NN Performance optimization:
How to achieve more with less cost

Software perspective

Pavlo Molchanov, pmolchanov@nvidia.com

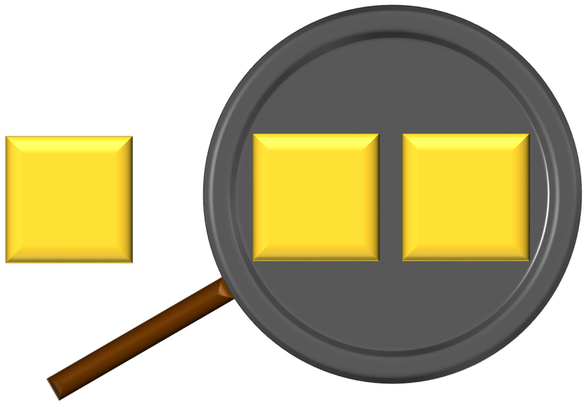
Distinguished Research Scientist, Manager

Disclaimer: Results, numbers and performance are reported from the research perspective.
For the exact performance please contact NVIDIA product managers.

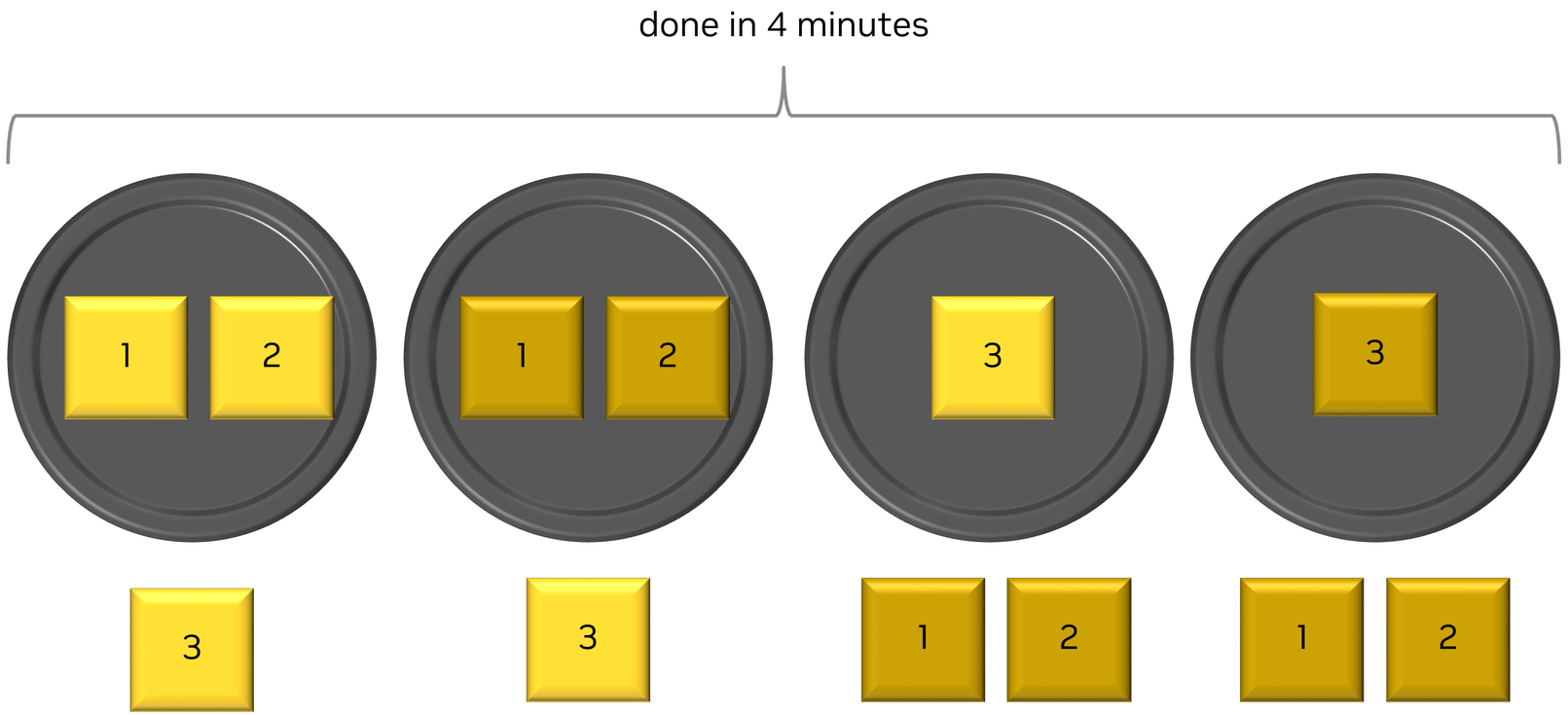
Slides credit:
Giuseppe Fiameni gfiameni@nvidia.com
https://docs.nvidia.com/deeplearning/performance/dl-performance-getting-started/index.html

Making toast fast

https://www.youtube.com/watch?v=gVPK81rI390



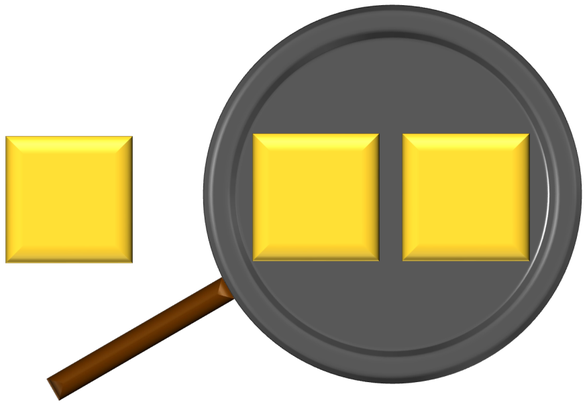
How long does it take to
fry 3 toast with a pan
if frying one side of
a toast takes 1 min?



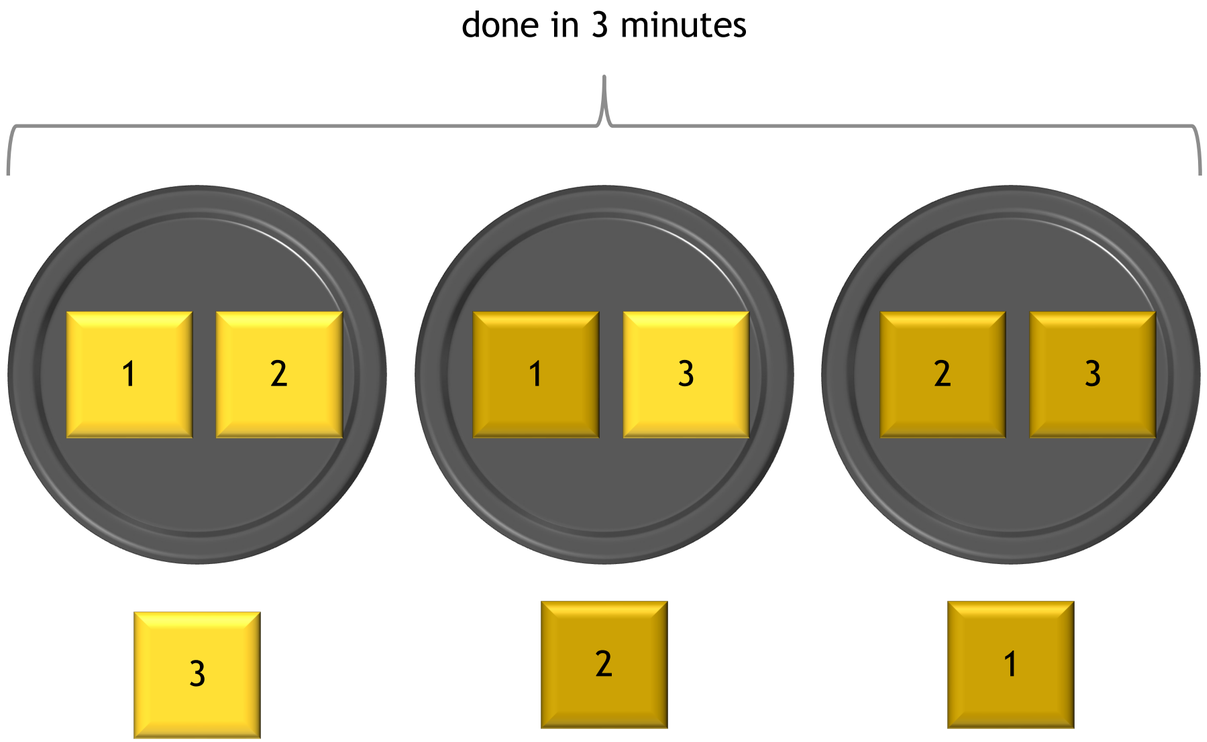
Pan – GPUs
Toast – layers or data

Making toast fast

https://www.youtube.com/watch?v=gVPK81rI390

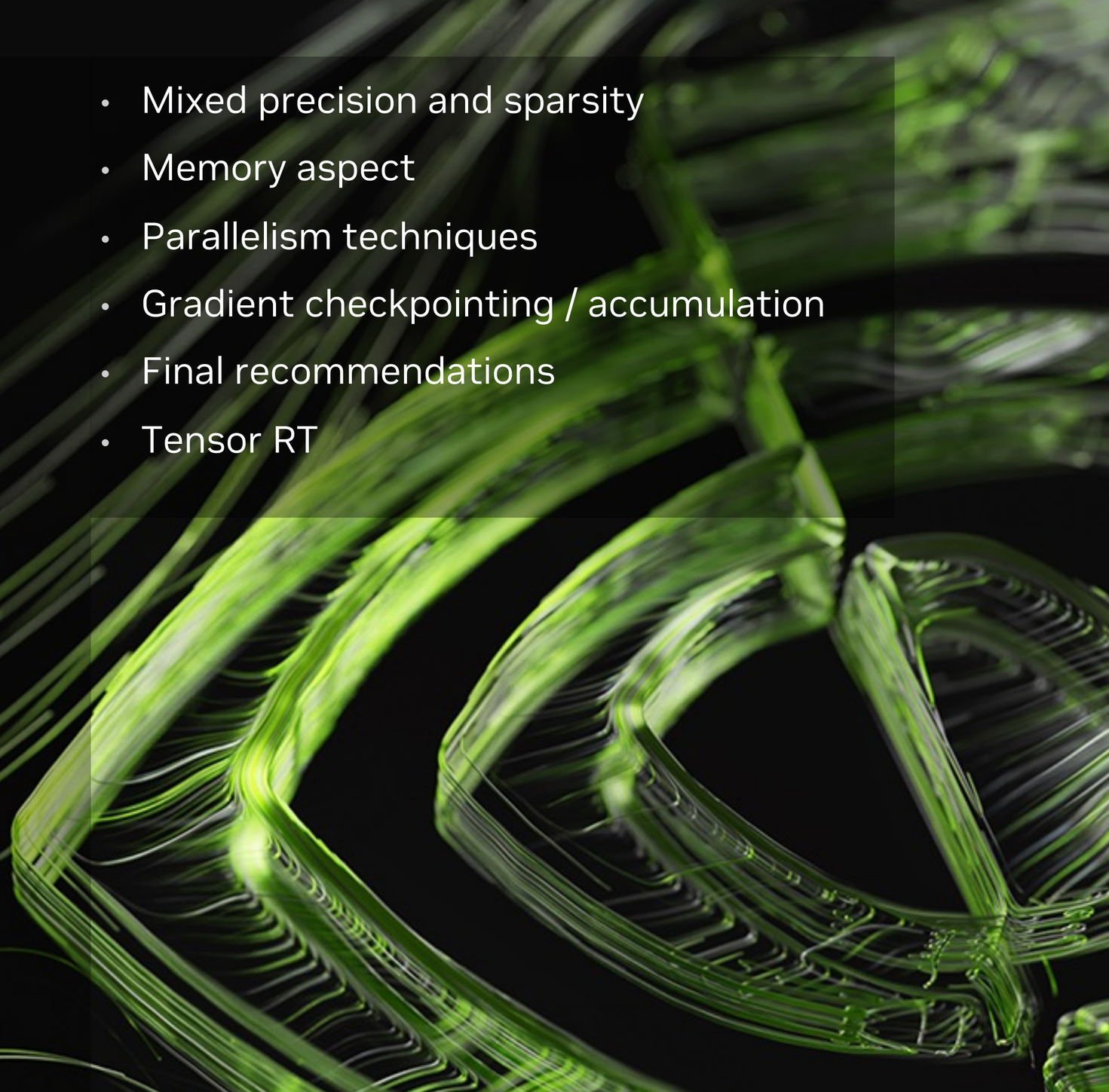


How long does it take to
fry 3 toast with a pan
if frying one side of
a toast takes 1 min?



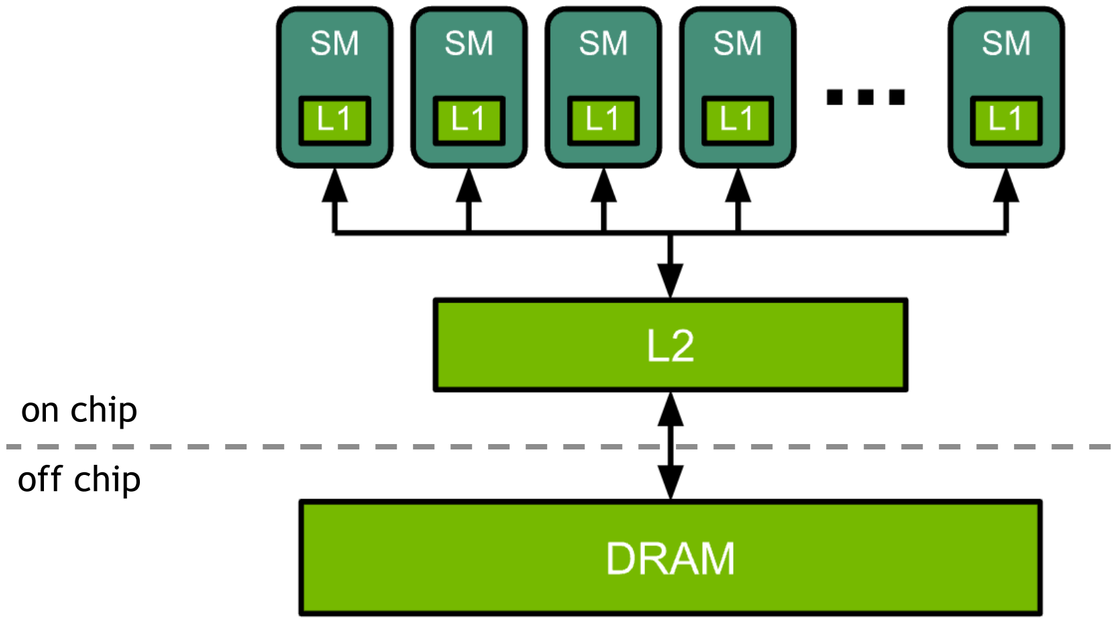
Pan – GPUs
Toast – layers or data

- **Background:**
  - GEMM
  - Math and memory bounds
  - GPU implementation
  - Tile and wave quantization
- Understanding DNN performance:
  - DNN Operation Categories
  - Transformer architecture example
  - What is the limit? Guide
- Recommendations I:
  - Hierarchical models and UNets
  - Tensor cores
  - Convolutions
  - Linear layers
  - Transformer
- Mixed precision and sparsity
- Memory aspect
- Parallelism techniques
- Gradient checkpointing / accumulation
- Final recommendations
- Tensor RT



GPU architecture SIMPLIFIED

- Arithmetic and other instructions are executed by the Streaming Multiprocessors (SMs).
- Data and code are accessed from high-bandwidth DRAM via the on-chip L2 cache.
- Example - NVIDIA A100 GPU contains:
  - 108 SMs
  - 40 MB L2 cache
  - 80 GB of HBM2 memory with up to 2039 GB/s bandwidth



Matrix matrix multiplications

- GEMMs (General Matrix Multiplications) are a fundamental building block for many operations in neural networks:
  - fully-connected layers
  - recurrent layers such as RNNs, LSTMs or GRUs
  - convolutional layers
  - attention layers
- GEMM is defined as: $C = \alpha \mathbf{AB} + \beta C$
  - with A and B as matrix inputs, α and β as scalar inputs, and C as a pre-existing matrix which is overwritten by the output

Math and Memory bounds

- GEMM: $C = \alpha \mathbf{AB} + \beta C$
  - Matrix A is an M x K; matrices B is K x N; and C is M x N matrices
  - The product of A and B has M x N values. It requires a total of M * N * K fused multiply-adds (FMAs)
  - Each FMA is 2 operations, a multiply and an add, so a total of 2 * M * N * K FLOPS are required
  - α and β can be ignored if K is sufficiently large
- To understand performance:

$$\text{Arithmetic Intensity} = \frac{\text{number of FLOPS}}{\text{number of byte accesses}} = \frac{2 \cdot (M \cdot N \cdot K)}{2 \cdot (M \cdot K + N \cdot K + M \cdot N)} = \frac{M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}$$

  - For Tensor Cores in V100 - FLOPS:B ratio is 138.9
  - If MxNxK = 8192x**128**x8192, AI = 124.1 FLOPS/B – memory bound
  - If MxNxK = 8192x**8192**x8192, AI = 2730 FLOPS/B – math bounded

Multiply-add operations per clock

- For FLOPS – multiply by 2
- Example how to calculate peak dense throughput for A100 GPU:
  - 108 SMs
  - 1.41 GHz clock rate
  - 156 TF32 TFLOPS and 312 FP16 TFLOPS

Figure 2. Multiply-add operations per clock per SM

	CUDA Cores				Tensor Cores					
NVIDIA Architecture	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4	INT1
Volta	32	64	128	256			512			
Turing	2	64	128	256			512	1024	2048	8192
Ampere (A100)	32	64	256	256	64	512	1024	2048	4096	16384
Ampere, sparse						1024	2048	4096	8192	

GPU Implementation

- GEMMs is implemented by partitioning the output matrix into tiles, which are then assigned to thread blocks:

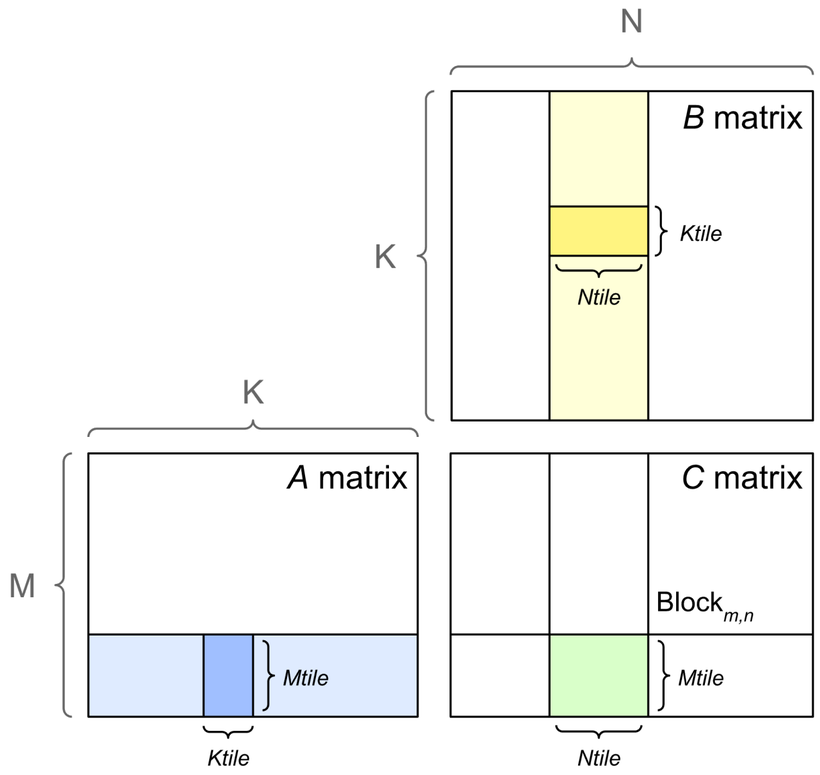


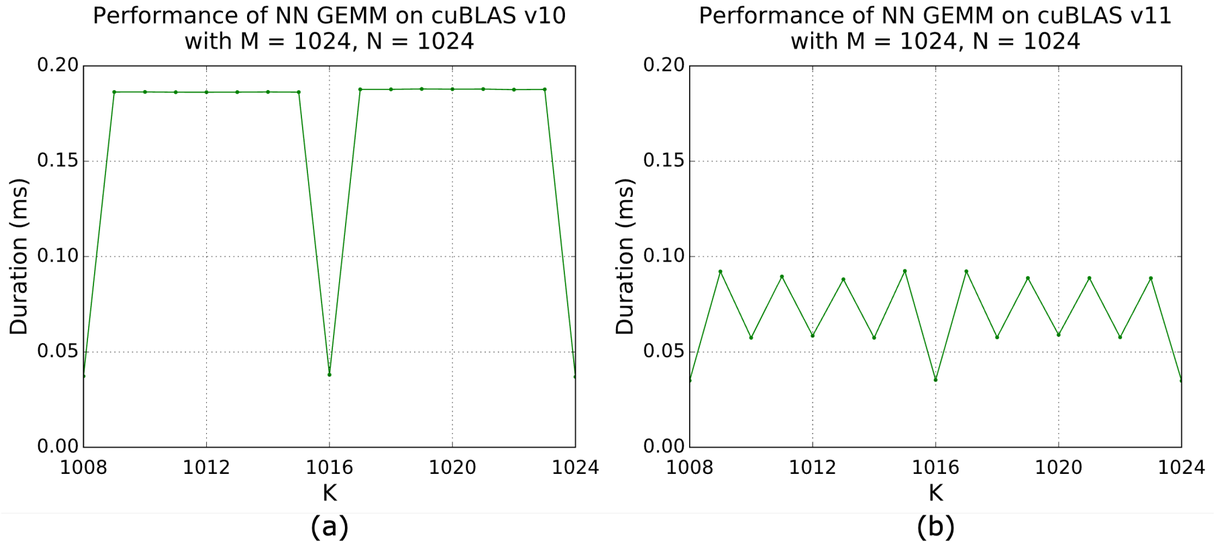
Table 1. Tensor Core requirements by cuBLAS or cuDNN version for some common data precisions. These requirements apply to matrix dimensions M, N, and K.

Tensor Cores can be used for...	cuBLAS version < 11.0 cuDNN version < 7.6.3	cuBLAS version ≥ 11.0 cuDNN version ≥ 7.6.3
INT8	Multiples of 16	Always but most efficient with multiples of 16; on A100, multiples of 128.
FP16	Multiples of 8	Always but most efficient with multiples of 8; on A100, multiples of 64.
TF32	N/A	Always but most efficient with multiples of 4; on A100, multiples of 32.
FP64	N/A	Always but most efficient with multiples of 2; on A100, multiples of 16.

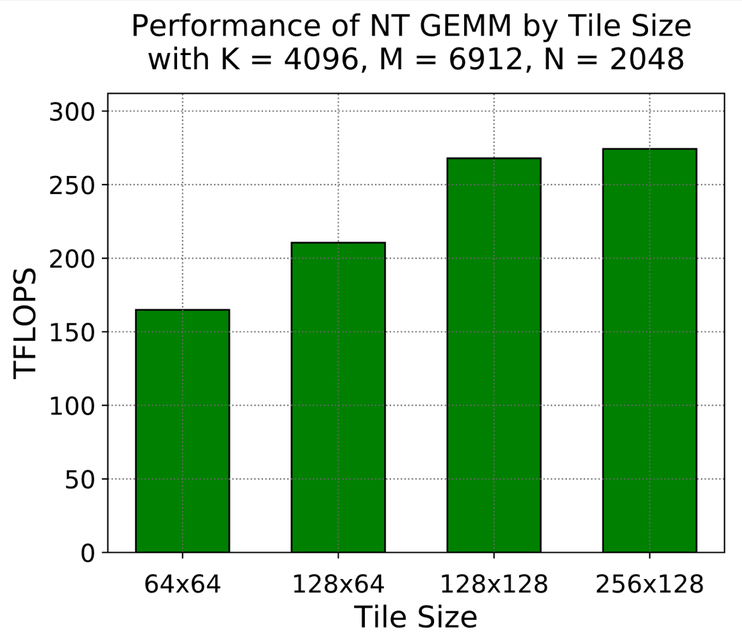
Tiling is important: **tile** and **wave quantization**

Optimality

Comparison of GEMM execution times with (a) cuBLAS 10.1 and (b) cuBLAS 11.0, both with FP16 data. Calculation is fastest (duration is lowest) when K is divisible by 8. “NN” means A and B matrices are both accessed non-transposed. NVIDIA V100-DGXS-16GB GPU.



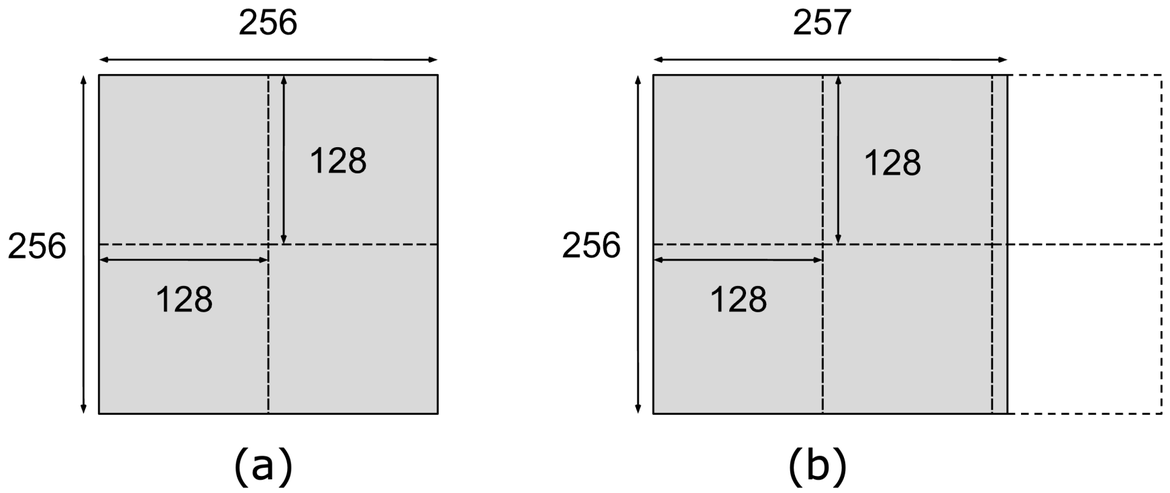
Larger tiles run more efficiently. The 256x128-based GEMM runs exactly one tile per SM, the other GEMMs generate more tiles based on their respective tile sizes. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.

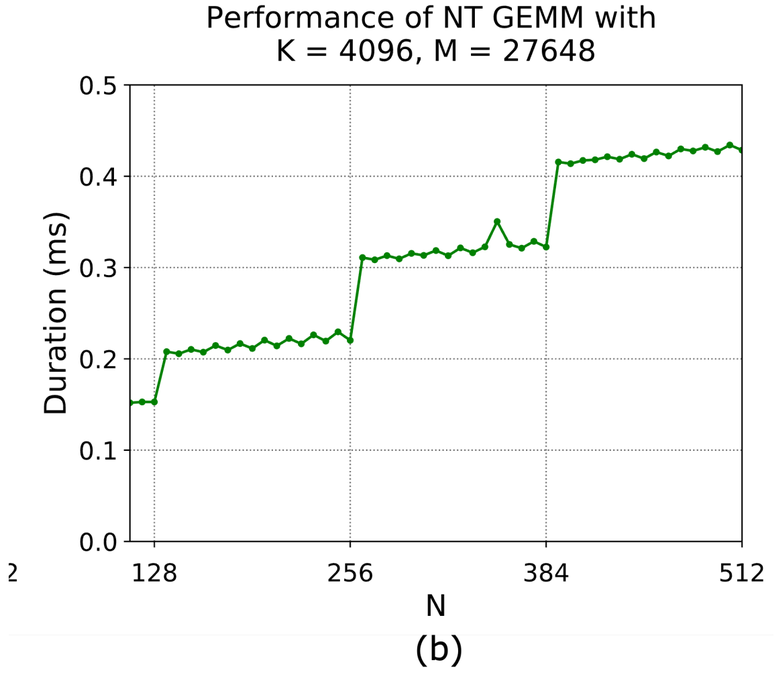


Tile Quantization

- Tile quantization occurs when matrix dimensions are not divisible by the thread block tile size.

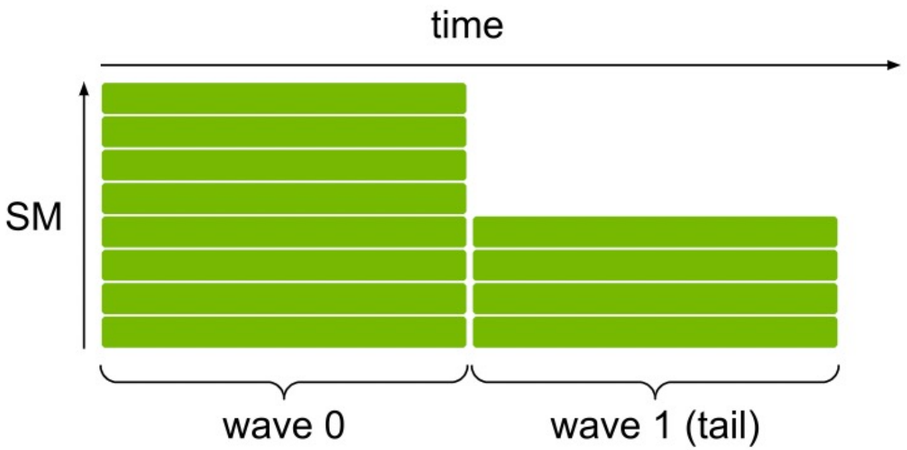
Figure. Example of tiling with 128x128 thread block tiles. (a) Best case - matrix dimensions are divisible by tile dimensions (b) Worse case - tile quantization results in six thread blocks being launched, two of which waste most of their work.



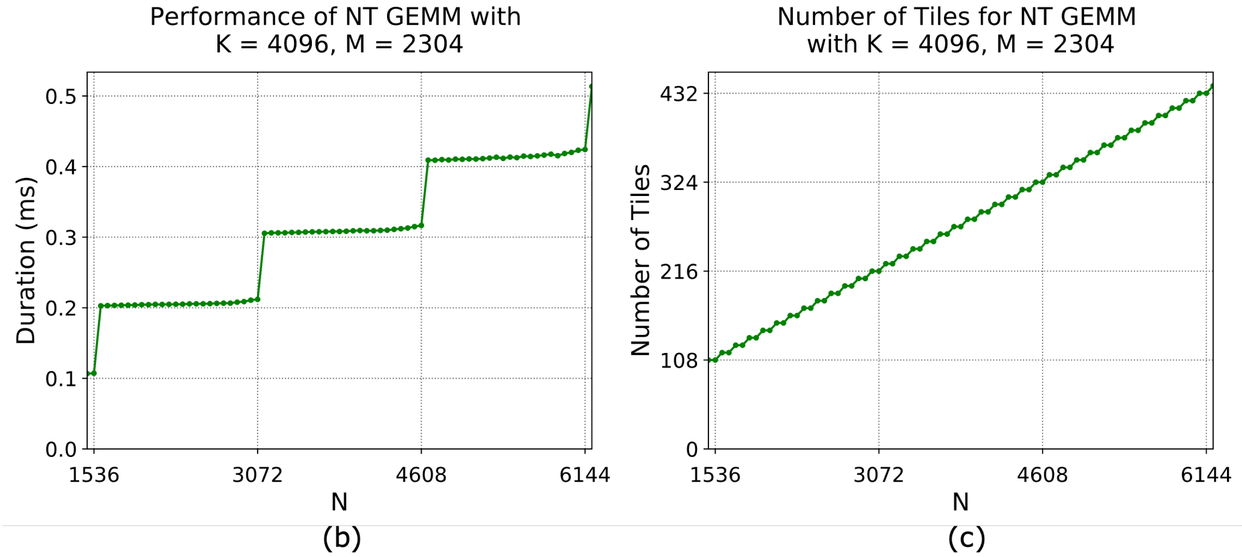


Wave quantization

- Total number of tiles is quantized to the number of multiprocessors on the GPU.



Utilization of an 8-SM GPU when 12 thread blocks with an occupancy of 1 block/SM at a time are launched for execution.



Example:

- An NVIDIA A100 GPU has 108 SMs
- Assume 256x128 thread block tiles, one thread block per SM - a wave size of 108 tiles that can execute simultaneously
- Best utilization - #tiles is an integer multiple of 108 or just below

- Background:
  - GEMM
  - Math and memory bounds
  - GPU implementation
  - Tile and wave quantization
- **Understanding DNN performance:**
  - DNN Operation Categories
  - Transformer architecture example
  - How to find a problem?
- Recommendations I:
  - Hierarchical models and UNets
  - Tensor cores
  - Convolutions
  - Linear layers
  - Transformer
- Mixed precision and sparsity
- Memory aspect
- Parallelism techniques
- Gradient checkpointing / accumulation
- Final recommendations
- Tensor RT

Memory and math limits, again

- Three factors; memory bandwidth, math bandwidth and latency
- Operations can be *memory-limited* and *math-limited*
- Math limited if: #ops / #bytes > BWmath / BWmem

Table 1. Examples of neural network operations with their arithmetic intensities. Limiters assume FP16 data and an NVIDIA V100 GPU.

Operation	Arithmetic Intensity	Usually limited by...
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 FLOPS/B	arithmetic
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 FLOPS/B	memory
Max pooling with 3x3 window and unit stride	2.25 FLOPS/B	memory
ReLU activation	0.25 FLOPS/B	memory
Layer normalization	< 10 FLOPS/B	memory

DNN Operation Categories

- **Elementwise Operations:**
  - Each element is independent of all other elements in the tensor:
    - Additional of two tensors
    - Most non-linearities (sigmoid, tanh, etc.), scale, bias, add, and others.
    - Tend to be memory-limited**,** as they perform few operations per byte accessed.
- **Reduction Operations:**
  - Produce values computed over a range of input tensor values.
    - Pooling layers, batch-normalization, layer-normalization, softmax.
    - Have a low arithmetic intensity and thus are memory limited.
- **Dot-Product Operations:**
  - Dot-products of elements from two tensors, like a weight tensor and an activation tensor.
    - Fully-connected layers, attention, etc.
    - Convolutions - one vector is the set of parameters for a filter, the other is an “unrolled” activation region
    - **Math-limited** if the corresponding matrices are large enough

Transformer example

- **Tensor Contractions**
  - Linear layers and components of Multi-Head Attention all do batched matrix-matrix multiplications.
- **Statistical Normalizations**
  - Softmax and layer normalization are less compute-intensive than tensor contractions, and involve one or more reduction operations.
- **Element-wise Operators**
  - Biases, dropout, activations, and residual connections.

Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

DATA MOVEMENT IS ALL YOU NEED: A CASE STUDY ON OPTIMIZING TRANSFORMERS https://arxiv.org/pdf/2007.00072.pdf

How to find a problem?

- Look up the number of SMs on the GPU, and determine the ops:bytes ratio for the GPU.
- Compute the arithmetic intensity for the algorithm.
- Determine if there is sufficient parallelism to saturate the GPU by estimating the number and size of thread blocks.
- The most likely performance limiter is:
  - Latency if there is not sufficient parallelism
  - Math if there is sufficient parallelism and algorithm arithmetic intensity is higher than the GPU ops:byte ratio.
  - Memory if there is sufficient parallelism and algorithm arithmetic intensity is lower than the GPU ops:byte ratio.

- Background:
  - GEMM
  - Math and memory bounds
  - GPU implementation
  - Tile and wave quantization
- Understanding DNN performance:
  - DNN Operation Categories
  - Transformer architecture example
  - What is the limit? Guide
- **Recommendations I:**
  - Hierarchical models and UNets
  - Tensor cores
  - Convolutions
  - Linear layers
  - Transformer
- Mixed precision and sparsity
- Memory aspect
- Parallelism techniques
- Gradient checkpointing / accumulation
- Final recommendations
- Tensor RT

Recommendations I

- 1. Operating In Math-Limited Regime Where Possible:
  - If the speed of a routine is limited by calculation rate (**math-limited** or **math-bound**), performance can be improved by enabling Tensor Cores.
- 2. Using Tensor Cores Efficiently With Alignment:
  - Use parameter shapes such that compute is aligned with Tensor Cores characteristics to minimize wave tails.
- 3. Choosing Parameters To Maximize Execution Efficiency:
  - GPUs perform operations efficiently by dividing the work between many parallel processes

Recommendations for hierarchical models

- For memory limited layers (usually first layers):
  - Use dense convolutions instead of depth-wise (especially for FP16, INT8)
  - Minimize pooling (e.g. no squeeze-and-excitation layers)
  - Avoid Layer Normalization
  - Use BN and activations as they are foldable during inference (TRT does automatically)
- For math limited layers (usually later layers):
  - Depth-wise should be fine
  - SE works
  - Layer normalization doesn’t hurt that much
- ReLU is the best activation as it can be fused with conv-BN
- For large image resolution use torch.nn.PixelShuffle(upscale_factor)

Maximizing Tensor Core usage:

- FP16 input/output channels for linear and conv should be multiplier 16, for INT8 – multiplier 32.
- Mini-batch to be a multiple of 8
- For sequence problems, pad the sequence length to be a multiple of 16 (FP16) or 32 (INT32)
- Concatenate matrices together like qkv for transformer

- EfficientNet v1 and MobileNet don’t follow these recommendations
- EfficientNet v2 is much faster

From https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#opt-tensor-cores

Convolutions

- Layout choice influences performance: NCHW is slower than NHWC
- Convolutions implemented for Tensor Cores require NHWC layout and are fastest when input tensors are laid out in NHWC.
- TensorFlow, Pytorch, MXNet support it

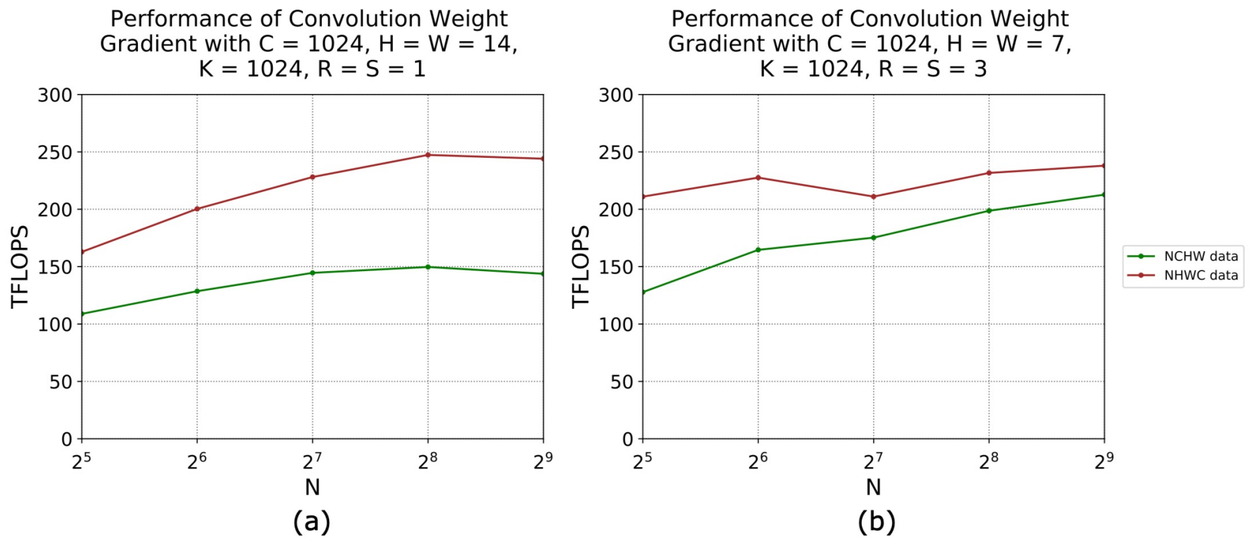


Figure 2. Kernels that do not require a transpose (NHWC) perform better than kernels that require one or more (NCHW). NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.

PyTorch:

```
input_data =
input_data.to(memory_format=torch.channels_last)
model = model.to(memory_format=torch.channels_last)
```

Convolutions

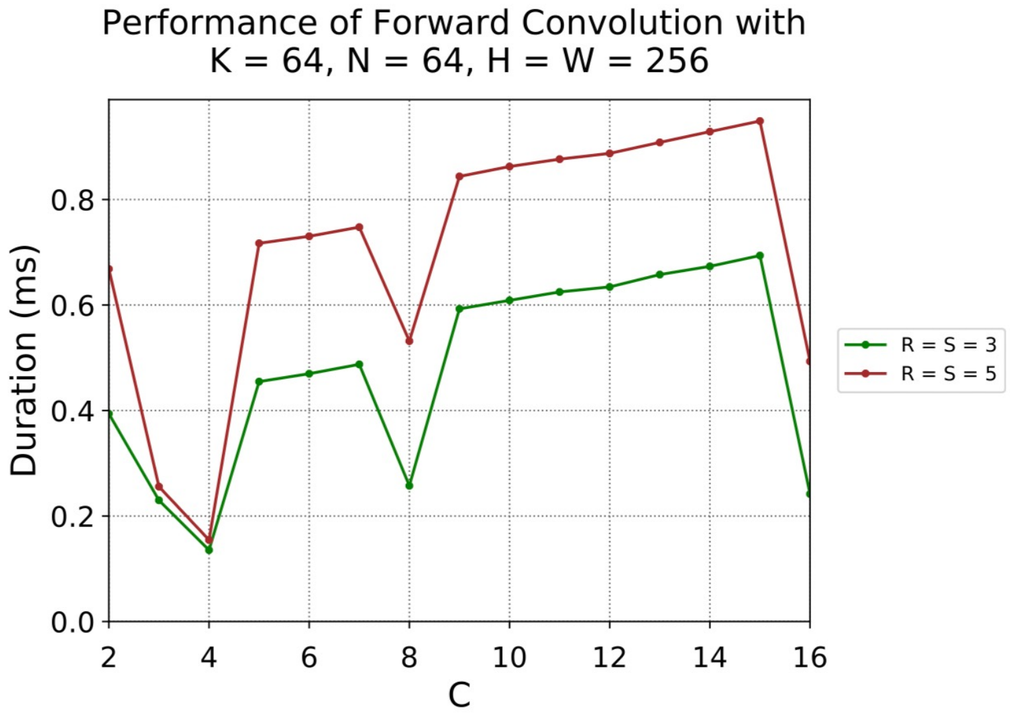
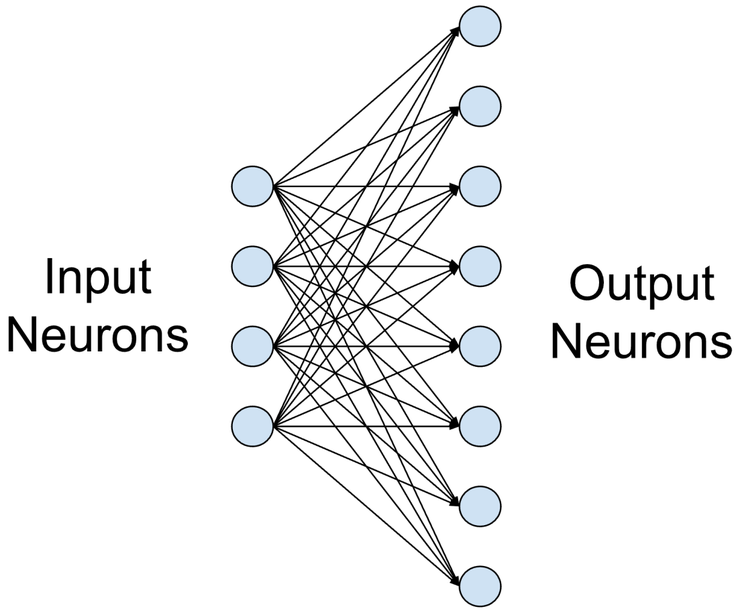


Figure 8. Specialized kernels for C = 4 speed up common first layers in convolutional neural nets (NHWC data used). Choosing C = 4 or a multiple of 8 gives best performance. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1

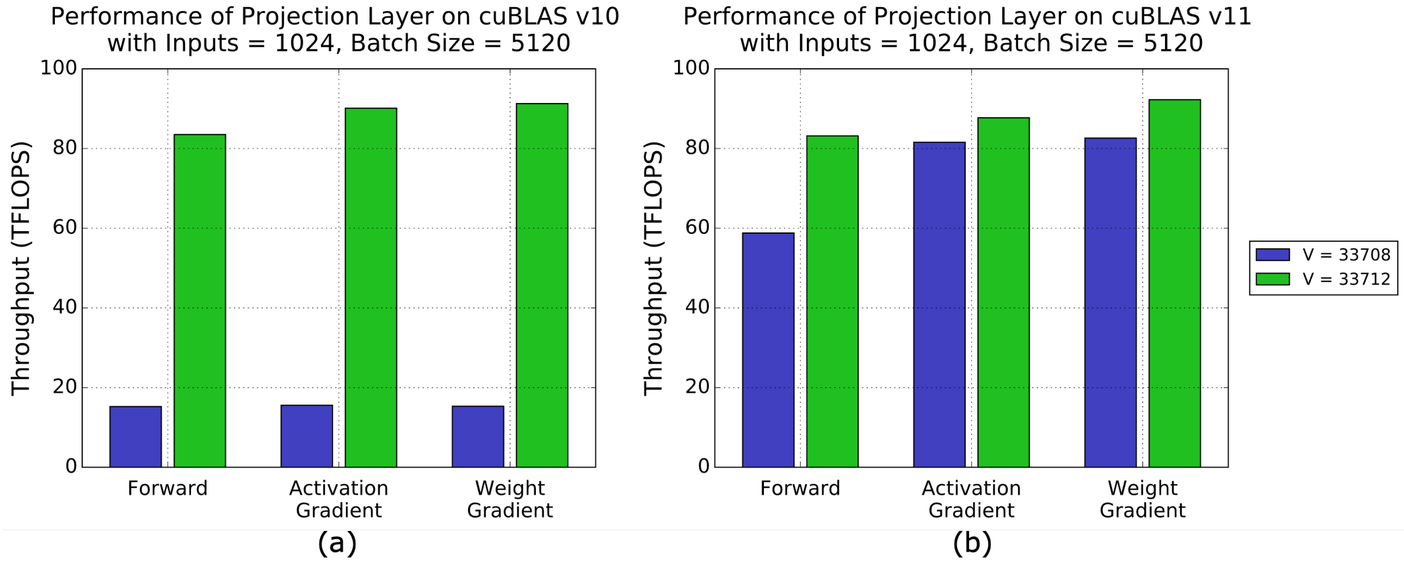
Linear Layers

- Batch size and the number of inputs and outputs to be divisible by 4 (TF32) / 8 (FP16) / 16 (INT8) to run efficiently on Tensor Cores.
- For A100: be divisible by 32 (TF32) / 64 (FP16) / 128 (INT8).
- When #parameters ↓: batch size and #inputs and #outputs to be divisible by at least 64 and ideally 256.
- Larger values for batch size and the number of inputs and outputs improve parallelization and efficiency.
- As a rough guideline, choose batch sizes and neuron counts greater than 128 to avoid being limited by memory bandwidth (NVIDIA®A100-SXM4-80GB; this threshold is similar for other A100 and V100 GPUs).

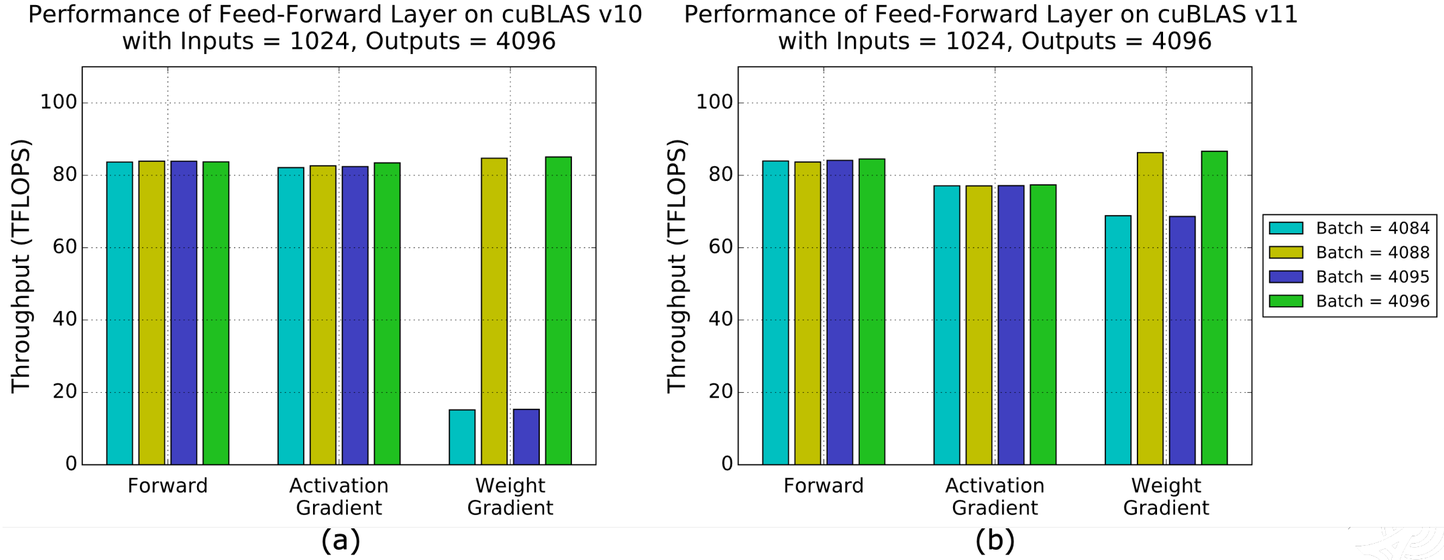


Transformers

Performance benefits substantially from choosing vocabulary size to be a multiple of 8 with both (a) cuBLAS version 10.1 and (b) cuBLAS version 11.0. The projection layer uses 1024 inputs and a batch size of 5120. NVIDIA V100-SXM2-16GB GPU



Weight gradient calculation for a fully-connected layer benefits from padding batch size to be a multiple of 8 with both (a) cuBLAS version 10.1 and (b) cuBLAS version 11.0. The first fully-connected layer (4096 outputs, 1024 inputs) from the Transformer feed-forward network is shown. NVIDIA V100-SXM2-16GB GPU.



- Background:
  - GEMM
  - Math and memory bounds
  - GPU implementation
  - Tile and wave quantization
- Understanding DNN performance:
  - DNN Operation Categories
  - Transformer architecture example
  - What is the limit? Guide
- Recommendations I:
  - Hierarchical models and UNets
  - Tensor cores
  - Convolutions
  - Linear layers
  - Transformer

- **Mixed precision and sparsity**
- Memory aspect
- Parallelism techniques
- Gradient checkpointing / accumulation
- Final recommendations
- Operator fusion
- Tensor RT

Mixed precision training

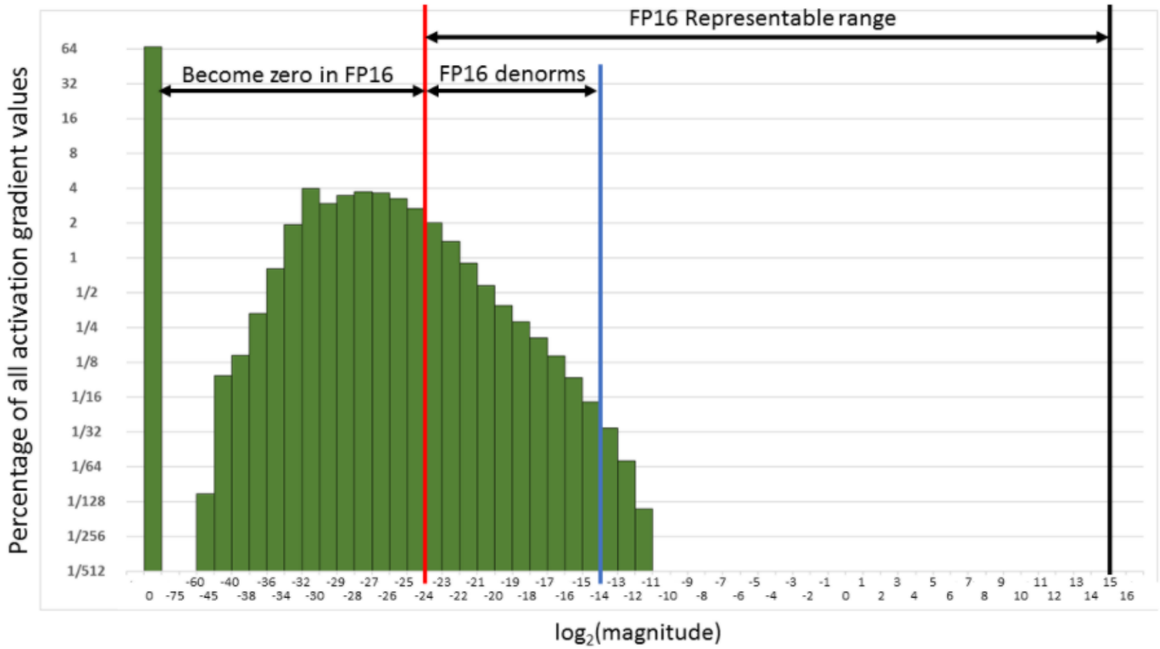
Example: FP32 training of Multibox SSD network

- Histogram shows activation gradient magnitudes throughout FP32 training; both axes are logarithmic.

- Observations:
  - Dynamic range of FP16 would be sufficient to cover the entire histogram. ☺
  - Without “shifting” the histogram, half of the activations would be casted to 0, however. ☹

- Idea: “shifting” = multiplication with a scale factor!

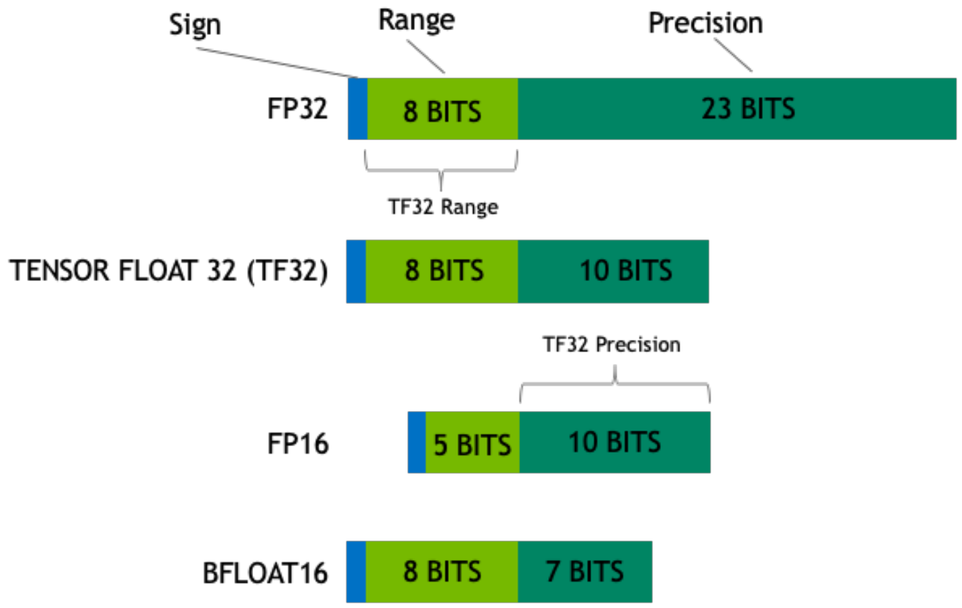
- Concern: Do I need to run a full training in order to find the scaling factor? → No, automatic mixed precision comes to the rescue! ☺



A note on data types

TF32 only makes sense

- Mixed precision training is mostly about the dynamic range and less about the precision:
  - exponent → dynamic range
  - significand field → precision
- TF32 is a great compromise between FP32 (same range) and FP16 (same precision)
- TF32 is automatically enabled in NGC containers
- No code change is necessary!



How to use it?

In Pytorch

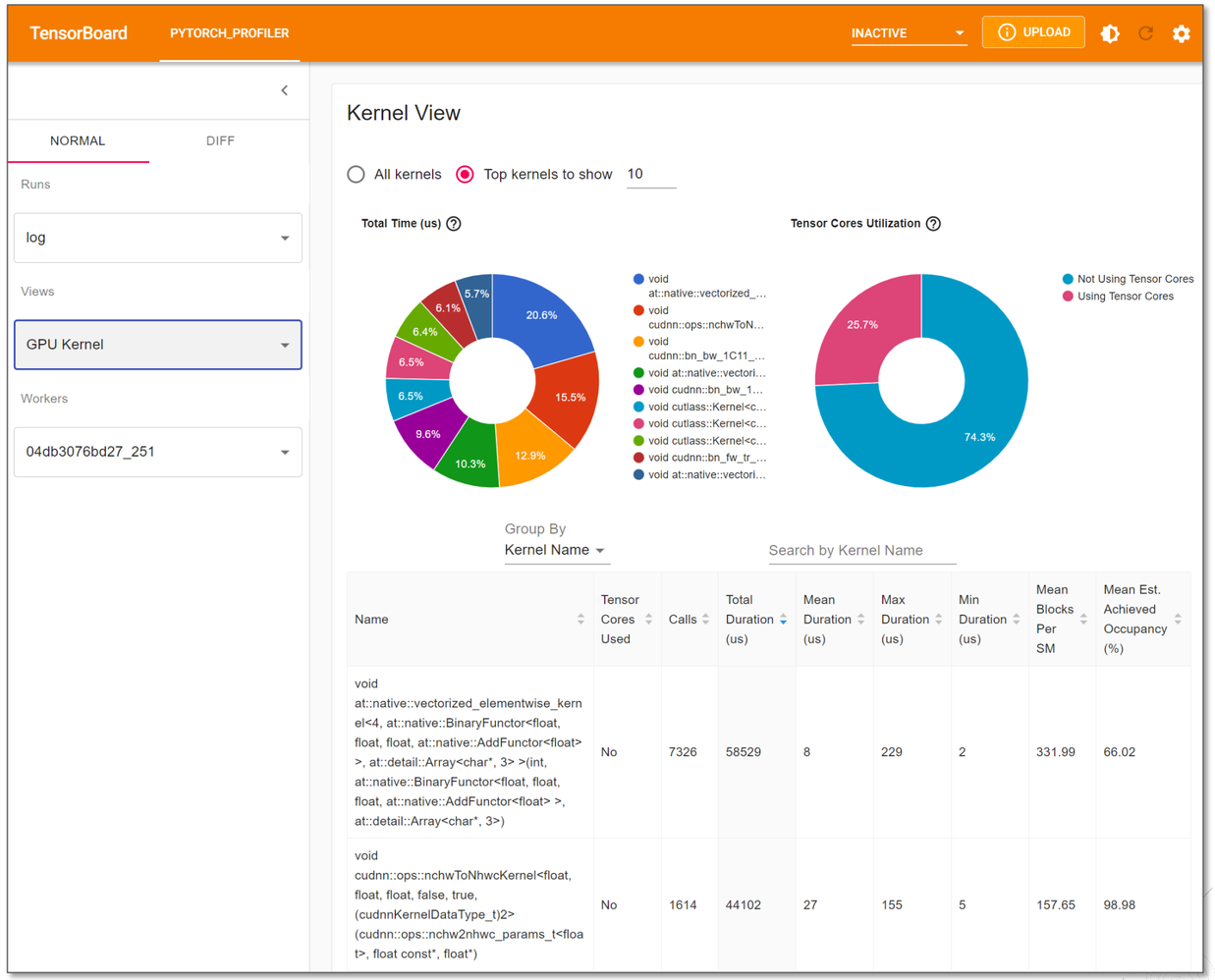
- Backward passes under autocast are not recommended.
- Backward ops run in the same dtype autocast chose for corresponding forward ops.
- scaler.step() first unscales the gradients of the optimizer's assigned params.
- If these gradients contain infs or NaNs, optimizer.step() is skipped.

```
# initialize gradient scaler
scaler = GradScaler()

# training loop
for epoch in epochs:
      for input, target in data:

            # zero gradient buffers
            optimizer.zero_grad()

            # forward pass with autocasting
            with autocast():
                  output = model(input)
                  loss = loss_fn(output, target)

            # call backward() on scaled loss
            scaler.scale(loss).backward()

            # update if no issues
            scaler.step(optimizer)

            # updates the scale for next iteration.
            scaler.update()
```

Am I use Tensor Cores?

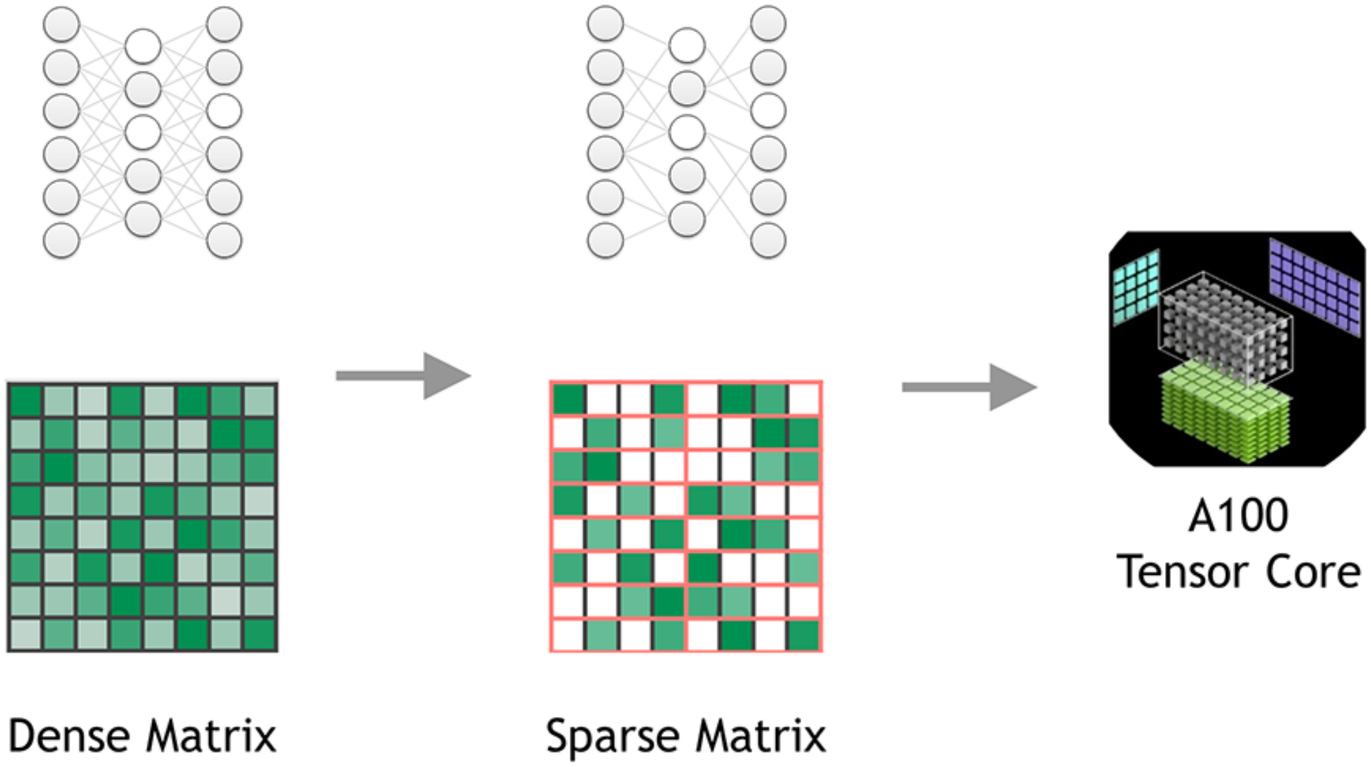
https://pytorch.org/docs/stable/profiler.html

```
from torch import profiler

prof_schedule = profiler.schedule(wait=2,
                                  warmup=2,
                                  active=5,
                                  repeat=0)


callback =
profiler.tensorboard_trace_handler(`./log`)

prof = profiler.profile(schedule=prof_schedule,
on_trace_ready=callback,
record_shapes=False,
with_stack=False)

prof.start()

for it in range(num_iterations):
    # code to be profiled
    ...
    prof.step()

prof.stop()
```

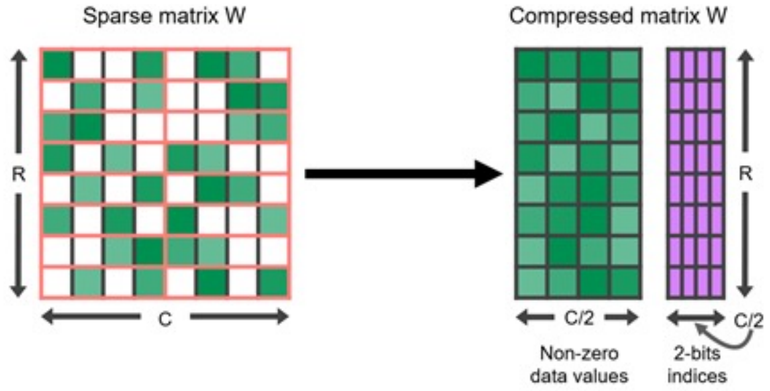


Ampere sparsity

2:4 Sparsity



- 2 out of 4 weights are zero
- Pruning can be done by magnitude
- Sometime requires finetuning
- Up to 30% speed up
- Available as part of APEX library

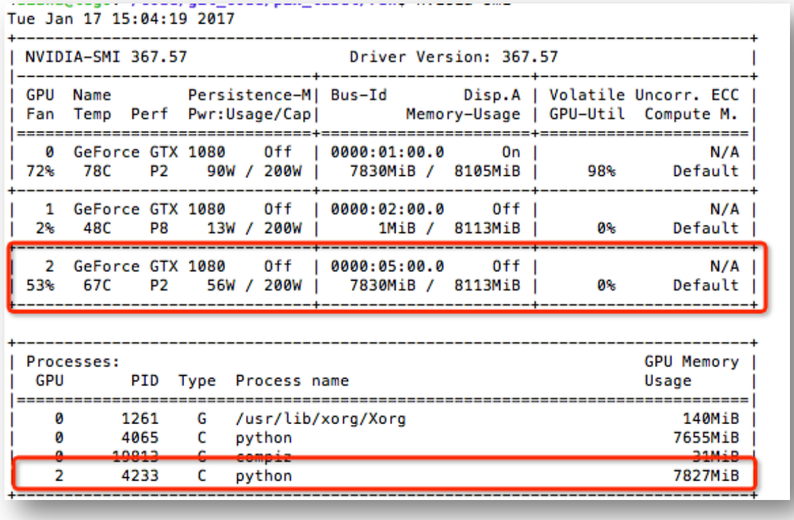


- Background:
  - GEMM
  - Math and memory bounds
  - GPU implementation
  - Tile and wave quantization
- Understanding DNN performance:
  - DNN Operation Categories
  - Transformer architecture example
  - What is the limit? Guide
- Recommendations I:
  - Hierarchical models and UNets
  - Tensor cores
  - Convolutions
  - Linear layers
  - Transformer
- Mixed precision and sparsity
- **Memory aspect**
- **Parallelism techniques**
- **Gradient checkpointing / accumulation**
- Final recommendations
- Operator fusion
- Tensor RT

GPU memory

During training

- **Model Weights**
  - 4 bytes * number of parameters for fp32 training
  - 6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16 in memory)
- **Optimizer States**
  - 8 bytes * number of parameters for normal AdamW (maintains 2 states)
  - 2 bytes * number of parameters for 8-bit AdamW optimizers like bitsandbytes
  - 4 bytes * number of parameters for optimizers like SGD with momentum (maintains only 1 state)
- **Gradients**
  - 4 bytes * number of parameters for either fp32 or mixed precision training (gradients are always kept in fp32)
- **Forward Activations**
  - size depends on many factors, the key ones being sequence length, hidden size and batch size

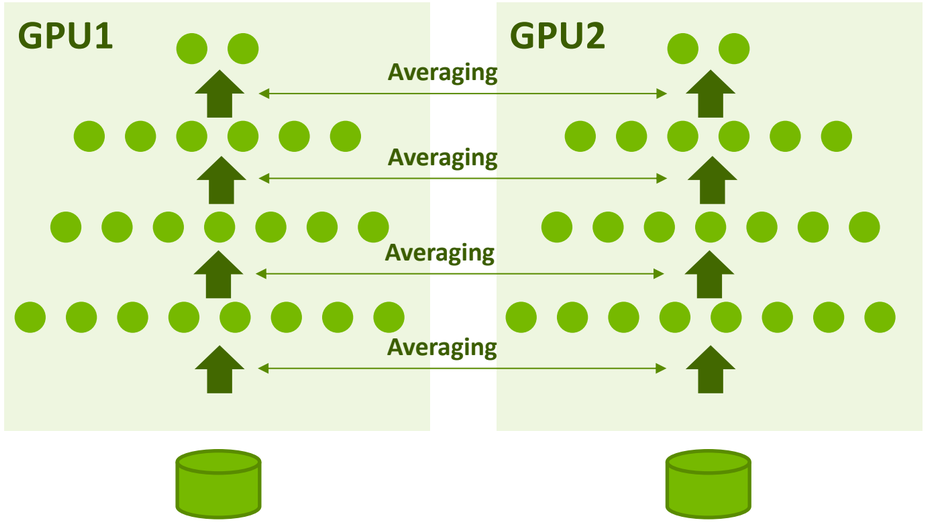


Data with model parallelism

Larger batchsize and larger models

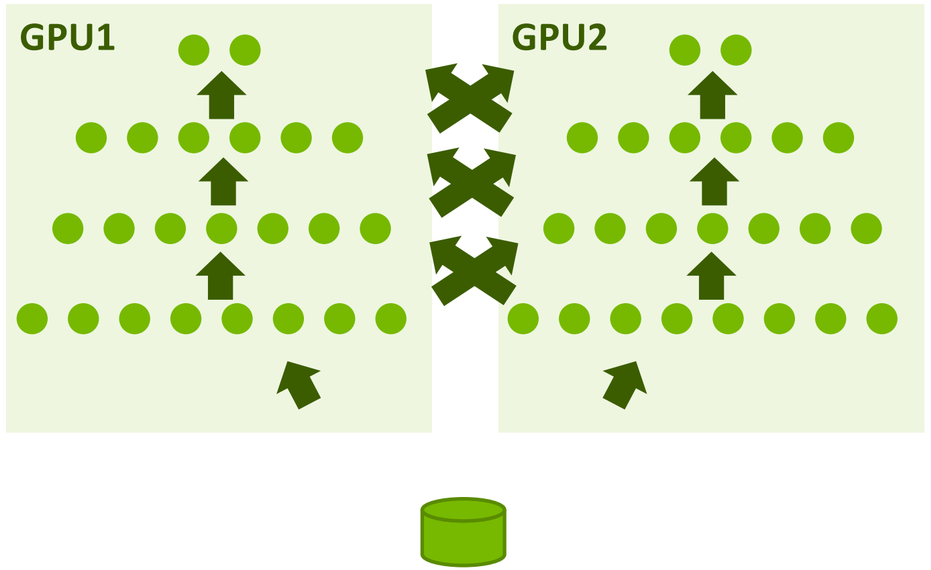
Data Parallelism

- Allows to speed up training
- All workers train on different data
- All workers have the same copy of the model
- Neural network gradients (weight changes) are exchanged



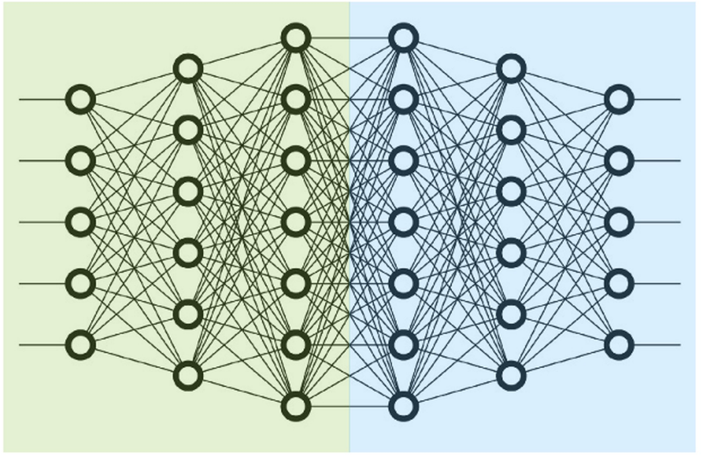
Model Parallelism

- Allows for a bigger model
- All workers train on the same data
- Parts of the model are distributed across GPUs
- Neural network activations are exchanged

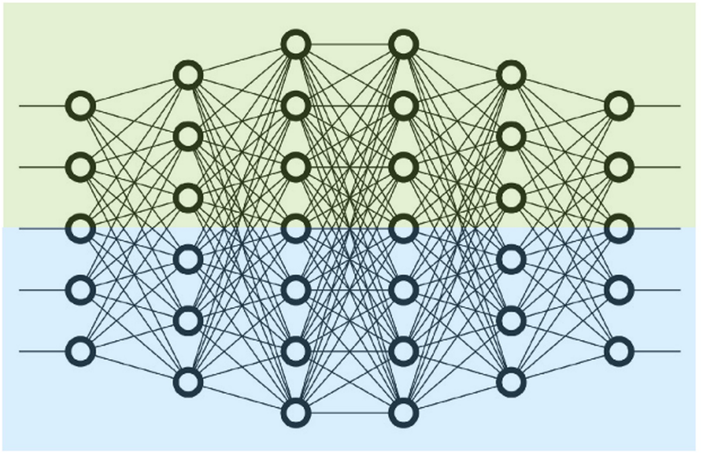


Model parallelism

- **Pipeline (Inter-Layer) Parallelism**
  - Split sets of layers across multiple devices
  - Layer 0,1,2 and layer 3,4,5 are on difference devices

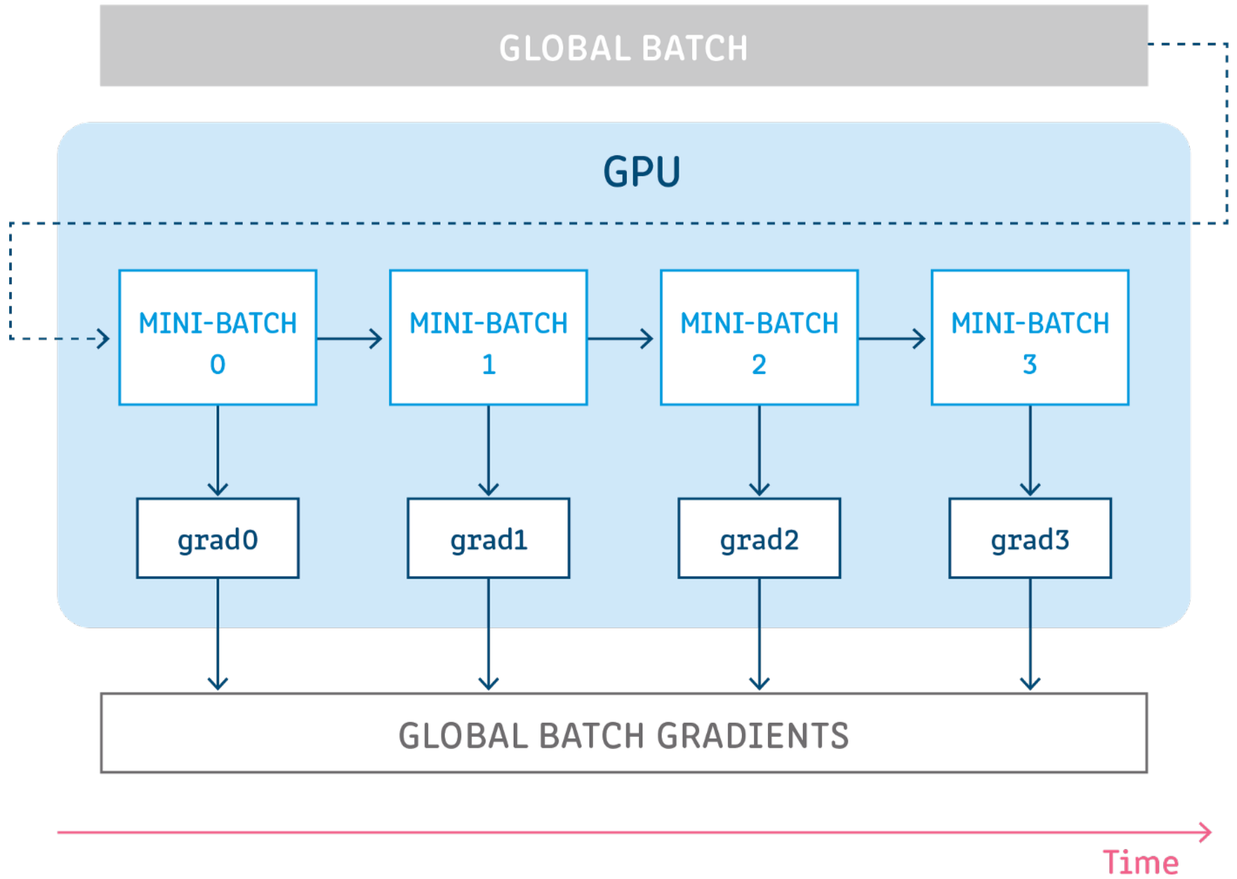


- **Tensor (Intra-Layer) Parallelism**
  - Split individual layers across multiple devices
  - Both devices compute different parts of Layer 0,1,2,3,4,5



Gradient accumulation

- Gradient accumulation is a mechanism to split the batch of samples — used for training a neural network — into several mini-batches of samples that will be run sequentially.



https://towardsdatascience.com/what-is-gradient-accumulation-in-deep-learning-ec034122cfa

Gradient accumulation

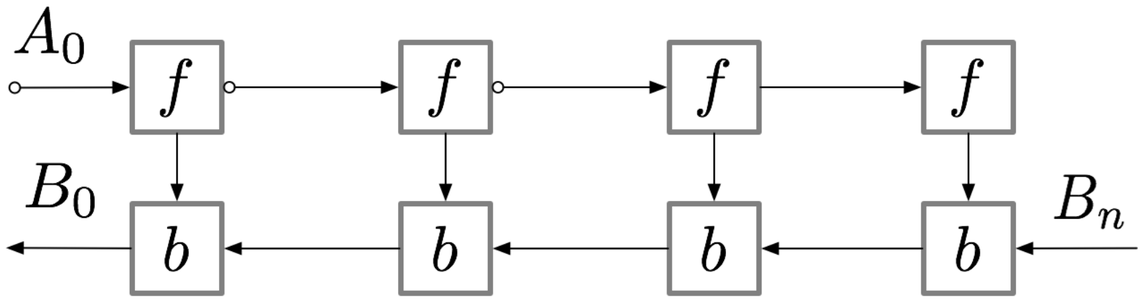
```
optimizer = ...


for epoch in range(...):
   for i, sample in enumerate(dataloader):
      inputs, labels = sample
      optimizer.zero_grad()
      # Forward Pass
      outputs = model(inputs)
      # Compute Loss and Perform Back-
      propagation
      loss = loss_fn(outputs, labels)
      loss.backward()
      # Update Optimizer          optimizer.step()
```

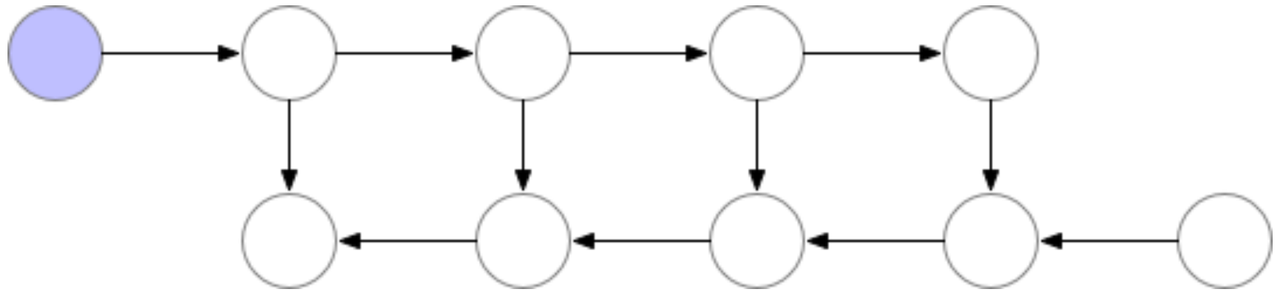
```
optimizer = ...
NUM_ACCUMULATION_STEPS = ...
for epoch in range(...):
        for idx, sample in enumerate(dataloader):
        inputs, labels = sample
        # Forward Pass
        outputs = model(inputs)
        # Compute Loss and Perform Back-    propagation
        loss = loss_fn(outputs, labels)
        # Normalize the Gradients
        loss = loss / NUM_ACCUMULATION_STEPS
        loss.backward()
        if ((idx + 1) % NUM_ACCUMULATION_STEPS ==
            0) or (idx + 1 == len(dataloader)):
                optimizer.zero_grad()
                # Update Optimizer
                optimizer.step()
```

Activation Re-computation or gradient checkpointing

https://pytorch.org/docs/stable/checkpoint.html

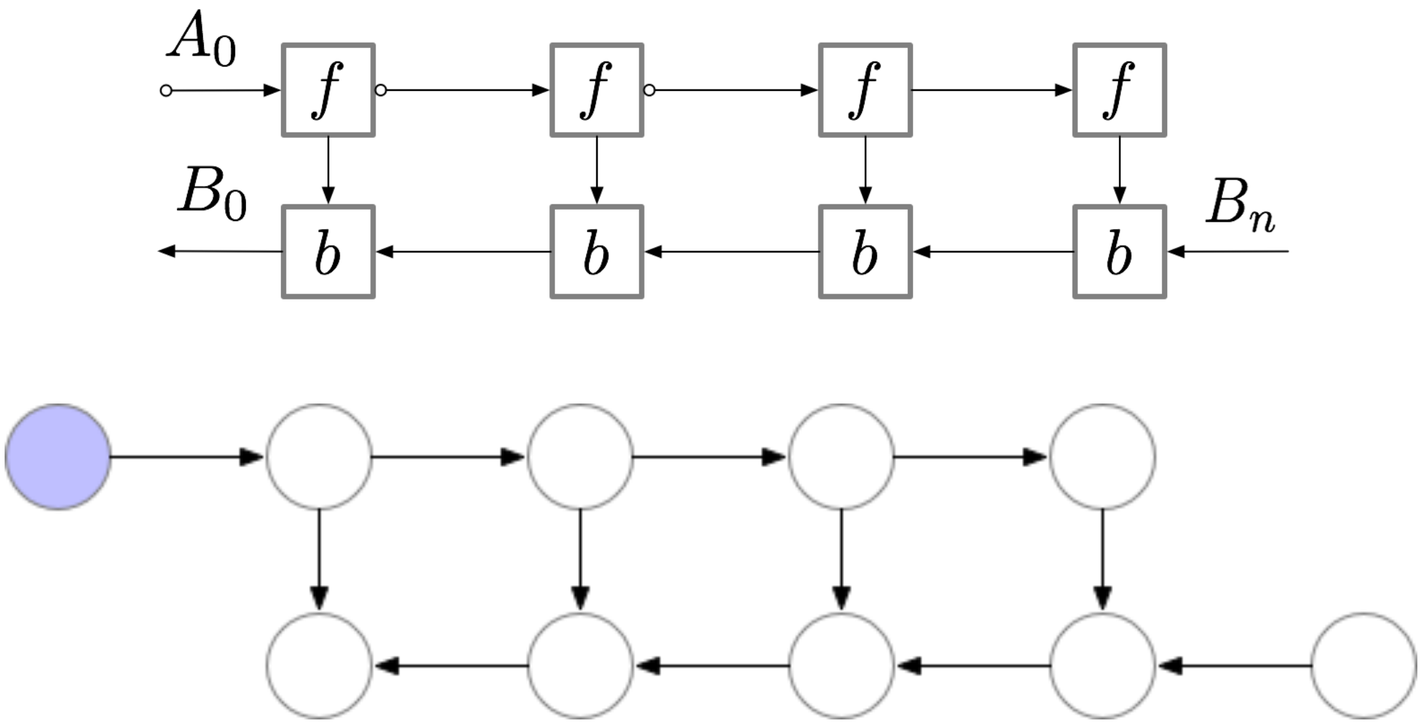
- The memory intensive part of training deep neural networks is computing the gradient of the loss by backpropagation.
- By checkpointing nodes in the computation graph defined by your model, and recomputing the parts of the graph in between those nodes during backpropagation, it is possible to calculate gradients at reduced memory cost.
- **When training deep feed-forward neural networks consisting of *n* layers, you can reduce the memory consumption to *O(sqrt(n))*, at the cost of performing one additional forward pass.**





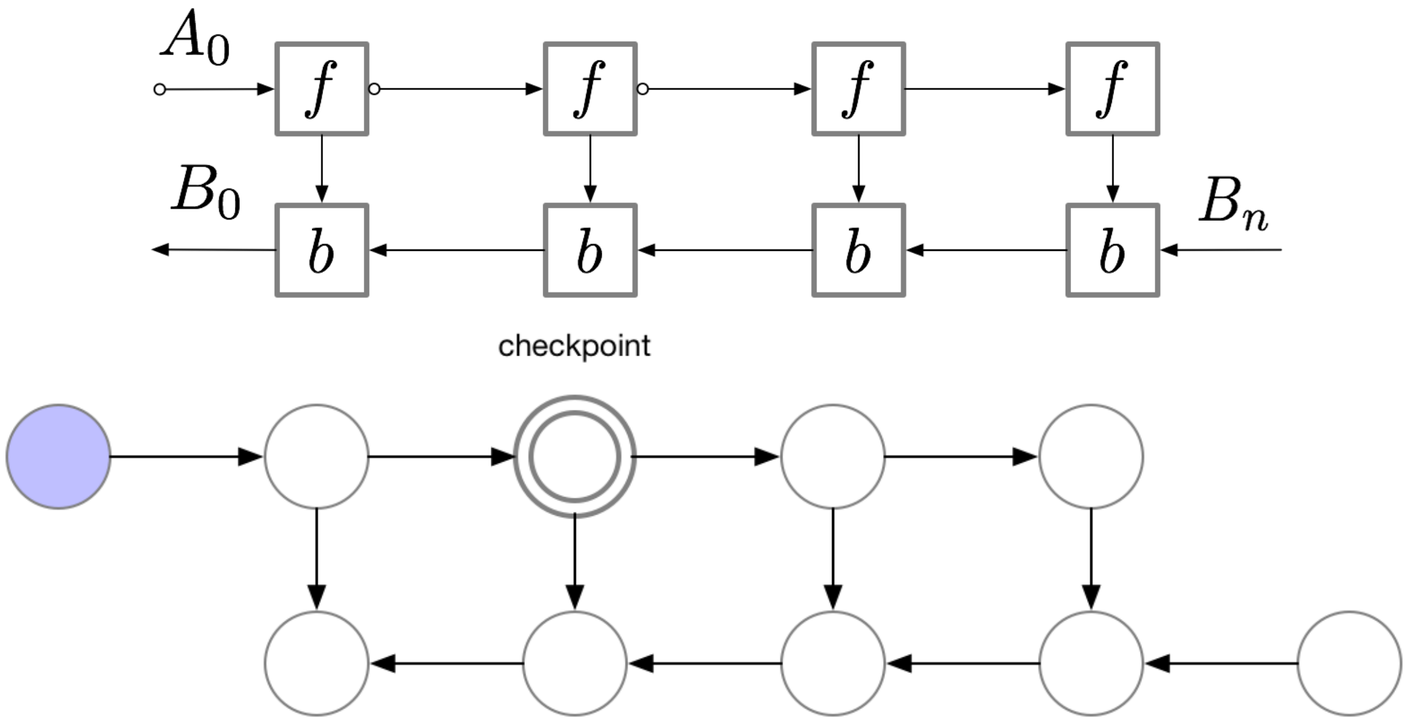
https://github.com/cybertronai/gradient-checkpointing

Activation Re-computation or gradient checkpointing



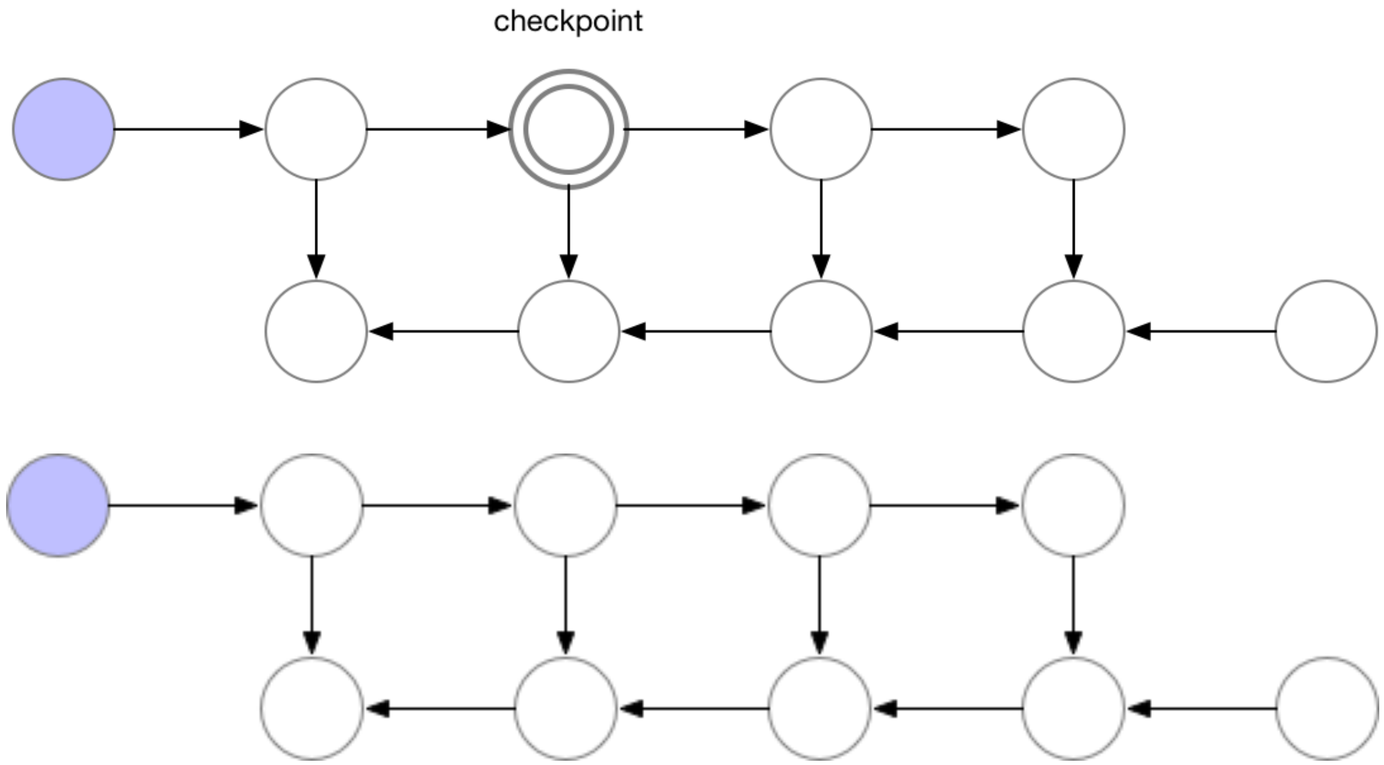
We might for instance simply recompute every node from the forward pass each time we need it.

Activation Re-computation or gradient checkpointing



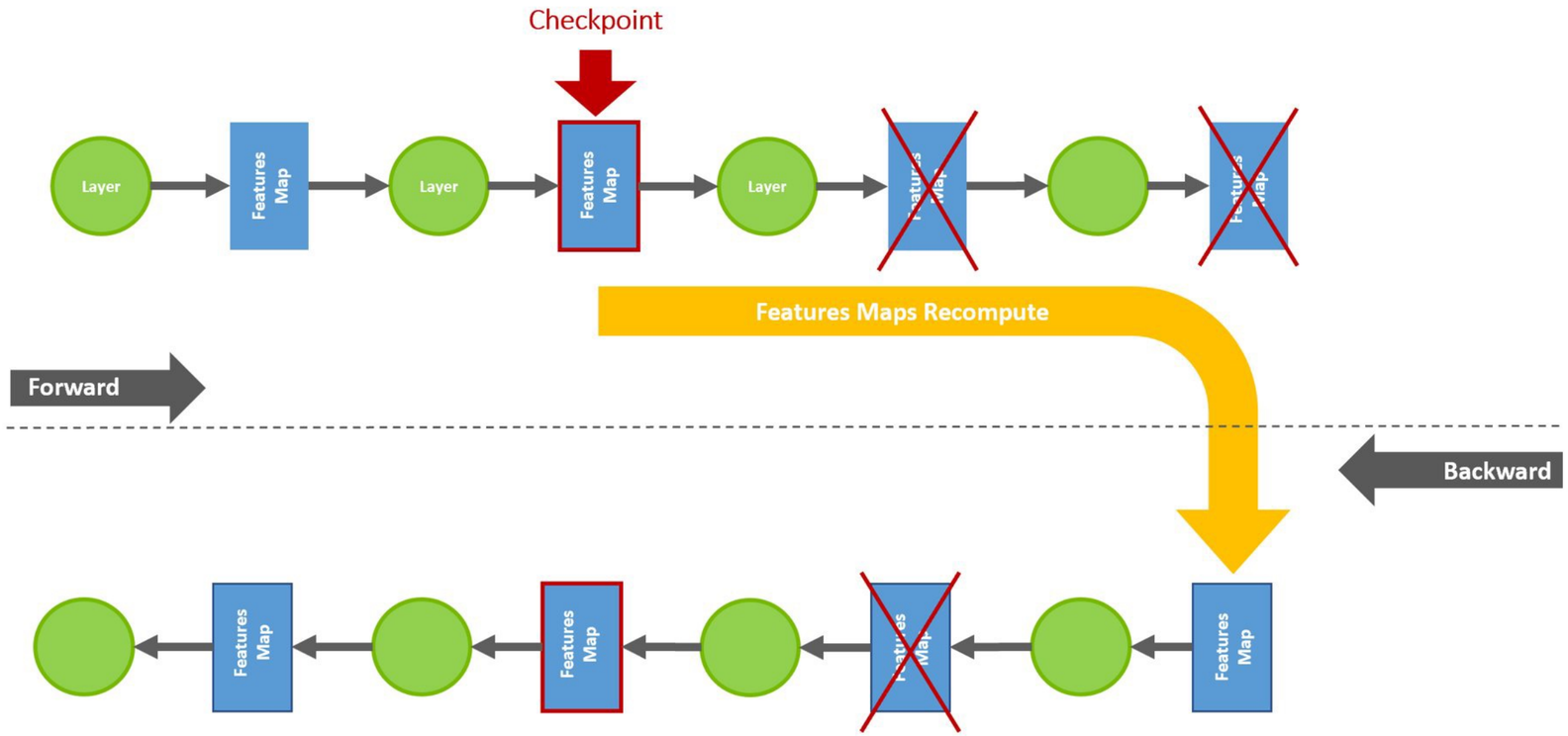
The strategy we use here is to mark a subset of the neural net activations as checkpoint nodes.

Activation Re-computation or gradient checkpointing



Activation recompute challenges

Activation Recompute



Balance the memory savings and computational overhead?

- Background:
  - GEMM
  - Math and memory bounds
  - GPU implementation
  - Tile and wave quantization
- Understanding DNN performance:
  - DNN Operation Categories
  - Transformer architecture example
  - What is the limit? Guide
- Recommendations I:
  - Hierarchical models and UNets
  - Tensor cores
  - Convolutions
  - Linear layers
  - Transformer
- Mixed precision and sparsity
- Memory aspect
- Parallelism techniques
- Gradient checkpointing / accumulation
- **Final recommendations**
- Operator fusion
- Tensor RT

Optimization techniques

- Leverage AMP (Automatic Mixed Precision)
- Activate Gradient Checkpoint
- Empty CUDA Cache:
  - torch.cuda.empty_cache()
- Manually launch Garbage Collection:
  - Import gc
  - gc.collect()
- Enable 8-bit Adam
- Remove bias for convolutional layers followed by batch normalization
- Implement Gradient Accumulation
- Batch size divisible by 8
- Input/output channels:
  - FP16 multiplier of 16
  - INT8 – multiplier 32
- Implement Parallelism
  - Data
  - Model
  - Pipeline
  - Tensor
- Offload tensors to CPU (e.g. EMA, gradient checkpoint)
- Overlap computing and communication
- Improve data loader
  - Constant Buffer Optimization
  - Contiguous Memory Optimization
- Use Fused Kernel
- Low-Rank adaptation (specific for LLM)

Optimization techniques

Method	Speed	Memory
Gradient accumulation	No	Yes
Gradient checkpointing	No	Yes
Mixed precision training	Yes	(No)
Batch size	Yes	Yes
Optimizer choice	Yes	Yes
DataLoader	Yes	No
DeepSpeed Zero	No	Yes

https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one

Important CUDNN flags

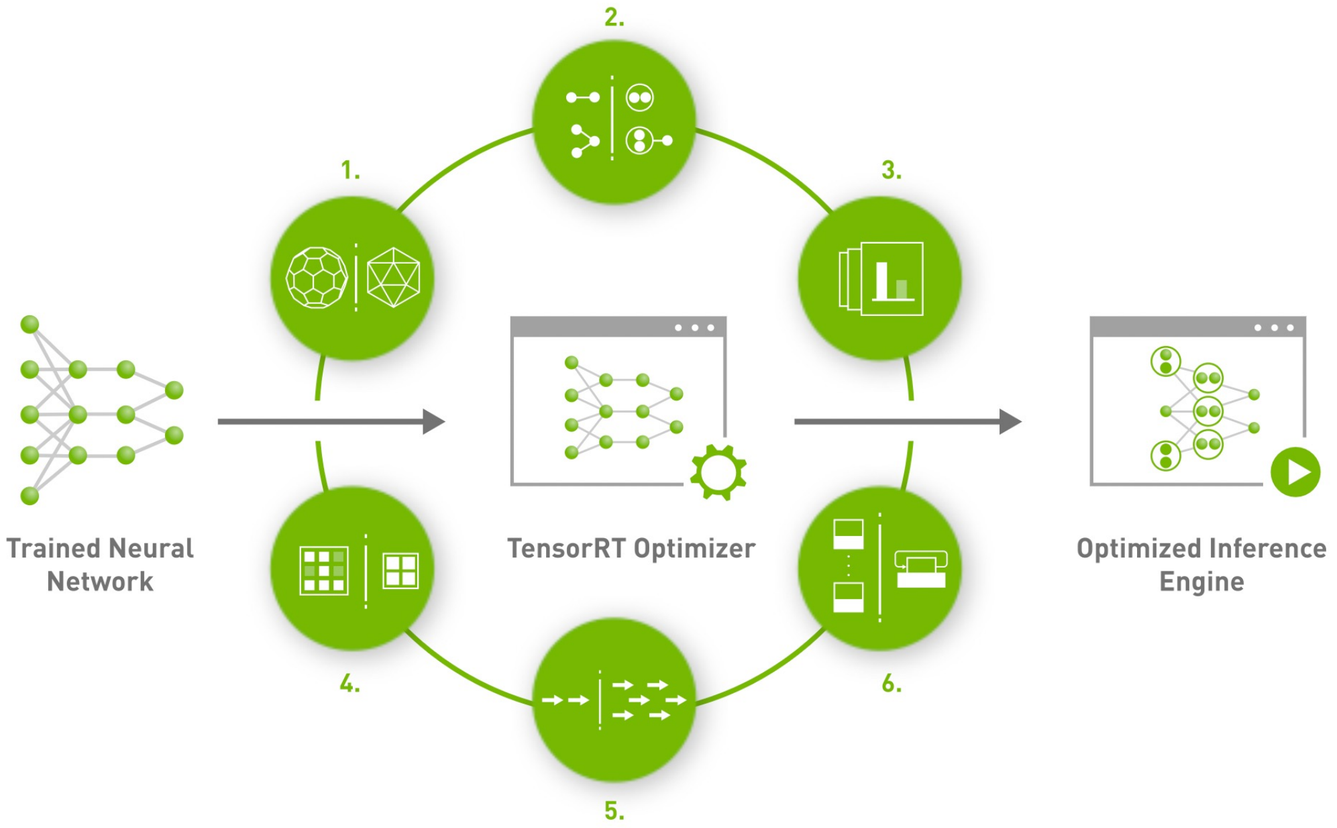
https://pytorch.org/docs/stable/backends.html#torch-backends-cudnn

Important aspects to consider:

- In NGC containers, the usage of TensorFloat-32 is enabled by default in order to accelerate FP32 calculations using tensor cores on Ampere or newer GPUs.
- Certain classes of CUDA functions are a potential source of non-determinism, such as atomicAdd, where the order of parallel additions to the same value is undetermined and, for floating-point variables, a source of variance in the results.
- cuDNN can automatically determine which combination of primitives is most optimal. Only use this flag when input sizes of a model are no changing!

```
# get the cuDNN version
torch.backends.cudnn.version()

# check availability
torch.backends.cudnn.is_available()

# enabling cuDNN (default = True)
torch.backends.cudnn.enabled = True

# enabling TF32 (default = True for DL)
torch.backends.cudnn.allow_tf32 = True

# enable determinism (default = False)
torch.backends.cudnn.deterministic = False

# enable auto-tuning (default = False)
torch.backends.cudnn.benchmark = True
```

What is TensorRT?



1. **Weight & Activation Precision Calibration**
   Maximizes throughput by quantizing models to INT8 while preserving accuracy

2. **Layer & Tensor Fusion**
   Optimizes use of GPU memory and bandwidth by fusing nodes in a kernel

3. **Kernel Auto-Tuning**
   Selects best data layers and algorithms based on target GPU platform

4. **Dynamic Tensor Memory**
   Minimizes memory footprint and re-uses memory for tensors efficiently

5. **Multi-Stream Execution**
   Scalable design to process multiple input streams in parallel

6. **Time Fusion**
   Optimizes recurrent neural networks over time steps with dynamically generated kernels

Thank you

Pavlo Molchanov, pmolchanov@nvidia.com

Slides credit:
Giuseppe Fiameni gfiameni@nvidia.com
Nikita Korobov nkorobov@nvidia.com
https://docs.nvidia.com/deeplearning/performance/dl-performance-getting-started/index.html

Disclaimer: Results, numbers and performance are reported from the research perspective.
For the exact performance please contact NVIDIA product managers