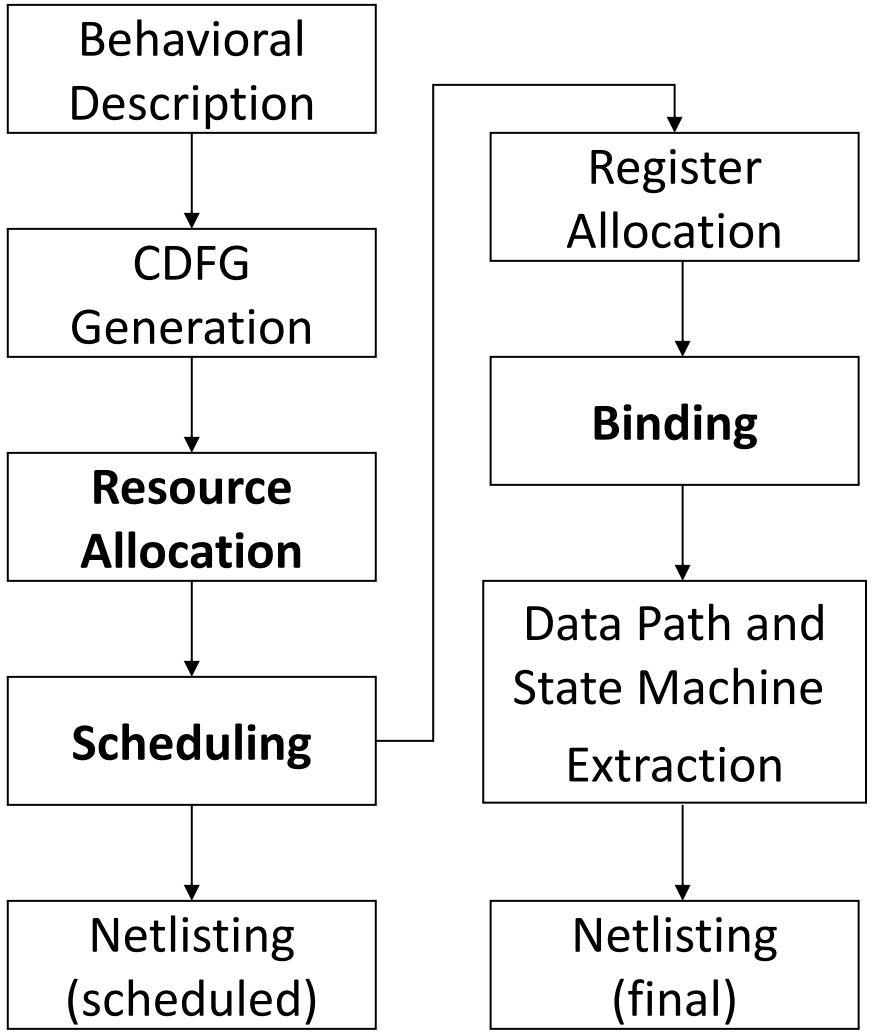
- Allocating resources (library components) to each of the operations, buses, MUXes, and registers for storage

Scheduling:

- Scheduling the operations in the control dataflow graph (CDFG) to minimize area, time and/or power

Binding:

- Determining the time of use of each component
- e.g., which registers to use and when



PyLog: Python-based FPGA Design Flow

PyLog Code

```python
import numpy as np
from pylog import *
@pylog
def add(a, b, c):
  c = a + b
  # c = map(lambda x, y: x+y, a, b)

if __name__ == "__main__":
  a = np.array([1, 3, ...])
  b = np.array([8, 9, ...])
  print(add(a, b, c))
```

accelerator

host

vecadd.py

PyLog Code Generator

- Analysis
- Type Inference
- Optimization
- Code Generation

HLS C Code

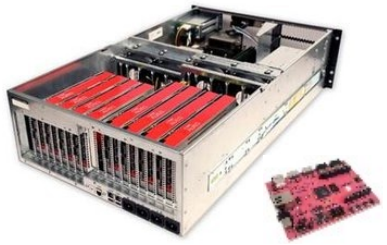
```c
void accel(int32* a, int32* b,
int32* c) {
  for(int i = 0; i < 32; i++) {
    for (ii...)
    #pragma HLS ...
  }
  ...
}
```

PyLog System Generator

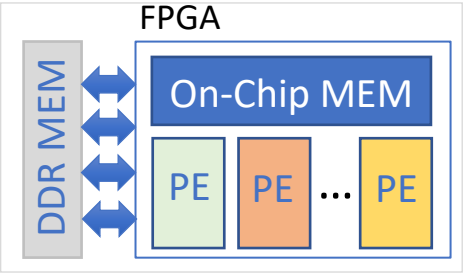
- Tcl Script
- IP Generation
- System Integration
- Design Generation

PyLog Runtime

- FPGA Programming
- Mem Allocation
- Start Accelerator
- Collect Results



Target Systems



System Design

```
>>> python3 vecadd.py
[9, 12, ...]
>>>
```

Execution Results

Synthesis Flow

Deployment Flow

https://github.com/hst10/pylog

Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, Wen-Mei Hwu.
PyLog: An Algorithm-Centric Python-based FPGA Programming and Synthesis Flow.
IEEE Transactions on Computers, 2021.

PyLog Language and Compiler Highlights

High-level operators, design space exploration

1D Conv:
```
y = map(lambda a: a[-1]+a[0]+a[1], x[1:-1])
```
2D Conv:
```
y = map(lambda a: dot(a[-1:2,-1:2], w), img[1:-1, 1:-1])
```

Multiple possible C code versions from high-level code

```
for (...) {
#pragma HLS unroll
}
```

```
for (...) {
#pragma HLS unroll factor=4
}
```

```
for (...) {
#pragma HLS pipeline
}
```

PyLog chooses the optimal implementation given the HW resource of the target platform

Unified FPGA and host programming

```
import numpy as np
from pylog import *

@pylog
def add(a, b):
    return a + b

if __name__ == "__main__":
    a = np.asarray([1, 3, 6, 7])
    b = np.asarray([8, 9, 10,5])

    print(add(a, b))
```

accelerator

host

Python statements, compute customization

```
@pylog
def pl_matmul(a, b, c, d):

  buf = np.empty([16, 16], pl_fixed(8, 3))
  pragma("HLS array_partition variable=buf")

  def matmul(a, b, c):
    for i in range(32):
      for j in range(32).unroll(4):
        tmp = 0.0
        for k in range(32).pipeline():
          tmp += a[i][k] * b[k][j]
        c[i][j] = tmp
```

Type inference and type checking

- No explicit type annotation required
- Top function has NumPy arrays as arguments, which carries input type and shape information
- Type engine infers type and shape of each object in the code

```
y = map(lambda a: dot(a[-1:2,-1:2], w), img)
```

pl_type: PLType(float, 0)
pl_shape: (0,)

pl_type: PLType(float, 2)
pl_shape: (3, 3)

pl_type: PLType(float, 2)
pl_shape: (27, 27)

pl_type: PLType(float, 0)
pl_shape: (0,)

(known)
pl_type: PLType(float, 2)
pl_shape: (3, 3)

(known)
pl_type: PLType(float, 2)
pl_shape: (27, 27)

PyLog High-Level Operations

- map operation: *map(f, x0, x1,...)*
  - Repeatedly apply function f to each element in *x0, x1,...*
  - Allow access neighbor elements inside function
- Dot operation: dot product of two arrays

Example: vector addition (a and b have same shape)

```
z = map(Lambda a, b: a + b, x, y)
```

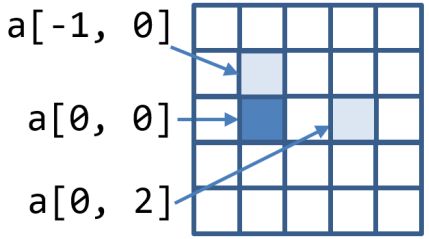
(a) *2D map*

PyLog

```
y = map(Lambda a0, a1,…: op(a0[-1, 0], a1[0, 2],…), x0, x1,...)
```

HLS C

```
L1: for(int i1 = 0; i1 < x0.dim(0); i1++) {
L2:   for(int i2 = 0; i2 < x0.dim(1); i2++) {
        y[i1][i2] = op(x0[i1-1][i2], x1[i1][i2+2],...); }}
```



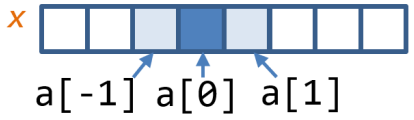
(b) *1D conv*

PyLog

```
y = map(Lambda a: a[-1]+a[0]+a[1], x[1:-1])
```

HLS C

```
L1: for(int i1 = 1; i1 < x.dim(0)-1; i1++) {
      y[i1] = x[i1-1] + x[i1] + x[i1+1]; }
```



(c) *2D conv*
(with multiple channels)

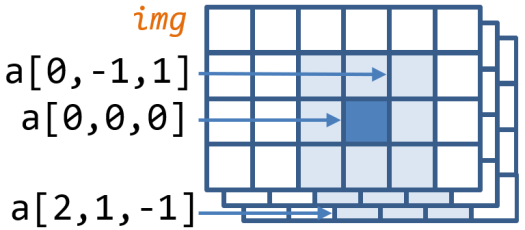
PyLog

```
y = map(Lambda a: dot(a[:,-1:2,-1:2], w), img[0,1:-1,1:-1])
```

(HLS C code omitted for simplicity)

3x3 multiplication and reduction across ***all*** *channels*

Image without outermost pixels



PyLog Supported Platforms

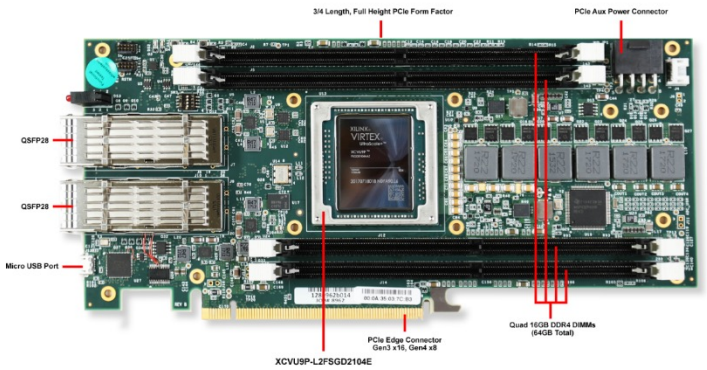
- Platforms supported by PyLog

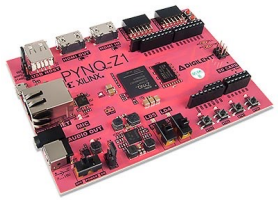
FPGA Platform Type	Platforms	Synthesis Flow	Runtime Library
SoCs and MPSoCs	ZedBoard, PYNQ, Ultra96	Vivado HLS + Vivado	PYNQ
PCIe-Based High-End FPGAs	Amazon EC2 F1 instance (XCVU9p), Alveo series (U200, U250, U280)	Vivado HLS + Vitis	PYNQ

- Design and deploy for different platforms easily
  - Simply change `@pylog` mode, no extra coding needed

Synthesize for AWS F1: `@pylog(mode='hwgen', board='aws_f1')`
Deploy on AWS F1: `@pylog(mode='deploy', board='aws_f1')`

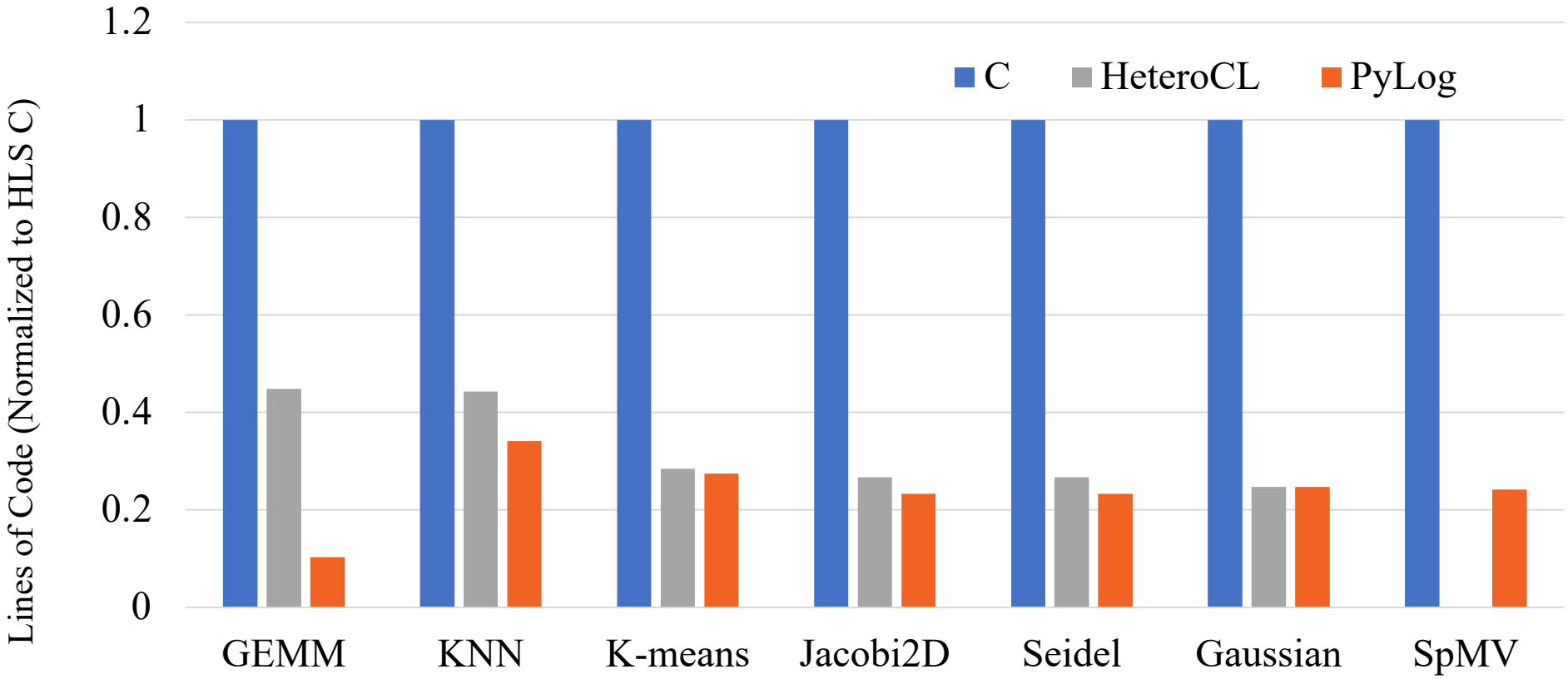
Synthesize for PYNQ: `@pylog(mode='hwgen', board='pynq')`
Deploy on PYNQ: `@pylog(mode='deploy', board='pynq')`





PyLog Evaluation: Expressiveness

- Compare LoC (Lines of Code) of HLS C code vs PyLog code
  - Normalized to HLS C LOC
  - **70%** reduction compared to HLS C code

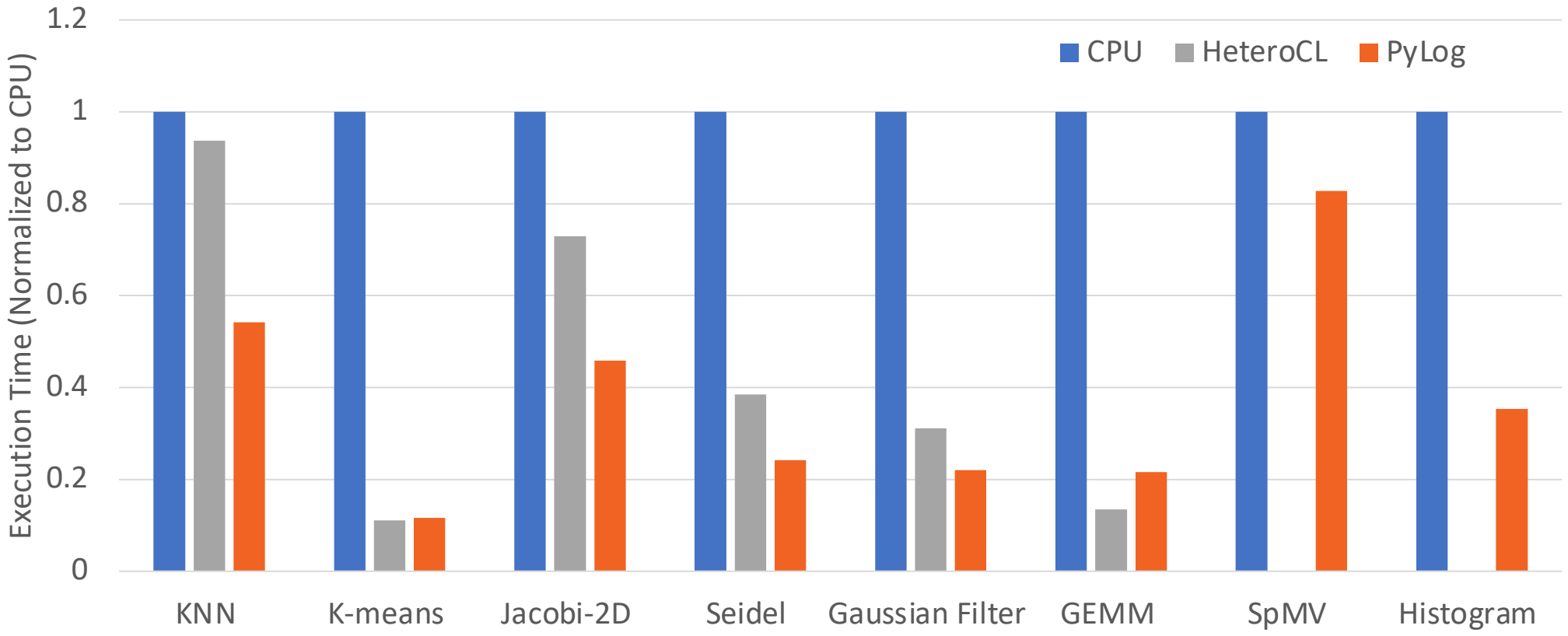


*Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. FPGA 2019.

PyLog Evaluation: Accelerator Performance

Compare accelerator performance vs CPU (C++) performance and HeteroCL accelerator performance

- Platform: Amazon AWS F1 instance (Xilinx Virtex UltraScale+ XCVU9P)
- **3.17x** and **1.24x** faster than optimized CPU and FPGA accelerators



CPU: Optimized CPU code running on single thread on 8-core Intel Xeon E5-2686 v4 CPU

SpMV in PyLog

- Baseline version

```
@pylog
def spmv(row_ptr, column_idx, values, y, x):
    for i in range(num_neuron):
        y0 = 0
        for k in range(row_ptr[i], row_ptr[i+1]):
            y0 += values[k] * x[column_idx[k]]
        y[i] = y0
```

SpMV in PyLog

- (lightly) optimized version

```
@pylog
def spmv(row_ptr, column_idx, values, y, x):
    for i in range(64).pipeline():
        y0 = 0
        for k in range(row_ptr[i], row_ptr[i+1], 7):
            pragma('HLS pipeline II=7')
            yt = values[k] * x[column_idx[k]]

            for j in range(7):
                if ((k+j) < row_ptr[i+1]):
                    yt += values[k+j] *x[column_idx[k+j]]
            y0 += yt
        y[i] = y0
```

PyLog is open-source. Give it a try!

https://github.com/hst10/pylog

PyLog usage

To use PyLog, import pylog and simply add PyLog decorator `@pylog` to the function that you'd like to synthesize into an FPGA accelerator. Pass NumPy arrays to the decorated function and call the decorated function. Then run the whole Python program. In the following example, `vecadd` functior

```
import numpy as np
from pylog import *

@pylog
def vecadd(a, b, c):
    for i in range(1024):
        c[i] = a[i] + b[i]
    return 0

if __name__ == "__main__":
    length = 1024
    a = np.random.rand(length).astype(np.float32)
    b = np.random.rand(length).astype(np.float32)
    c = np.random.rand(length).astype(np.float32)

    vecadd(a, b, c)
```

You can also pass arguments to `@pylog` decorator to control the b
passed to `@pylog`:

- `mode`: You can pass one of the following strings to control the
  - `'cgen'` or `'codegen'`: Generates HLS C code only;
  - `'hwgen'`: Generates HLS C code and call Vivado HLS and
  - `'pysim'`: Run the code with standard Python interpreter. code to use `pysim`;
  - `'deploy'` or `'run'` or `'acc'`: Run PyLog in deploy mod invoke FPGA and collect results.
- `path`: This overwrites the `WORKSPACE` string in `pylog.py`.
- `board`: The target FPGA board. Currently PyLog support `zedb`

353 lines (353 sloc) | 8.29 KB

Raw Blame

PyLog HLS C Code Generation

Preparation

Set PyLog path and import PyLog.

In [ ]:
```
import sys
sys.path.extend(['/home/shuang91/pylog/'])

import numpy as np
from pylog import *
```

Subscripts

PyLog supports Python subscripts with arbitrary experssions.

To annotate a Python function for compilation, simply decorate the function with `@pylog` decorator.

In [ ]:
```
@pylog
def pl_subscript(a, b):
    for i in range(4, 15, 2):
        a[b[i]][f(a * b + 13)][csdf*w34 - 1] = b
    return c
```

Call the decorated Python function with sample input will compile the function:

In [ ]:
```
a = np.array([1, 3, 6, 7, 10])
b = np.array([1, 3, 6, 7, 10])
c = pl_subscript(a, b)
```

Now let's take a look at the generated HLS C code:

In [ ]:
```
!pygmentize /home/shuang91/vivado_projects/pylog_projects/pl_subscript/pl_subscript.cpp
```

PyLog 2.0: New Features

PyLog x Heterogeneous Computing

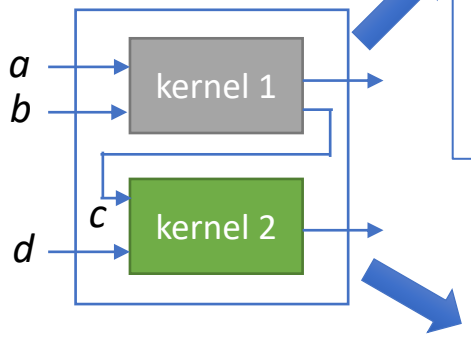
```
@pylog
def acc1(a, b):
    ...
@pylog
def acc2(c, d):
    ...
if __name__ == "__main__":
    a = np.array([1, 3, ...])
    b = np.array([8, 9, ...])
    d = np.array([8, 9, ...])
    combine(acc1, acc2)(a, b, d)
```

kernel 1

kernel 2

host

- Returns a new function that combines kernels
- Connects inputs and outputs with kernels



Generic programmable logic: Vitis connectivity configuration

```
[connectivity]
#nk=<kernel name>:<number>:<cu_name>.<cu_name>...
nk=vadd:3:vadd_X.vadd_Y.vadd_Z
#sp=<compute_unit_name>.<interface_name>:<bank name>
sp=cnn_1.m_axi_gmem:DDR[0]
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>
stream_connect=vadd_1.stream_out:vadd_2.stream_in
```

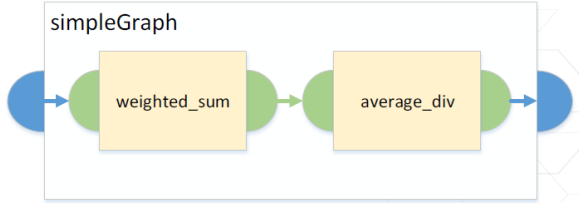
Creating Multiple Instances of a Kernel

Mapping Kernel Ports to Global Memory

Specify streaming connections

AI Engines: Dataflow model (on-going)

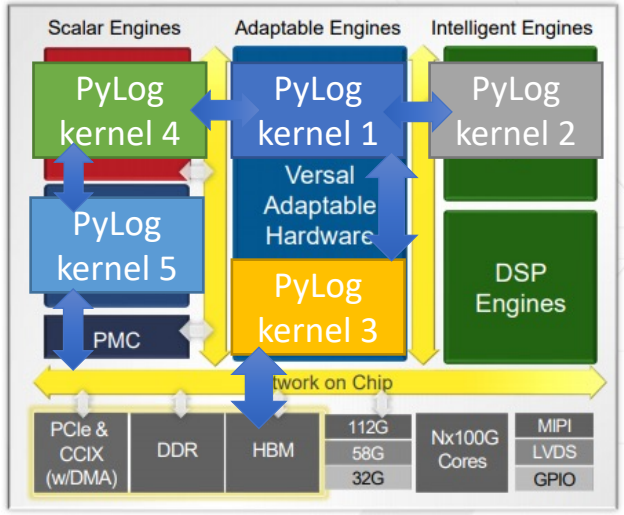
Connecting Kernels in graph.h




```
simpleGraph () {
  // Bind a function to each of the declared kernels
  k1  = kernel::create(weighted_sum);
  k2  = kernel::create(average_div);

  // create nets to connect kernels and IO ports
  connect<window<128> > net0 (in, k1.in[0]);
  connect<window<128> > net1 (k1.out[0], k2.in[0]);
  connect<window<128> > net2 (k2.out[0], out);
};
```

- New heterogeneous platforms



Xilinx Versal devices

Collaborative computing:

- Across multiple adaptable logic kernels
- Across scalar engines, adaptable engines, and AIEs
- Across multiple ACAP / FPGA devices

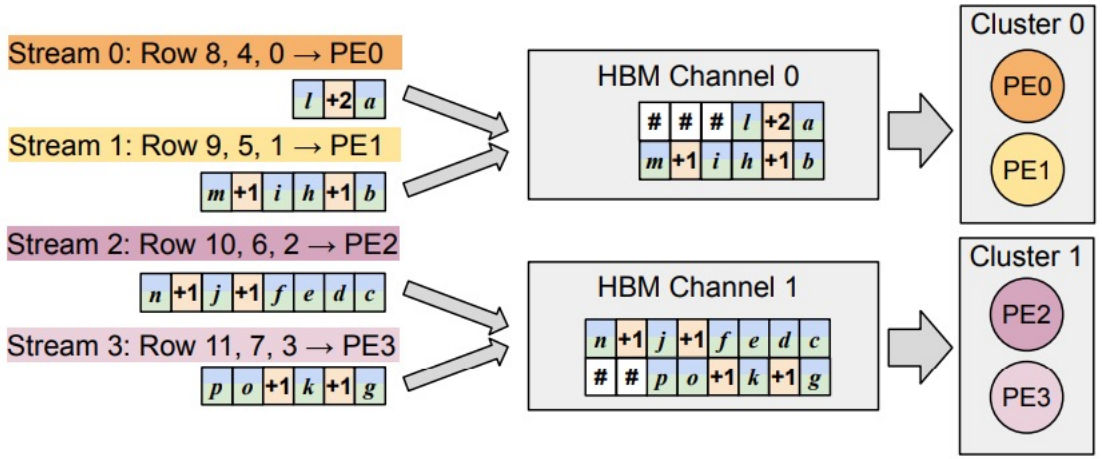
Data placement and movement:

- Data sources: PCIe, DDR, HBM, Ethernet, other engines, etc.

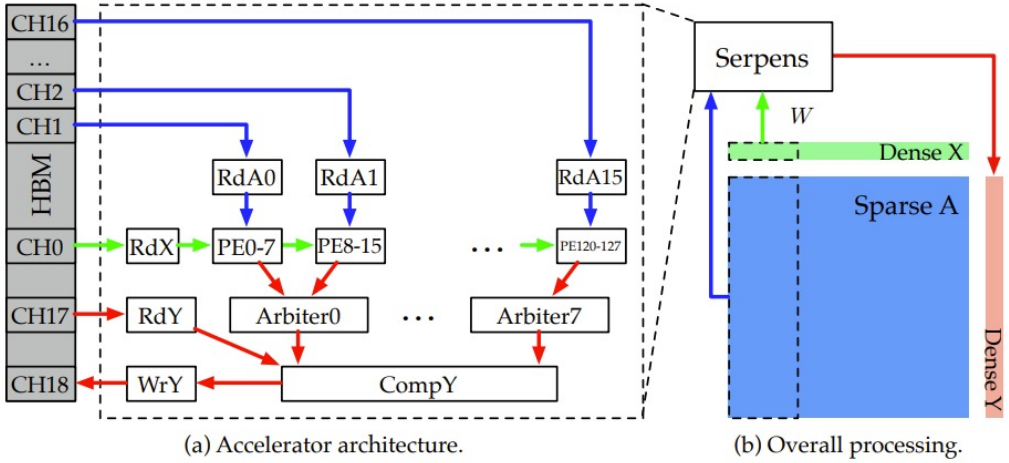
Other Optimizations for Sparse Accelerators

- Optimizations for HBM [1, 2]
  - HBM bandwidth utilization
  - On-chip memory utilization
  - Compute occupancy
- New compute patterns [3]
  - Outer product based GEMM
  - Huffman tree scheduler for balanced reduction

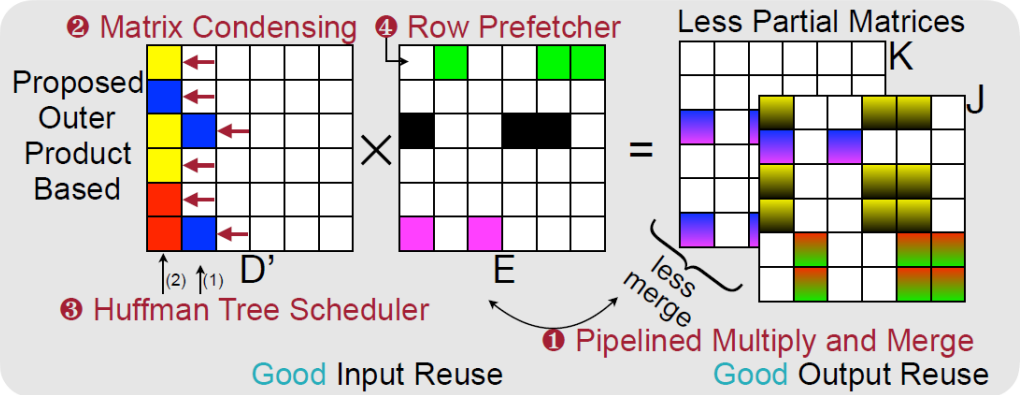
[1]



[2]



[3]



[1] Yixiao Du , Yuwei Hu , Zhongchun Zhou , Zhiru Zhang. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. FPGA 2022.
[2] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication. DAC 2022.
[3] Zhekai Zhang, Hanrui Wang, Song Han, William J. Dally. SpArch: Efficient Architecture for Sparse Matrix Multiplication. HPCA 2020.

Future Works

- New optimizations for new architecture features
  - HBM, CXL, etc.
  - Hard accelerator cores
- Higher level programming abstractions and synthesis flow
  - Python-based programming to improve programmability and DSE efficiency
- System-level optimizations for heterogeneous platforms
  - Extending optimizations to system level
  - Support heterogeneous environments with diverse accelerators

Summary

- Optimizing sparse computation on custom accelerators
  - Understand platform strength and limitations
  - Balance memory vs compute
  - Exploring design space of multiple dimensions
- High-level programming flows improves productivity
  - PyLog: Python-based FPGA programming flow
  - PyLog 2.0: supporting heterogenous systems
  - Opens up new dimensions of design space
- Future directions

2:4 Sparsity on GPU Tensor Cores

Jeff Pool and Rakesh Nagi

jpool@nvidia.com nagi@illinois.edu

Tutorial | MLSys 2022 | Aug 30, 2022

https://mlsys.org/virtual/2022/tutorial/2199

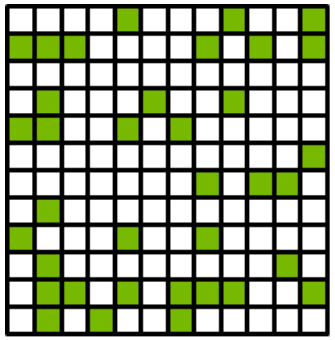
Outline

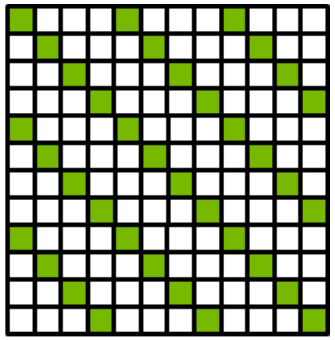
1. 2:4 Structured Sparsity – Why and How
2. Performance
3. Maintaining Accuracy
4. Channel Permutations
5. Q&A

2:4 Sparsity: Why and How – Sparsity Taxonomy

Structure:

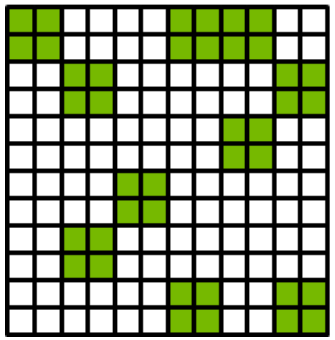
- Unstructured: irregular, no pattern of zeros
- Structured: regular, fixed set of patterns

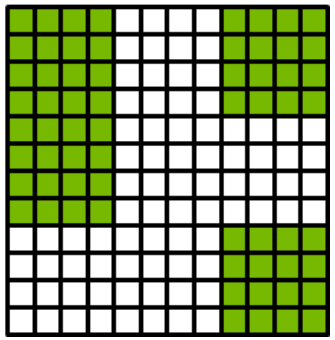




Granularity:

- Finest: prune individual values
- Coarser: prune blocks of values
- Coarsest: prune entire layers





2:4 Sparsity: Why and How – State of Sparsity Research

Lots of research in two areas:

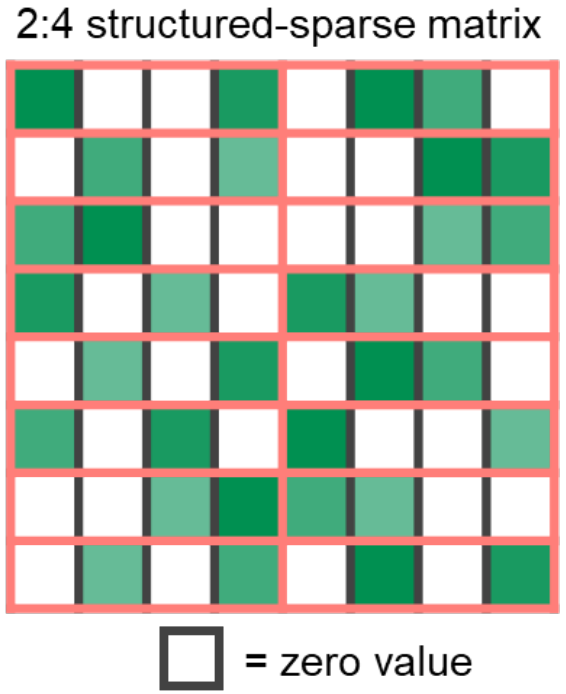
- High amounts (80-95%) unstructured, fine-grained sparsity
- Coarse-grained sparsity for simpler acceleration

Challenges not resolved for these approaches:

- **Accuracy** loss
  - High sparsity often leads to accuracy loss of a few percentage points, even after advanced training techniques
- Absence of a training approach that works across different tasks and networks (**Recipe**)
  - Training approaches to recover accuracy vary from network to network, often require hyper-parameter searches
- Lack of **speedup**
  - Math: unstructured data struggles to take advantage of modern vector/matrix math instructions
  - Memory access: unstructured data tends to poorly utilize memory buses, increases latency due to dependent sequences of reads
  - Storage overheads: metadata can consume 2x more storage than non-zero weights, undoing some of compression benefits

Fine-grained structured sparsity for Tensor Cores

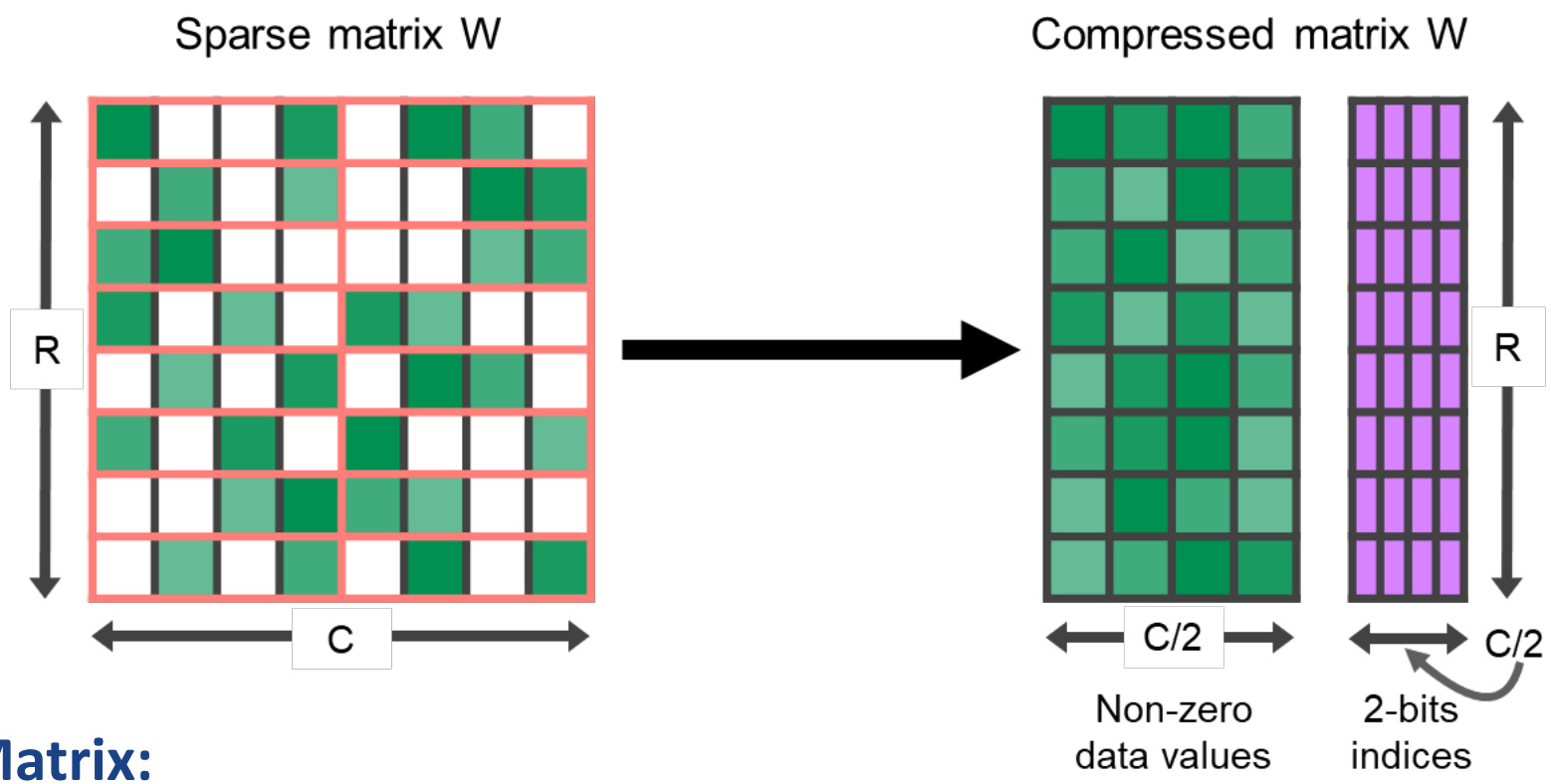
- 50% fine-grained sparsity
- 2:4 pattern: 2 values out of 4 must be zero



Addresses the three challenges:

- **Accuracy:** maintains accuracy of the original, unpruned network
  - Medium sparsity level (50%), fine-grained
- **Recipe**: shown to work across tasks and networks
- **Speedup**
  - Math: Specialized Tensor Core support for sparse math
  - Structured nature allows aligned, regular memory accesses and minimal metadata overheads

2:4 Sparsity: Why and How – 2:4 Compressed Matrix Format



Compressed Matrix:

- Data: ½ size
- Metadata:

  2b per nonzero element (index into its location in the 4-element region)

  16b data: 12.5% overhead

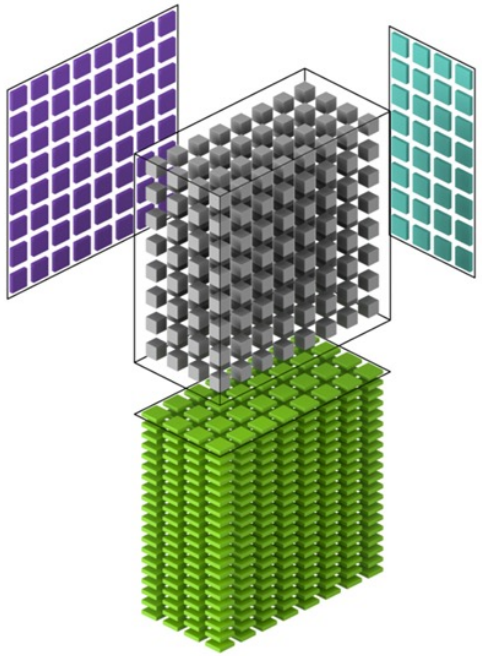
  8b data: 25% overhead

2:4 Sparsity: Why and How – Sparse Tensor Cores

Applicable for:

- Convolutions
- GEMMs of all types (linear layers, MLPs, recurrent cells, transformer blocks, …)

Inputs: sparse (weights), dense (activations)
Outputs: dense (activations)



Outline - Performance

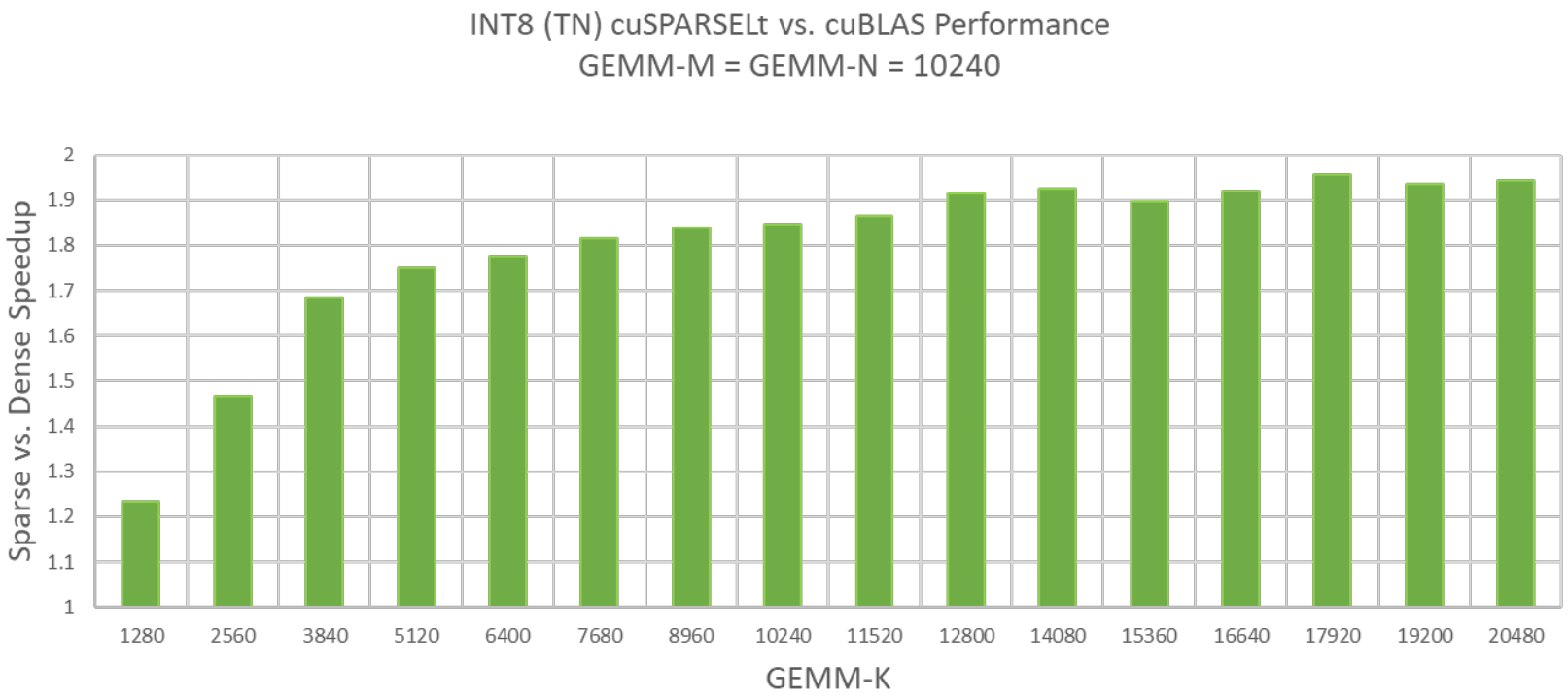
1. 2:4 Structured Sparsity – Why and How
2. Performance
    1. Tensor Core
    2. GEMM Operation
    3. Full Network

Sparse Tensor Cores are 2x Faster

INPUT OPERANDS	ACCUMULATOR	A100 TOPS	Dense vs. FFMA	Sparse Vs. FFMA
FP32	FP32	19.5	-	-
TF32	FP32	156	8X	16X
FP16	FP32	312	16X	32X
BF16	FP32	312	16X	32X
FP16	FP16	312	16X	32X
INT8	INT32	624	32X	64X

Sparse GEMMs are up to 1.95x Faster

- Snapshot of cuSPARSELt and cuBLAS performance
- A100, 1410MHz



https://docs.nvidia.com/cuda/cusparselt/index.html

Performance – Network

A realtime detection network is 1.56x faster

- RetinaNet with a ResNet-34 backbone
- 1920x1280 input, batch size = 1
- NVIDIA Orin SOC, 1275MHz

Dense Latency (ms)	Sparse Latency (ms)	Speedup
15.66	10.07	1.56x

Outline

1. 2:4 Structured Sparsity – Why and How
2. Performance
3. Maintaining Accuracy
    1. Goals
    2. Recipe
    3. Results

Maintaining Accuracy – Goals for a Training Recipe

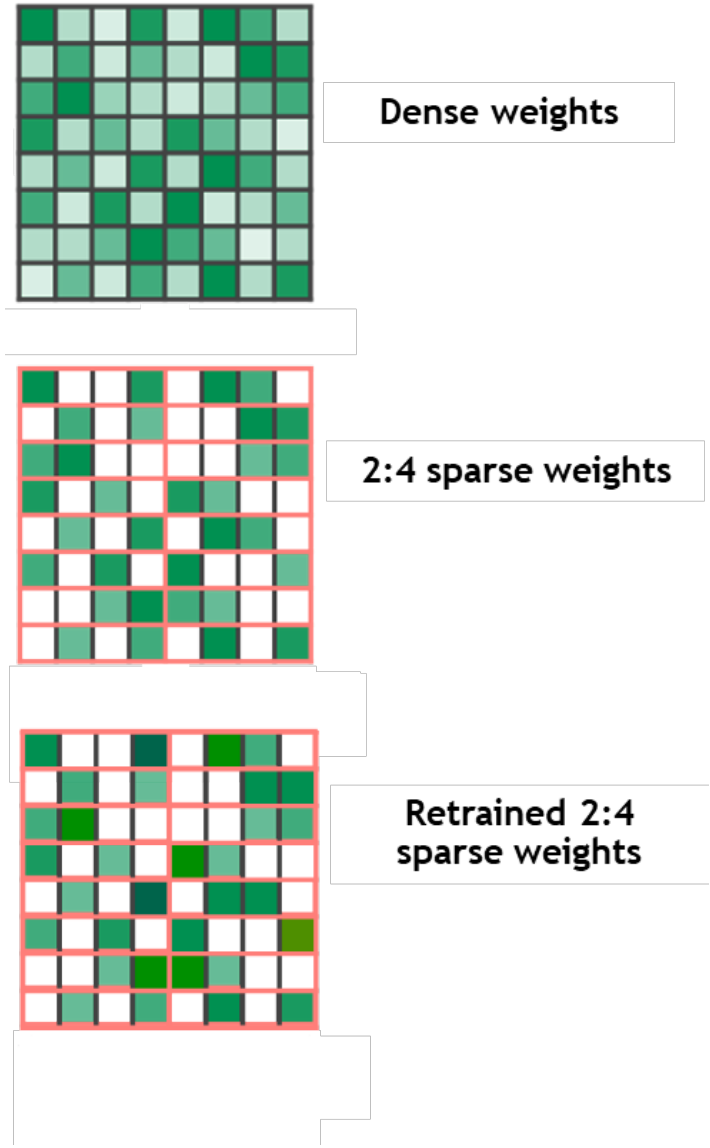
1. **Maintain accuracy**
2. **Apply to various tasks, network architectures, and optimizers**
3. **Do not require hyperparameter searches**

We're sharing a simple recipe we've found that satisfies these criteria – others exist!

***Accelerating Sparse Deep Neural Networks*, Mishra et al., https://arxiv.org/abs/2104.08378**

Maintaining Accuracy – Our Recipe

1. Train (or obtain) a dense network



2. Prune for 2:4 sparsity

- By magnitude, set 2 out of every 4 weights to zero
- Each group of 4 weights is independent
- Store a mask of which values were set to zero

3. Repeat the original training procedure

- Same hyperparameters as step (1)
- Initialize weights from step (2)
- Maintain the 0 pattern from step (2), no need to recompute the mask

ASP: Automatic SParsity (PyTorch)

- Part of the APEX library
- Three lines of code add 2:4 sparsity to your final network

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

ASP.prune_trained_model(model, optimizer)

x, y = DataLoader(…) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

ASP: https://github.com/NVIDIA/apex/tree/master/apex/contrib/sparsity

Maintaining Accuracy – Our Recipe's Results (ILSVRC2012, top1)

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
ResNet-34	73.7	73.9	0.2	73.7	-
ResNet-50	76.6	76.8	0.2	76.8	0.2
ResNet-101	77.7	78.0	0.3	77.9	-
ResNeXt-50-32x4d	77.6	77.7	0.1	77.7	-
ResNeXt-101-32x16d	79.7	79.9	0.2	79.9	0.2
DenseNet-121	75.5	75.3	-0.2	75.3	-0.2
DenseNet-161	78.8	78.8	-	78.9	0.1
Wide ResNet-50	78.5	78.6	0.1	78.5	-
Wide ResNet-101	78.9	79.2	0.3	79.1	0.2
Inception v3	77.1	77.1	-	77.1	-
Xception	79.2	79.2	-	79.2	-
VGG-16	74.0	74.1	0.1	74.1	0.1
VGG-19	75.0	75.0	-	75.0	-

***Accelerating Sparse Deep Neural Networks*, Mishra et al., https://arxiv.org/abs/2104.08378**

Maintaining Accuracy – Our Recipe's Results (COCO 2017, bbox AP)

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
MaskRCNN-RN50	37.9	37.9	-	37.8	-0.1
SSD-RN50	24.8	24.8	-	24.9	0.1
FasterRCNN-RN50-FPN-1x	37.6	38.6	1.0	38.4	0.8
FasterRCNN-RN50-FPN-3x	39.8	39.9	-0.1	39.4	-0.4
FasterRCNN-RN101-FPN-3x	41.9	42.0	0.1	41.8	-0.1
MaskRCNN-RN50-FPN-1x	39.9	40.3	0.4	40.0	0.1
MaskRCNN-RN50-FPN-3x	40.6	40.7	0.1	40.4	-0.2
MaskRCNN-RN101-FPN-3x	42.9	43.2	0.3	42.8	-0.1
RetinaNet-RN50-FPN-1x	36.4	37.4	1.0	37.2	0.8
RPN-RN50-FPN-1x	45.8	45.6	-0.2	45.5	-0.3

Many more in our whitepaper...

- Generative tasks (GANs)
- Translation
- Language modeling

***Accelerating Sparse Deep Neural Networks*, Mishra et al., https://arxiv.org/abs/2104.08378**

Outline

1. 2:4 Structured Sparsity – Why and How
2. Performance
3. Maintaining Accuracy
4. Channel Permutations
    1. Problem/Solution
    2. Applying Permutations
    3. Finding Permutations
    4. Results

Small, parameter-efficient models can lose accuracy after pruning

ILSVRC'12	Default 2:4	
Network	Δ Top 1	Δ Top 5
MobileNet v2	-1.98	-1.13
MobileNet v3 (Small)	-2.73	-1.67
SqueezeNet v1.0	-4.01	-2.96
SqueezeNet v1.1	-1.25	-0.93
ShuffleNet v2	-1.35	-0.87
EfficientNet B0	-1.27	-0.52

Channel Permutations for N:M Sparsity, Pool and Yu, NeurIPS 2021

Column (channel) permutations allow better pruning choices

A	B	C	D	E	F	G	H
0.4	1.1	1.5	6.9	0.8	6.7	5.4	8.8
1.5	0.6	1.3	0.5	8.8	4.2	8.9	8.9

Original

Prune →

A	B	C	D	E	F	G	H
0	0	1.5	6.9	0	6.7	0	8.8
1.5	0	1.3	0	0	0	8.9	8.9

2:4 Sparse

Total Magnitude: 44.5

Permute Columns

E	G	A	D	C	H	B	F
0.8	5.4	0.4	6.9	1.5	8.8	1.1	6.7
8.8	8.9	1.5	0.5	1.3	8.9	0.6	4.2

Permuted

Prune →

E	G	A	D	C	H	B	F
0	5.4	0	6.9	0	8.8	0	6.7
8.8	8.9	0	0	0	8.9	0	4.2

2:4 Sparse (Permuted)

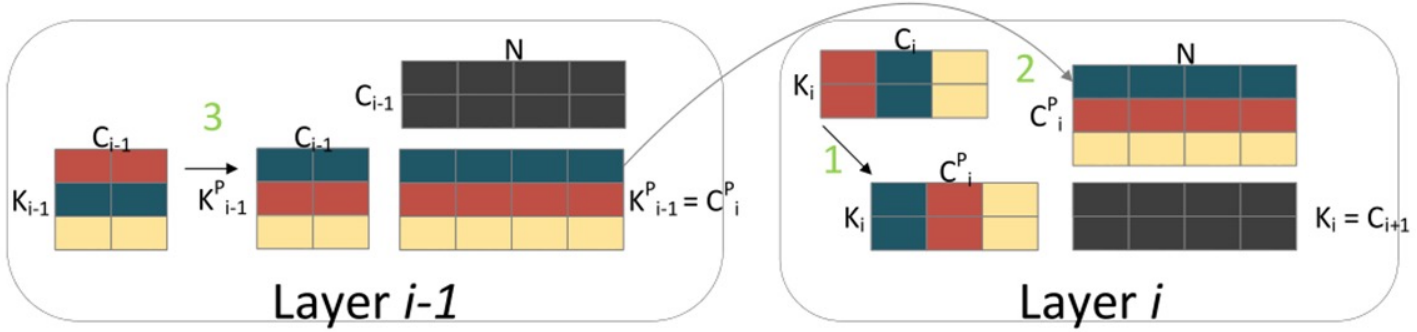
Total Magnitude: 58.6

Channel Permutations for N:M Sparsity, Pool and Yu, NeurIPS 2021

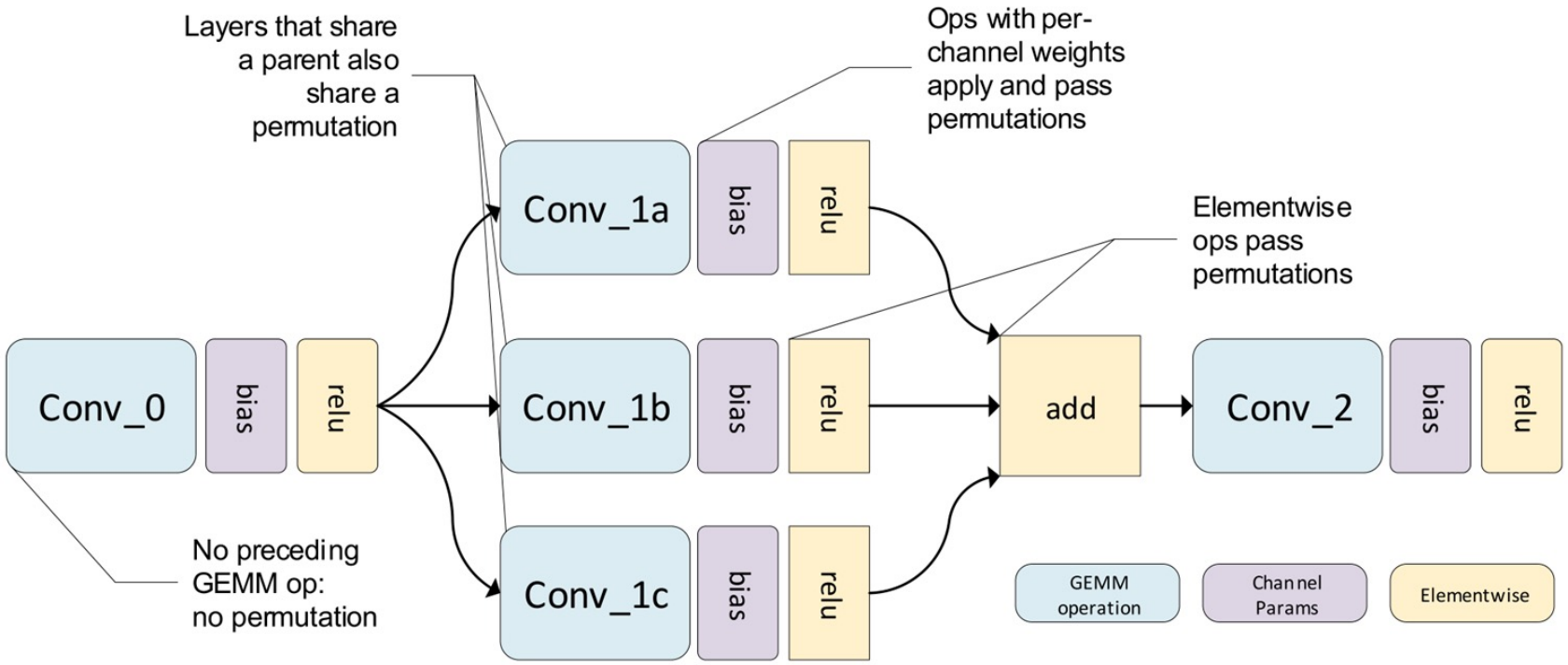
C is part of the inner dimension of a GEMM, so permuting that dimension of both operands does not change the result

- Permuting the *C* dimension of the **weights** is done offline
- Permuting the *C* dimension of layer *i*'s **activations** can be done by permuting the *K* dimension (output channels) of layer *i*-1, which is accomplished by permuting the weights' rows (offline)

1. Maintain magnitude with a desired weight permutation
2. Match the new layout in the corresponding activations
3. Cause the permutation in (2)



Applying permutations to a whole network requires care



Channel Permutations – Finding a Good Permutation

There are many, many unique permutations with N:M sparsity for even modest channel counts.

We adapted and improved past work[1], resulting in a heuristic approach that performs well:

- Gives good quality permutations (minimizes lost weight magnitude)
- Converges quickly in practice
- Scalable: trade off solution quality for search time

[1] *TETRIS: TilE-matching the TRemendous Irregular Sparsity*, Ji et al., NeurIPS 2018

Some terminology:

- Stripe: vertical slice of a matrix consisting of one M-wide group (4 values in 2:4 sparsity)
- Stripe Group: a group of stripes (e.g., {0,1}, {0,3}, {2,3,6}, etc.) concatenated together

Stripe 0 | Stripe 1 | ... Stripe 3

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	...	*P*
0.4	1.1	1.5	6.9	0.8	6.7	5.4	8.8		3.2
1.5	0.6	1.3	0.5	8.8	4.2	8.9	8.9		4.4

We developed a greedy, divide-and-conquer brute-force heuristic:

- **Brute-force**: a stripe group's best permutation is found exhaustively

  2-stripe groups = 8 columns = 35 unique permutations
  3-stripe groups = 12 columns = 5,775 unique permutations
  4-stripe groups = 16 columns = 2,627,625 unique permutations

We developed a greedy, divide-and-conquer brute-force heuristic:

- **Brute-force**: a stripe group's best permutation is found exhaustively
- **Divide-and-conquer**: all potential stripe groups are brute-forced independently

  Improvement in post-pruning magnitude is collected for all groups:
  {0,1} = +1.4
  {0,2} = +0.3
  {1,2} = +1.5

We developed a greedy, divide-and-conquer brute-force heuristic:

- **Brute-force**: a stripe group's best permutation is found exhaustively
- **Divide-and-conquer**: all potential stripe groups are brute-forced independently
- **Greedy**: the stripe group with the largest improvement is selected

  That group's permutation is applied to that group
  Columns outside of the winning group are unchanged

We developed a greedy, divide-and-conquer brute-force heuristic:

- **Brute-force**: a stripe group's best permutation is found exhaustively
- **Divide-and-conquer**: all potential stripe groups are brute-forced independently
- **Greedy**: the stripe group with the largest improvement is selected

The greedy step changes the columns that constitute many groups, so we iterate:

1. Generate all stripe groups
2. Divide-and-conquer the groups
3. If there is a stripe group with a nonzero improvement:
    1. Take the greedy step with the winner
    2. GOTO(1)
4. Converged!

This heuristic converges, but it's probably to a local optimum.

Bounded regressions can find a better solution:

- After convergence, choose two columns at random and swap them
- This permutation *will be worse* - but that's okay!
- One iteration of the greedy heuristic will either:
  - Find the previous solution and swap the columns back
  - Find a permutation in a stripe group with a better improvement, and repeat until converging at a better solution

We have two knobs to trade solution quality with search time:

1. The size of the stripe groups to optimize (default: 2 groups = 8 columns)
2. The number of bounded regression attempts to allow (default: 100)

Channel Permutations – Finding a Good Permutation: Results

Efficacy: quality of the found permutation

Optimal Found: how many times the strategy found the optimal permutation

Channel Permutations – Finding a Good Permutation: Results

Efficacy: quality of the found permutation

Optimal Found: how many times the strategy found the optimal permutation

Channel Permutations – Finding a Good Permutation: Results

Efficacy: quality of the found permutation

Optimal Found: how many times the strategy found the optimal permutation

Channel Permutations – Finding a Good Permutation: Results

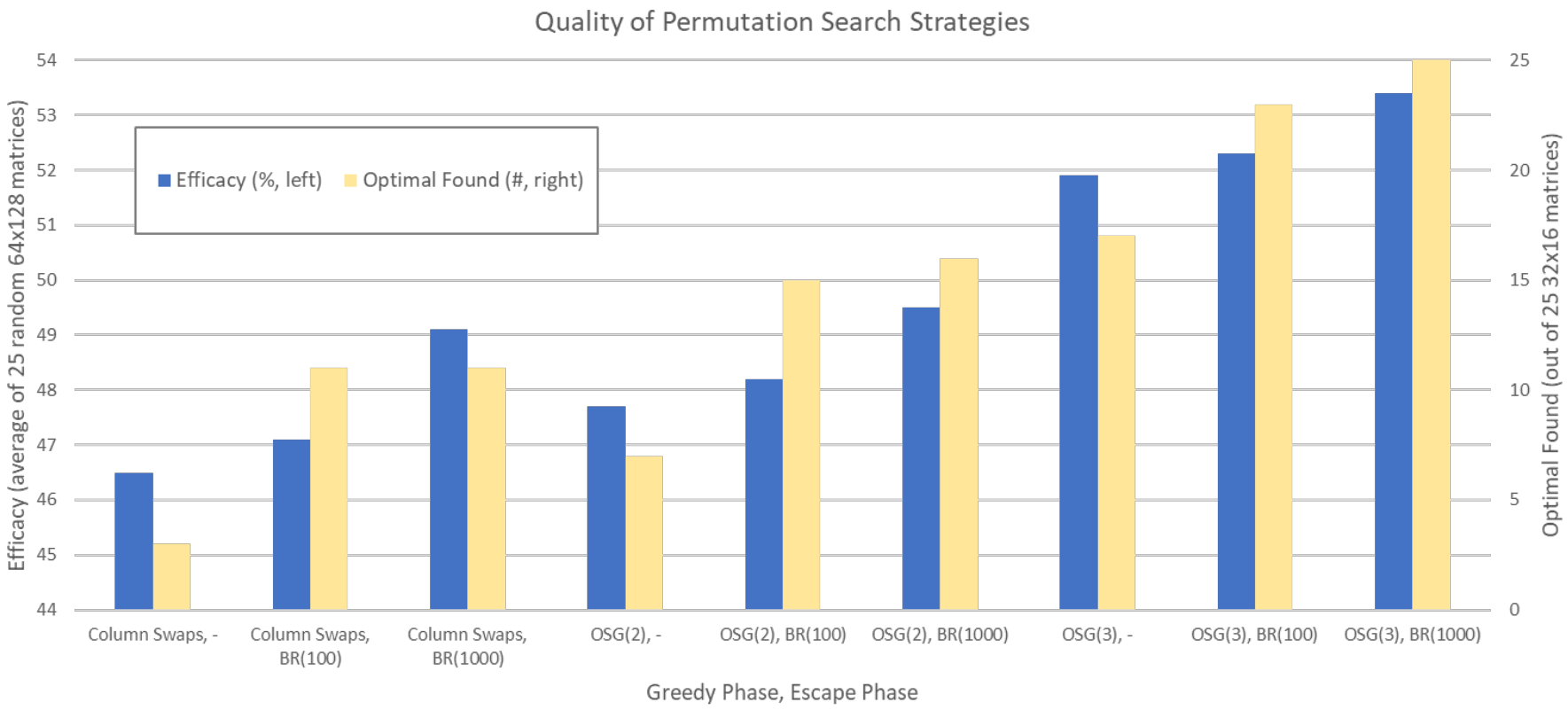
Efficacy: quality of the found permutation

Optimal Found: how many times the strategy found the optimal permutation

Channel Permutations – Finding a Good Permutation: Results

Efficacy: quality of the found permutation

Optimal Found: how many times the strategy found the optimal permutation



Increase escape attempts **and** the size of the stripe groups for even better solutions.

Channel Permutations – Finding a Good Permutation: Network Results

	Baseline (ILSVRC2012)		Default 2:4		Permuted 2:4 (ours)	
Network	Top 1	Top 5	Δ Top 1	Δ Top 5	Δ Top 1	Δ Top 5
MobileNet v2	71.55	90.28	-1.98	-1.13	0.01	0.02
MobileNet v3 (Small)	67.67	87.40	-2.73	-1.67	0.10	0.15
MobileNet v3 (Large)	74.04	91.34	-0.91	-0.40	0.10	0.04
SqueezeNet v1.0	58.09	80.42	-4.01	-2.96	0.60	0.54
SqueezeNet v1.1	58.21	80.62	-1.25	-0.93	0.03	0.05
MNASNet 1.0	73.24	91.36	-1.25	-0.58	0.02	0.00
ShuffleNet v2	68.32	88.36	-1.35	-0.87	0.10	0.01
EfficientNet B0	77.25	93.59	-1.27	-0.52	0.04	0.07
ResNet-50	76.16	92.88	0.05	0.13	0.13	0.26
ResNeXt-50	77.62	93.70	0.01	0.05	0.13	0.07
DenseNet-161	77.14	93.56	0.82	0.41	0.92	0.52

Channel Permutations for N:M Sparsity, Pool and Yu, NeurIPS 2021

Part 2: 2:4 (N:M) Sparsity

Rakesh Nagi

Department of Industrial and Enterprise Systems Engineering

University of Illinois at Urbana-Champaign

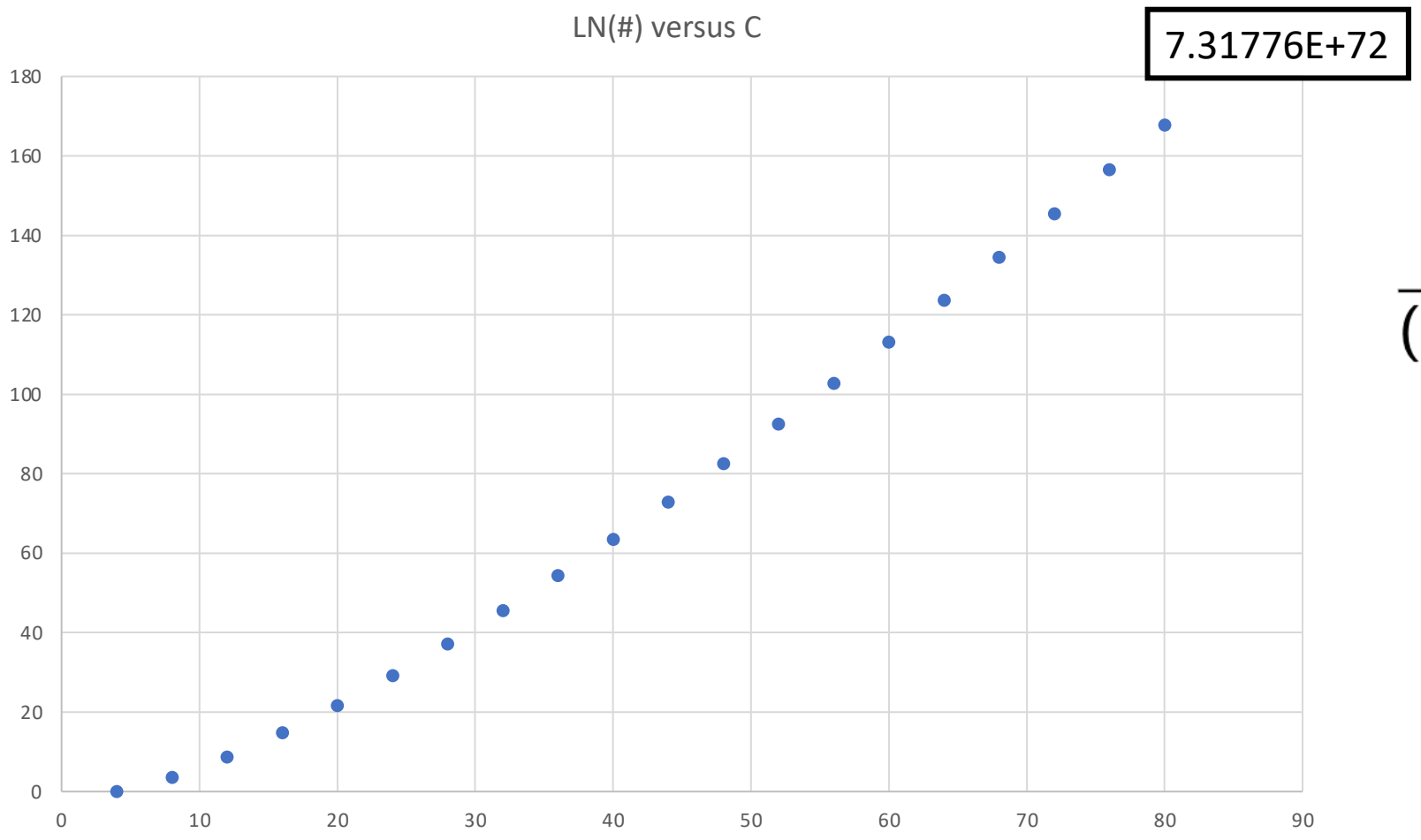
Channel Permutations: Preliminaries of Partitioning C elements into groups of M

- Sequence Approach
  - Form a sequence of C elements = $C!$
  - But the order of these partitions does not matter (divide by $(C/M)!$)
  - Order within each partition does not matter (divide by $(M!)^{(C/M)}$)

$$\boxed{\frac{C!}{(M!)^{C/M}\left(\frac{C}{M}\right)!}}$$

- Partition Approach
  - For first partition choose M from C = $\binom{C}{M} = \frac{C(C-1)(C-2)....(C-M+1)}{M!}$
  - For 2nd partition choose M from C-M = $\binom{C-M}{M} = \frac{(C-M)(C-M-1)(C-M-2)...(C-2M+1)}{M!}$
  - But different orderings of (C/M) parts don't matter

Preliminaries: Growth as a function of problem size



$$\frac{C!}{(M!)^{C/M}(\frac{C}{M})!}$$

Let's review performance of various mathematical programming optimization approaches.

Binary Quadratic Program (BQP) - Naïve Version

- Indexes: i for column; j for row; k for partition (k=C/4)
- Constant: $A_{ij} \geq 0$ Matrix (weight) value in column i and row j
- Binary variables:
  - $x_{ik} \in \{0,1\}$ 1 if column i is assigned to partition k; 0 otherwise.
  - $y_{ij} \in \{0,1\}$ 1 if entry column i and row j is selected for deletion; 0 otherwise.
- Constraints:
  - $\sum_k x_{ik} = 1 \; \forall i$ (every column is assigned to exactly one partition)
  - $\sum_i x_{ik} = 4 \; \forall k$ (every partition has exactly 4 columns)
  - $\sum_i x_{ik} y_{ij} = 2 \; \forall j, k$ (delete 2 entries from every partition) **<= Quadratic!**
- Objective Function:
  - Min $\sum_{ij} A_{ij} y_{ij}$ (Minimize smallest entries to obtain 2: 4 sparsity)

Number of Binary Variables C^2/M and C x R (# Rows)

Binary Linaear Program (BLP)

Number of Binary Variables
C^2/M
and
Continuous Variables
$C^2/M \times R$

- Indexes: i for column; j for row; k for partition
- Constant: $A_{ij} \geq 0$ Matrix (weight) value in column i and row j
- Binary variables:
  - $x_{ik} \in \{0,1\}$ 1 if column i is assigned to partition k; 0 otherwise.
  - $1 \geq z_{ijk} \geq 0$; 1 if entry column i and row j in partition k is selected for deletion; 0 o/w.
- Constraints:
  - $\sum_k x_{ik} = 1 \; \forall i$ (every column is assigned to exactly one partition)
  - $\sum_i x_{ik} = 4 \; \forall k$ (every partition has exactly 4 columns)
  - $z_{ijk} \leq x_{ik} \; \forall i, j, k$ (can only delete entry if column i is in partition k)
  - $\sum_i z_{ijk} = 2 \; \forall j, k$ (delete 2 entries from every partition)
- Objective Function:
  - Min $\sum_{ijk} A_{ij} z_{ijk}$ (Minimize smallest entries to obtain 2: 4 sparsity)

Further Analysis: 4-Dimensional Assignment Formulation

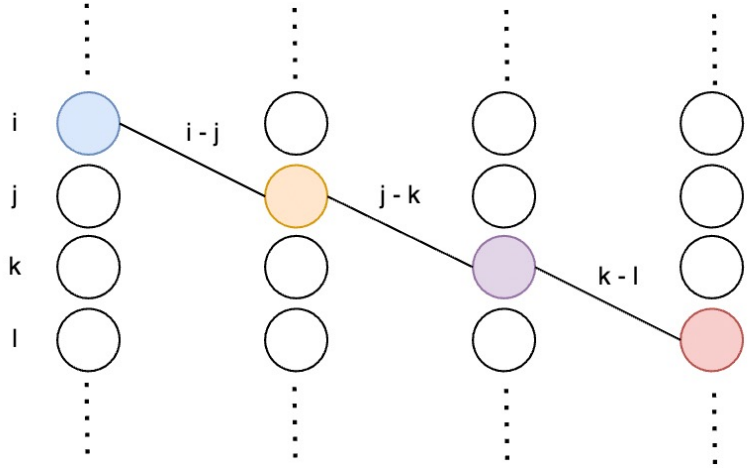
- Symmetry in BLP and BQP:

	k = 1				k = 2			
Solution 1	1	2	3	4	5	6	7	8
Solution 2	5	6	7	8	1	2	3	4

- 4-Dimensional Network Structure:
  - C × 4 nodes
  - Nodes represent columns,

  edges represent columns of same partition

  $Z_{ijkl} \in \{0, 1\}$, represents the 4D assignment



$(i\text{-}j, j\text{-}k, k\text{-}l) \rightarrow (i,j,k,l)$

$$\min \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{N} C_{ijkl} Z_{ijkl}$$

$$\text{s.t.} \sum_{j=1}^{N} X_{ij}^{s} \leq 1 \quad \forall i \in \{1, 2, \cdots, N\} \; \forall s \in \{1, 2, 3\}$$

$$\sum_{i=1}^{N} X_{ij}^{s} \leq 1 \quad \forall j \in \{1, 2, \cdots, N\} \; \forall s \in \{1, 2, 3\}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N} X_{ij}^{s} = \frac{N}{4} \quad \forall s \in \{1, 2, 3\}$$

$$\sum_{j=1}^{N}\sum_{s=1}^{3} X_{ij}^{s} = 1 \quad \forall i \in \{1, 2, \cdots, N\}$$

$$\sum_{k=1}^{N}\sum_{l=1}^{N} Z_{ijkl} = X_{ij}^{1} \quad \forall i \in \{1, 2, \cdots, N\} \; \forall j \in \{1, 2, \cdots, N\}$$

$$\sum_{l=1}^{N}\sum_{i=1}^{N} Z_{ijkl} = X_{jk}^{2} \quad \forall j \in \{1, 2, \cdots, N\} \; \forall k \in \{1, 2, \cdots, N\}$$

$$\sum_{j=1}^{N}\sum_{k=1}^{N} Z_{ijkl} = X_{kl}^{3} \quad \forall k \in \{1, 2, \cdots, N\} \; \forall l \in \{1, 2, \cdots, N\}$$

$$X_{ik} \in \{0, 1\}, \quad 0 \leq Z_{ijkl} \leq 1$$

4th Formulation: SET-COVERING

- Given, $Q = \{q : q \subset \{1, 2, 3, \cdots, N\}, |q| = 4\}$
- For some $q \in Q : q \rightarrow (i, j, k, l)$ define deletion cost of partition $D_{ijkl} = D_q$
- Define a constant matrix H:

$$H_{iq} = \begin{cases} 1 & \text{if column-}i \in q \\ 0 & \text{otherwise} \end{cases}$$

- $X_q \in \{0, 1\}$ represents the selection of q among the optimal column partitions.

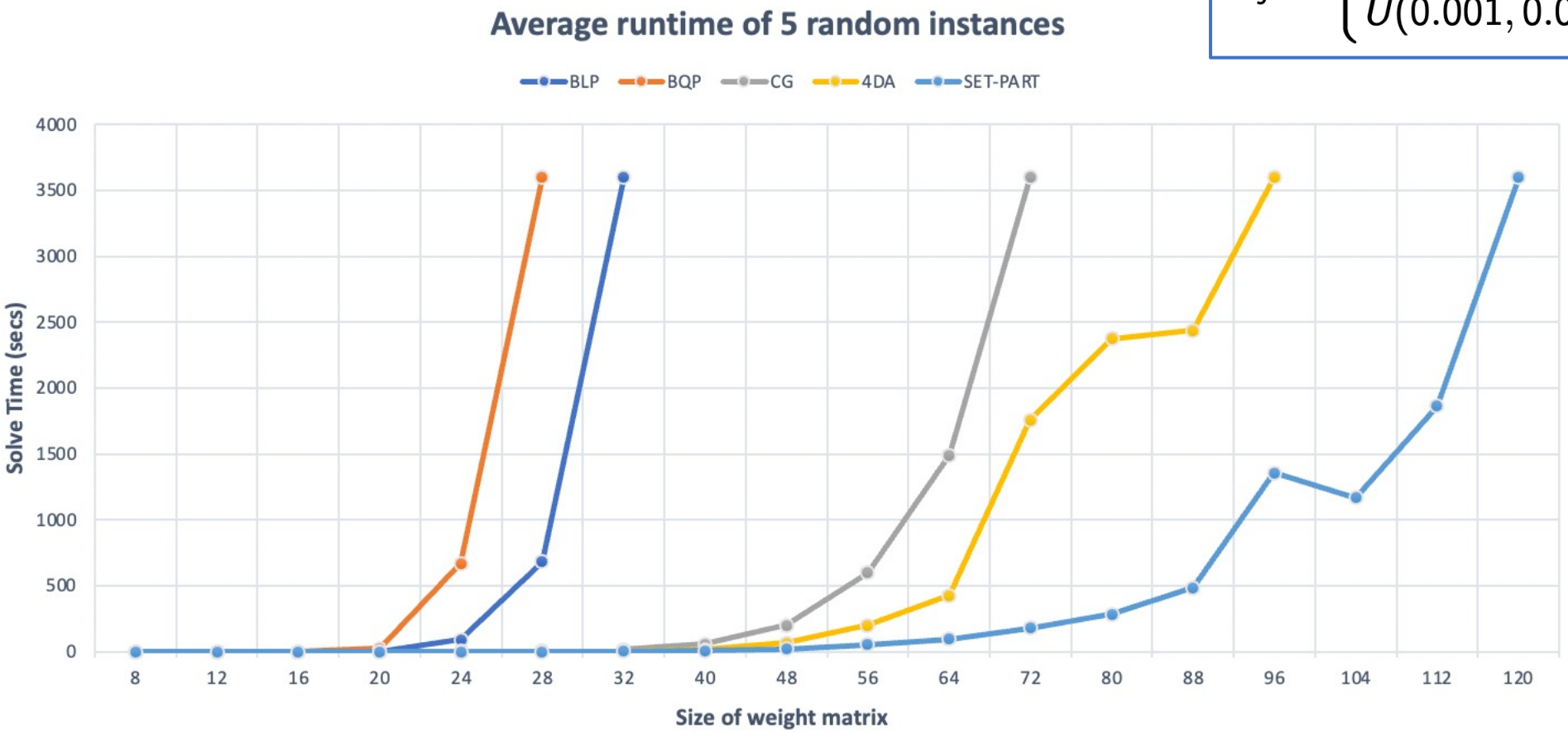
$$\begin{aligned} \min \quad & \sum_{q \in Q} D_q X_q \\ \text{s.t.} \quad & \sum_{q \in Q} H_{iq} X_q = 1 \quad \forall i \in \{1, 2, \cdots, N\} \\ & X_q \in \{0, 1\} \end{aligned}$$

- Note: We need $O(N^4)$ cost terms. Set-Covering is NP-hard.

Comparison of Math Programming Formulations

Random Matrix A (with p=0.5)

$$A_{ij} = \begin{cases} U(0.1, 1) & \text{with probability: } p \\ U(0.001, 0.01) & \text{with probability: } 1-p \end{cases}$$

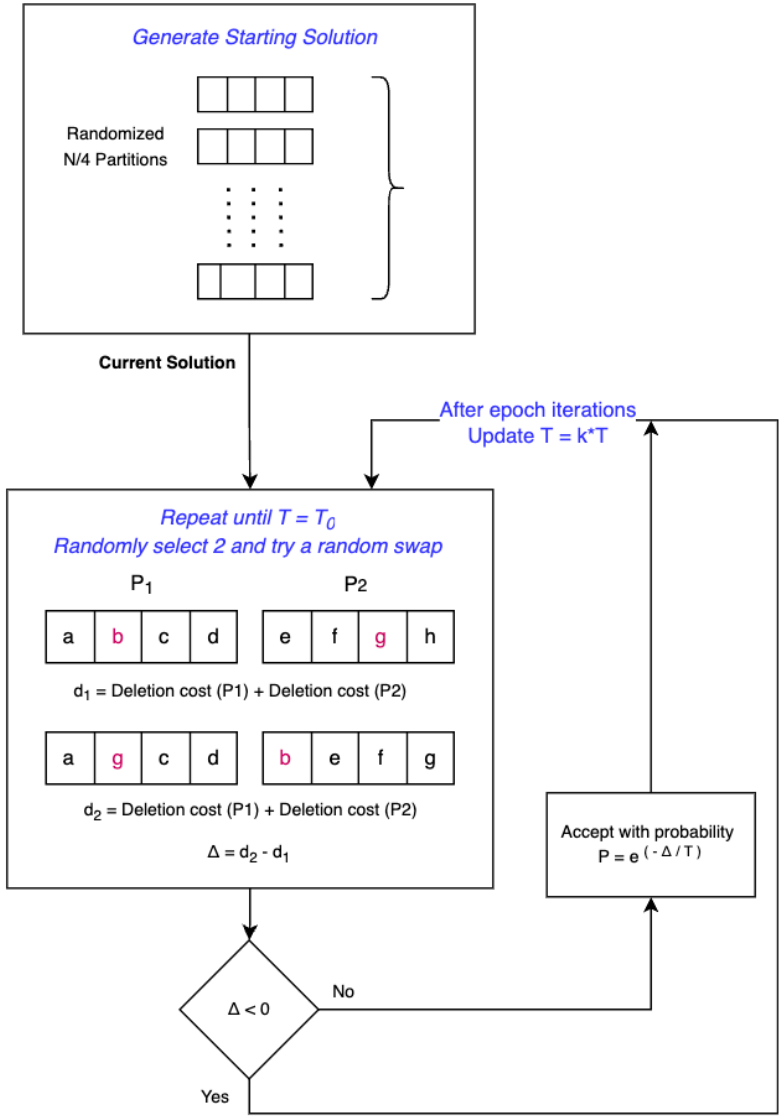


- Timeout=1hr
- Gurobi 9.1.2 solver, Intel Core i7-4790 CPU, 16 GB RAM
- BQP and BLP were timed-out beyond matrix size of 24
- 4DA, SET-COVER and Colum Generation (CG) perform better, but timeout at 120

- Note: We need $O(N^4)$ cost terms. Set-Covering is NP-hard.

Scalable Heuristics: Simulated Annealing

- Given the Optimal N:M Channel Sparsity problem is NP-hard, we need scalable heuristics for larger problem instances.
- Recall hill-climbing heuristics discussed earlier.
- Simulated Annealing is a probabilistic method for achieving near global optimality.
- (We are currently testing this approach.)
- Other metaheuristics can also be applied.



Future Work and Closing Thoughts

Organizers

Wen-mei Hwu, Jinjun Xiong, Mert Hidayetoğlu, Rakesh Nagi,

Jeff Pool, Sitao Huang, Vikram Mailthody

Tutorial | MLSys 2022 | Aug 30, 2022

https://mlsys.org/virtual/2022/tutorial/2199

Slack for discussions: https://bit.ly/3KLQEUN

Summary

1. Opening remarks and overview of sparsity in ML
    1. EMOGI, PyTorch-Direct are available in DGLv0.8+ with UnifiedTensor representation
    2. https://github.com/illinois-impact/EMOGI
2. Tiled SpMM and its performance model on GPUs
    1. PyTorch binding is available on request. Code:

       https://github.com/merthidayetoglu/SpDNN_Challenge2020
3. Sparse deep neural network inference on FPGAs
4. 2:4 Sparsity on GPU Tensor Cores
5. Future work and closing remarks

Existing Models Larger Than CPU Memory

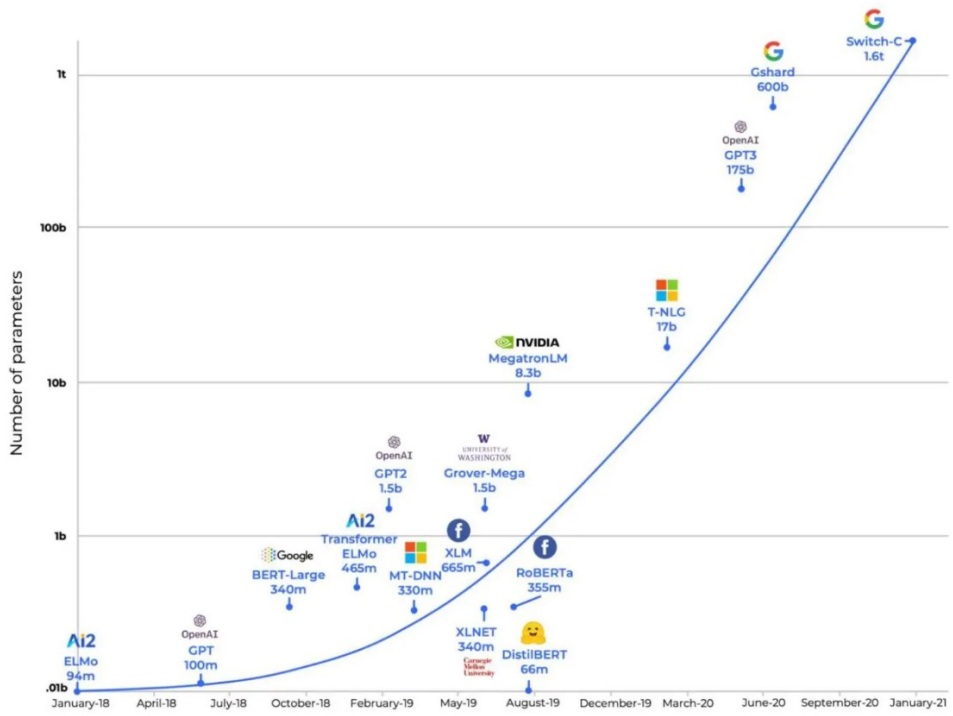
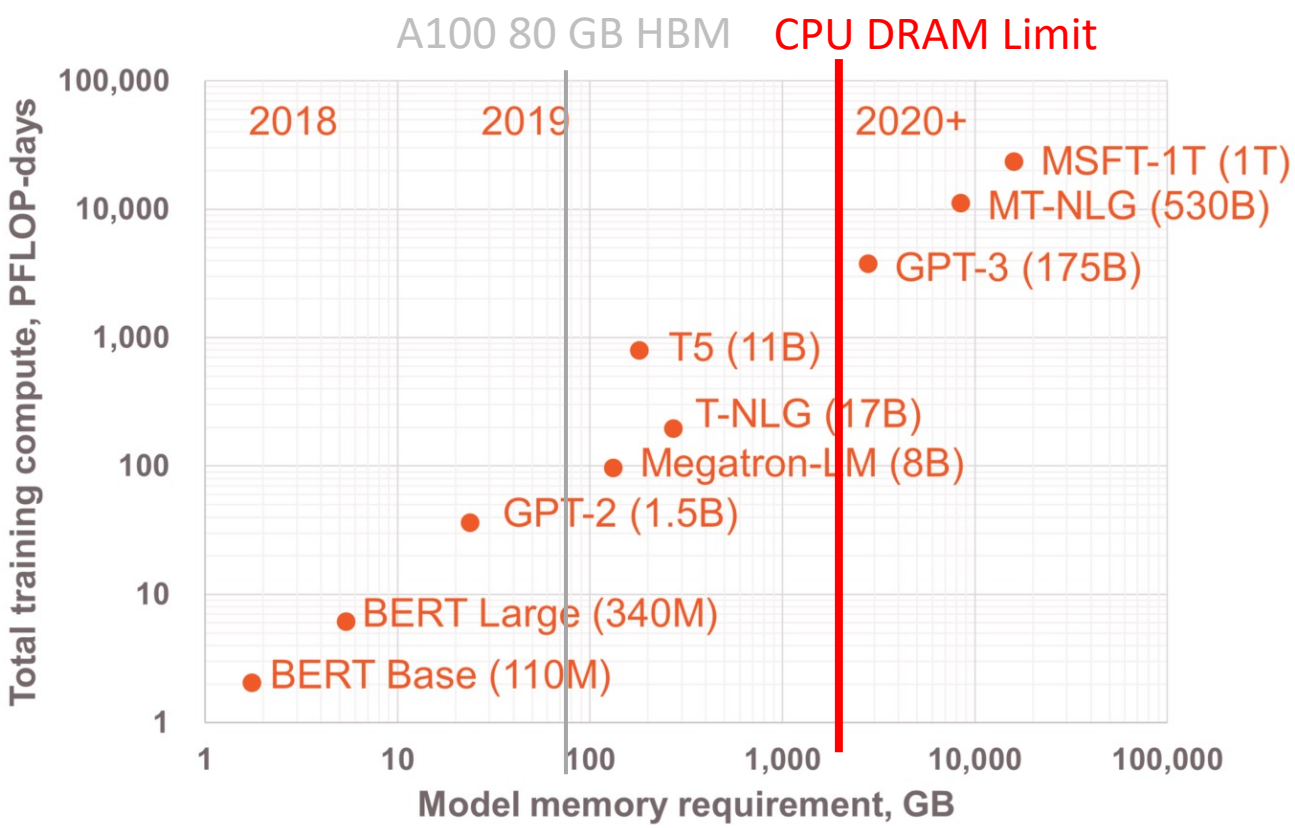


Fig: *Exponential growth of number of parameters in deep learning models*

Source: https://deci.ai/blog/sota-dnns-overview/



Source: Cerebras Systems, HotChips 2022

Lower compute utilization with scale-out!

Next Challenges and Opportunities

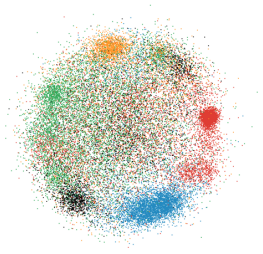
1. Very large data objects with sparse access patterns
   - What if the data object does not fit into the host memory of one compute node?
   - E.g., efficient communication for SpMM across **multiple compute nodes**
   - E.g., efficient access to node feature tensors that has to be accessed from **storage**

- Training spare models rather than sparsifying dense models to enable training of models that are orders of magnitude larger than what we can train today

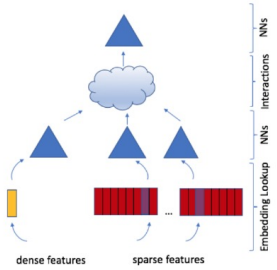
- What else can we do to the compute devices and memory interfaces to support sparsity much better?

Emerging Application Characteristics

Application Characteristics



Graphs



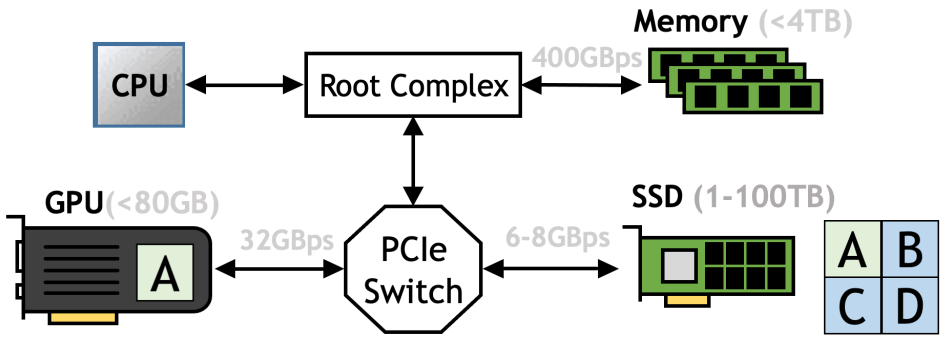
Recommender Systems

	Name	Team	Number	Position	Age	Height	Weight	College
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN
5	Amir Johnson	Boston Celtics	90.0	PF	29.0	6-9	240.0	NaN
6	Jordan Mickey	Boston Celtics	55.0	PF	21.0	6-8	235.0	LSU
7	Kelly Olynyk	Boston Celtics	41.0	C	25.0	7-0	238.0	Gonzaga
8	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville
9	Marcus Smart	Boston Celtics	36.0	PG	22.0	6-4	220.0	Oklahoma State
10	Jared Sullinger	Boston Celtics	7.0	C	24.0	6-9	260.0	Ohio State
11	Isaiah Thomas	Boston Celtics	4.0	PG	27.0	5-9	185.0	Washington
12	Evan Turner	Boston Celtics	11.0	SG	27.0	6-7	220.0	Ohio State
13	James Young	Boston Celtics	13.0	SG	20.0	6-6	215.0	Kentucky
14	Tyler Zeller	Boston Celtics	44.0	C	26.0	7-0	253.0	North Carolina

Data Analytics

- Massive data structures
- Fine-grain data dependent access
- Sparse accesses

Problems With CPU-Centric Model



- Complex tiling strategies
- I/O and memory management overhead
- Gross I/O amplification

Summary

Interested in collaborating with us, please reach out to us in slack:

https://bit.ly/3KLQEUN

Email: vsm2@Illinois.edu or hidayet2@illinois.edu

Thank you!