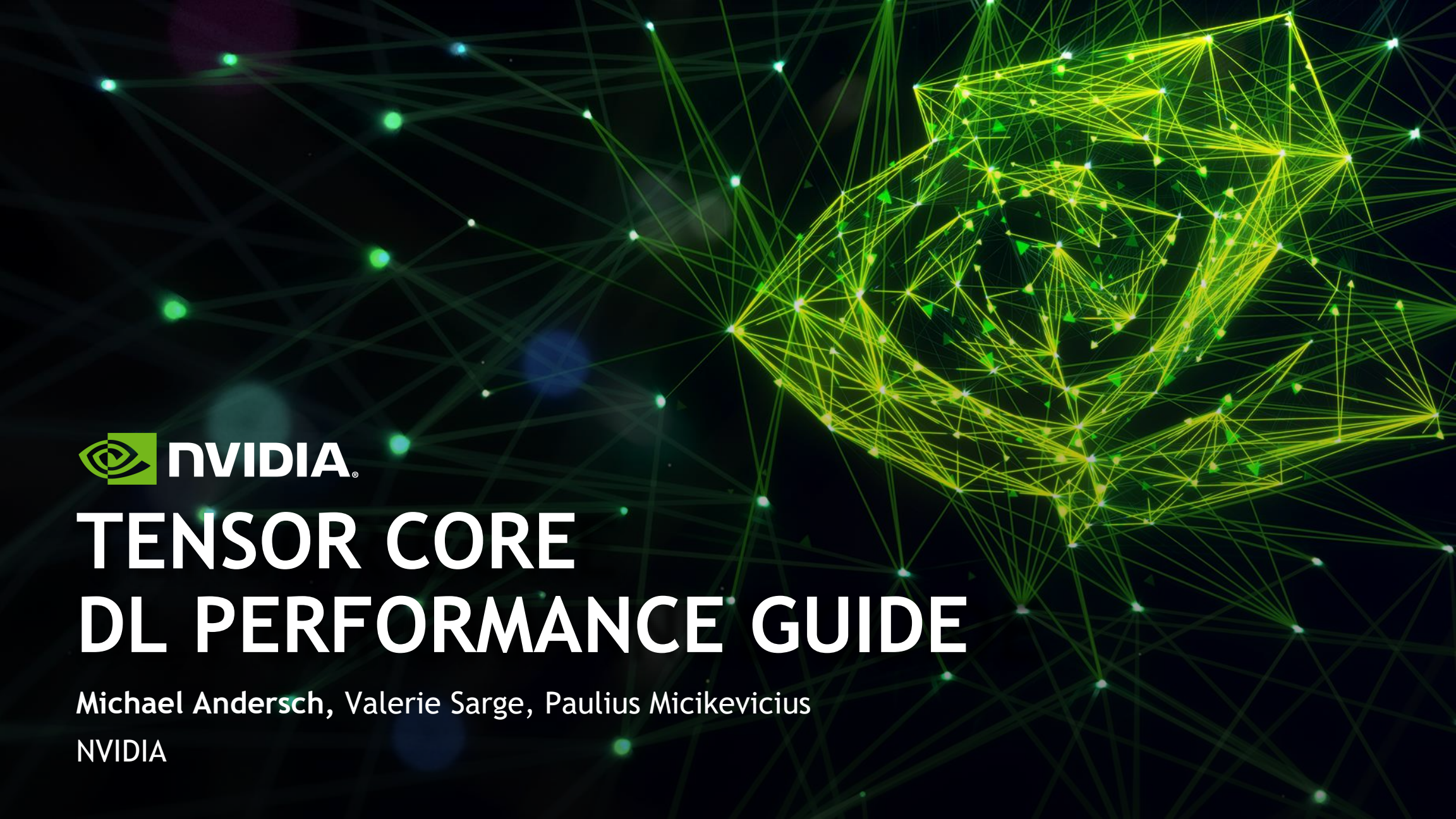


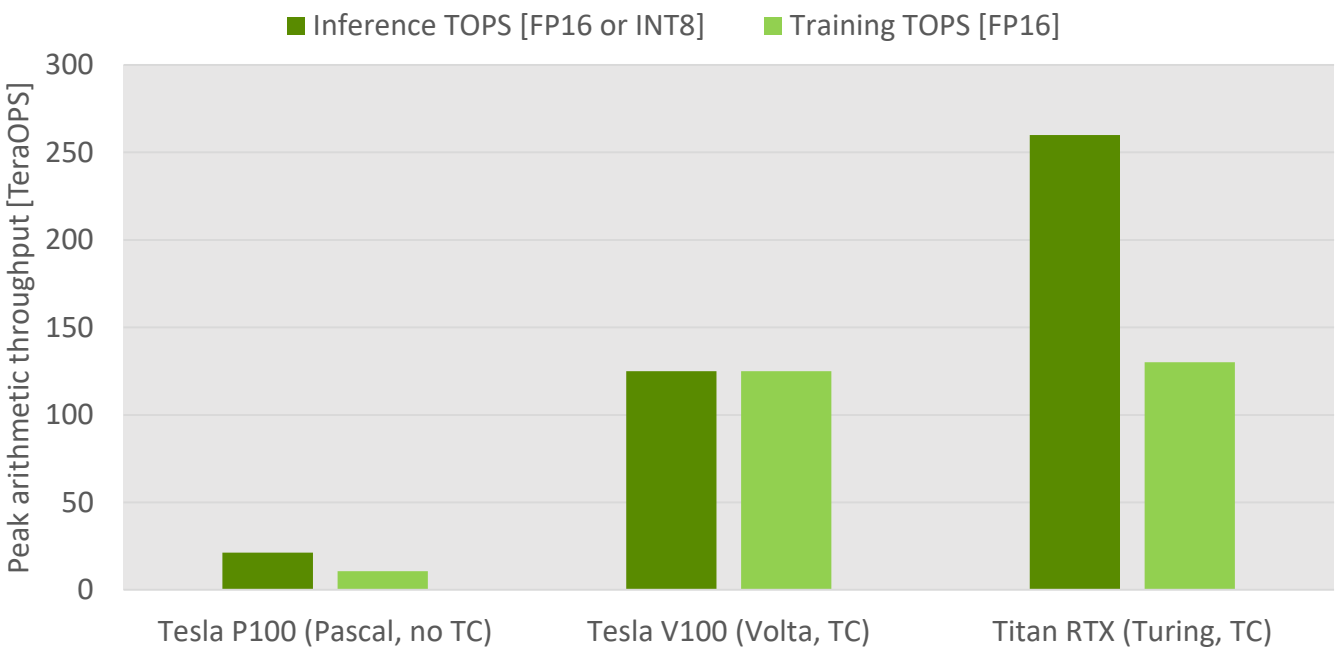
# TENSOR CORE DL PERFORMANCE GUIDE

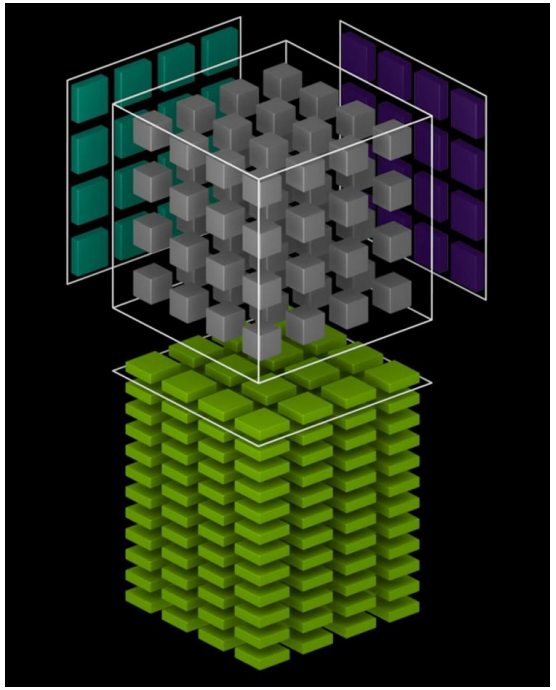
**Michael Andersch,** Valerie Sarge, Paulius Micikevicius

NVIDIA

# TENSOR CORES: BUILT TO ACCELERATE AI

## Available on NVIDIA Volta and Turing Tensor Core GPUs



This talk: Learn basic guidelines to best harness the power of Tensor Core GPUs!

# OUTLINE

1. Tensor Core refresher – what, how, why?
2. Reasoning about Deep Learning performance
3. Guidelines for ideal Tensor Core performance
4. Case studies

# TENSOR CORES: A REFRESHER

## Introduced on NVIDIA Volta V100 GPU

Tensor Cores are …

… special hardware execution units

… built to accelerate deep learning

… executing matrix multiply operations

Volta Tensor Cores

FP16/FP16 and FP16/FP32 modes

Turing Tensor Cores

+ INT8/INT32, INT4/INT32, INT1/INT32




NVIDIA

# HOW TO USE TENSOR CORES FOR TRAINING


NVIDIA cuDNN, cuBLAS, TensorRT

**Tensor Core Optimized Frameworks and Libraries**

Enable mixed precision training

S9143 - Mixed Precision Training of Deep Neural Networks

Easiest way: **AMP**

**A**utomatic **M**ixed **P**recision

S9998 - Automatic Mixed Precision in PyTorch

S91003 – MxNet Models Accelerated with Tensor Cores

S91029 - Automated Mixed-Precision Tools for TensorFlow Training

This talk: How to maximize perf once MP is enabled


DEEP LEARNING
PERFORMANCE
BASICS

# DOES <X> USE TENSOR CORES?

Or: Am I using TCs effectively? AKA: “Only 50 TFLOPS?!”

# GPU PERFORMANCE BASICS

## The GPU: a highly parallel, scalable processor

GPUs have processing elements (SMs), on-chip memories (e.g. L2 cache), and off-chip DRAM

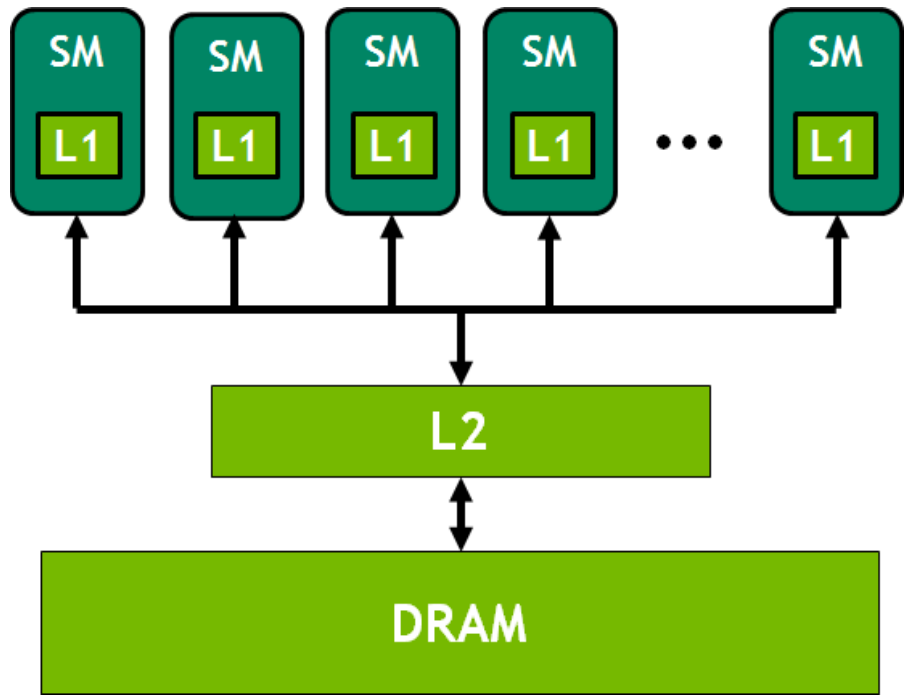
Tesla V100: 125 TFLOPS, 900 GB/s DRAM

What limits the performance of a computation?

$$time_{math\ operations} > time_{data\ movement}$$

$$\frac{FLOPS}{math\ throughput} > \frac{bytes}{memory\ bandwidth}$$

$$\frac{FLOPS}{bytes} > \frac{math\ throughput}{memory\ bandwidth}$$



# LIMITER ANALYSIS

## Lesson 1: Understand your performance limiters

Math limited if: $\frac{FLOPS}{bytes} > \frac{math\ throughput}{memory\ bandwidth}$

Left metric is algorithmic mix of math and memory ops called **arithmetic intensity**

Right metric is the processor's **ops/byte ratio** - e.g. V100 can execute 125/0.9=139 FLOPS/B

**Comparing arithmetic intensity to ops/byte ratio indicates what algorithm is limited by!**

| Operation | Arithmetic Intensity | Limiter |
|---|---|---|
| Residual addition | 0.166 | Memory |
| ReLU activation | 0.25 | Memory |
| Batch normalization | O(10) | Memory |
| Convolution | 1-10000+ | Memory/Math |

(assumes FP16 data)

# HOW TO CHECK IF TENSOR CORES ARE USED

## Simplest method: run GPU profiler

Run *nvprof* and look for [i|s|h][some numbers] in function names

volta_h884gemm_...

turing_fp16_s1688cudnn_fp16_...

But: not comprehensive

some kernels use TCs but don't follow this naming scheme

no trivial mapping back to neural network operations

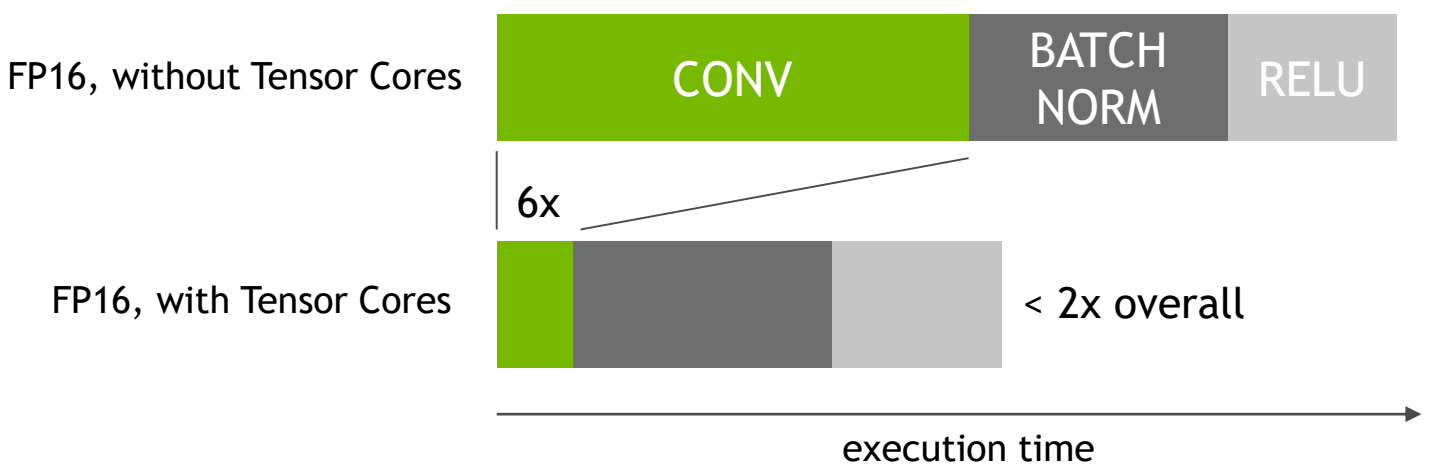
Useful as a first check: Am I using Tensor Cores, and are they close to being the top function?

# END-TO-END PERFORMANCE

## Lesson 2: Total Tensor Core speedup depends on memory limited time

The end-to-end network speedup depends on layer mix

Amdahl's law: if you speed up X% of your runtime, then the (1-X)% limit your overall speedup



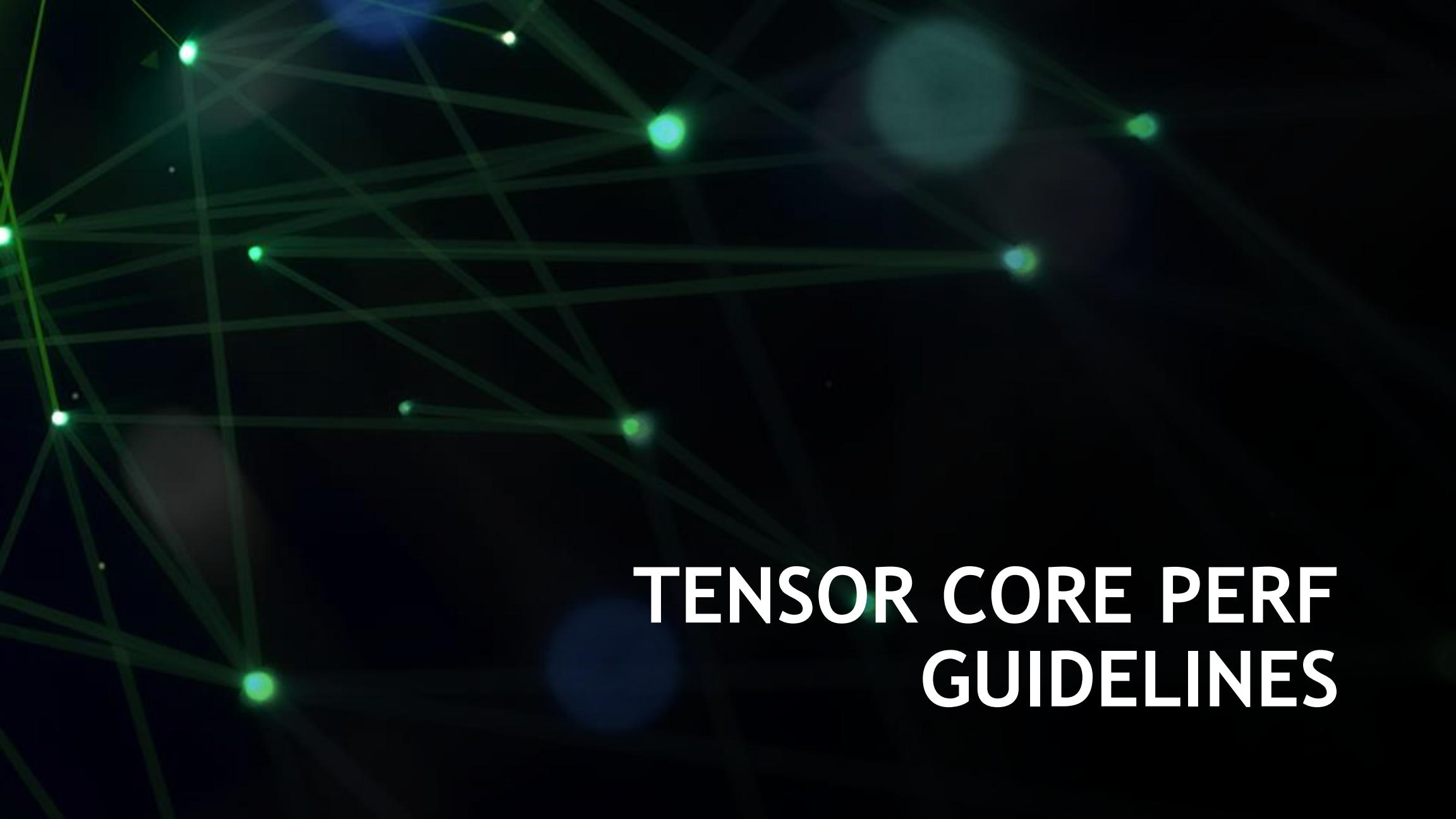
# GPU PERF BASICS: SUMMARY

## Before we dig into the details

Tensor Cores accelerate processing (not memory) by providing higher matrix math throughput

**Rules of thumb to remember**

1. Check arithmetic intensity against GPU ops/byte ratio to see if math or memory limited
2. End-to-end speedup from Tensor Cores depends on operation mix in the neural network
3. Use *nvprof* as a quick check to see if you are using Tensor Cores at all



# TENSOR CORE PERF GUIDELINES

# TENSOR CORE ACCELERATION

## Which operations *do* benefit?

*Dot product operations*

GEMMs (Dense/Linear/FullyConnected/…)

Convolutions

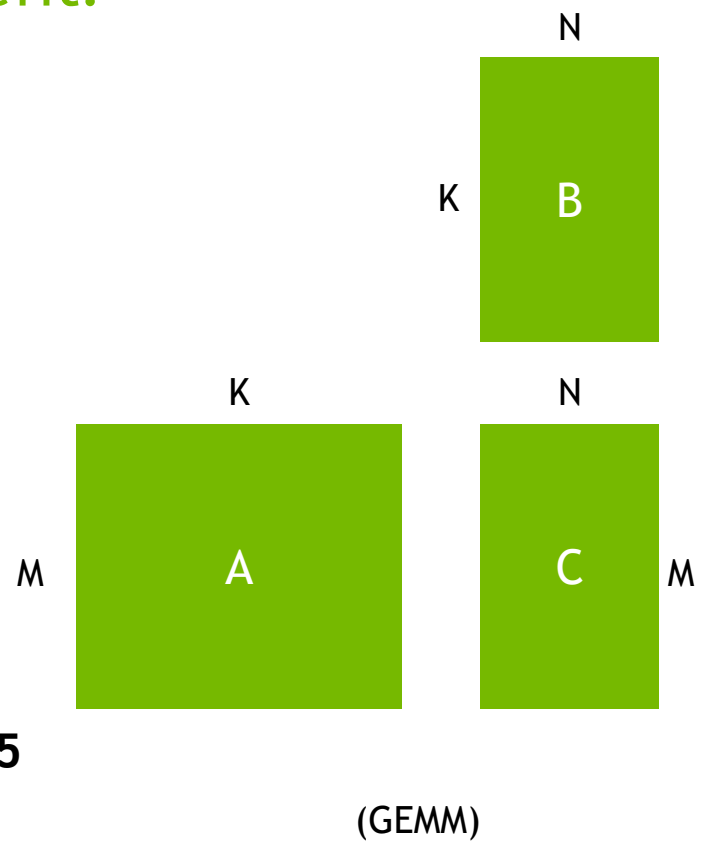
RNN/LSTM/GRU/…

Can be thought of as matrix-matrix multiplications

Arithmetic intensity = MNK/(MK+KN+MN)

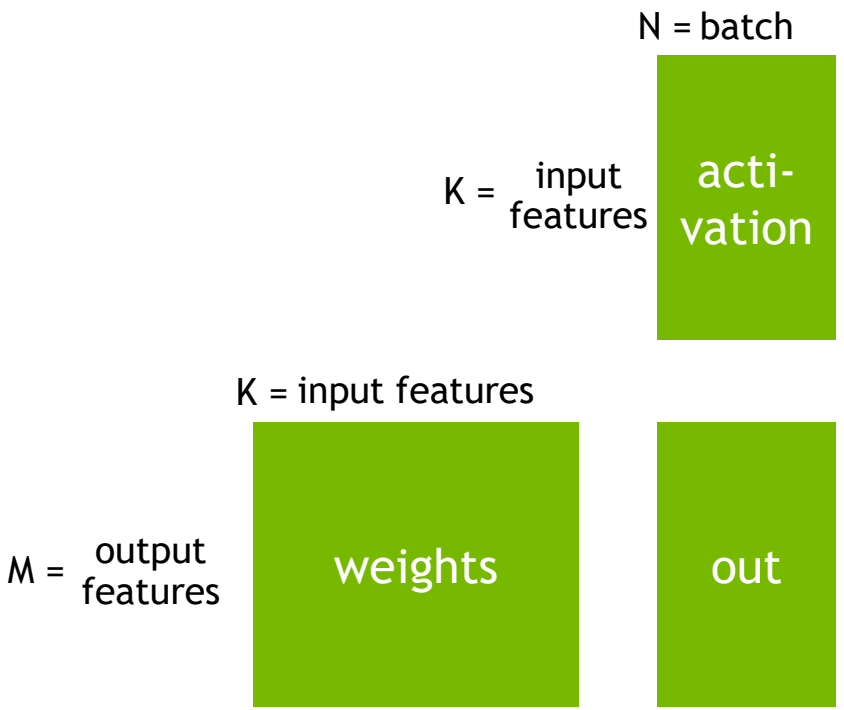
E.g. MxNxK = 4096x4096x4096: Arith. Intensity = **1365**

But: becomes BW bound if any dimension is small



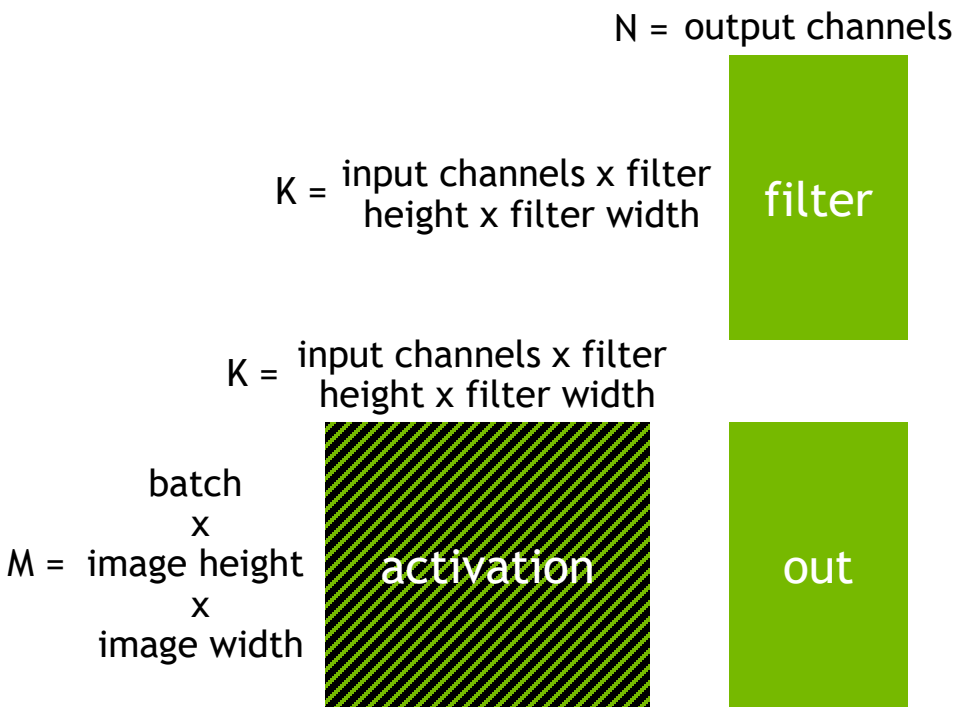
# DNN OPERATION MAPPING TO GEMM

## Forward pass mappings




**Fully Connected / Dense / Linear**

(PyTorch nn.Linear, TensorFlow swaps A and B)




**Convolution**

(*implicit* GEMM algorithm,
matrices are never actually created)

# BACKGROUND: TC-ACCELERATED GEMM

## Output matrix partitioned into thread block tiles

GPUs execute work by mapping computation to threads

Threads are grouped into thread blocks to cooperate

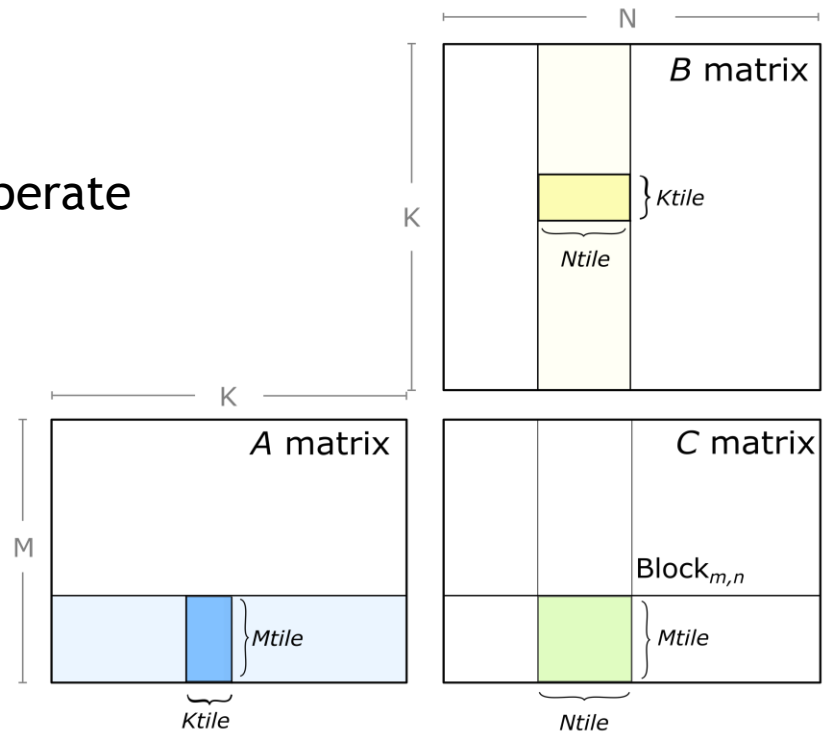
Thread blocks are scheduled onto GPU SMs

GEMM algorithm: blocks produce output matrix tiles

**Tiles require alignment for efficient access**

**If problem cannot be tiled cleanly, perf is lost**

**Smaller tiles are less efficient**



# FUNCTIONAL REQUIREMENTS

## Multiple-of-8 and multiple-of-16 rule

**Choose layer sizes as multiple of 8 (FP16) or 16 (INT8)**

Linear: inputs, outputs, batch size

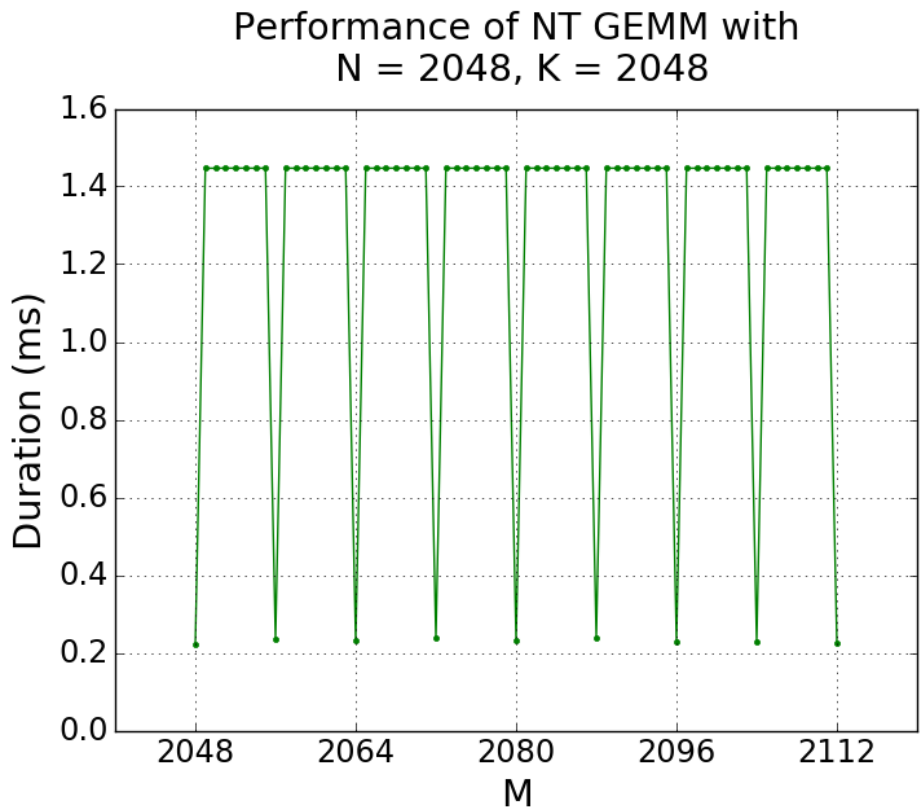
Convolution: input/output channels

RNNs: hidden, embedding, batch, vocabulary

Tensor Core speeds require efficient aligned data accesses to keep the cores fed

Hardware uses CUDA cores as fallback

4-8x slower than Tensor Cores

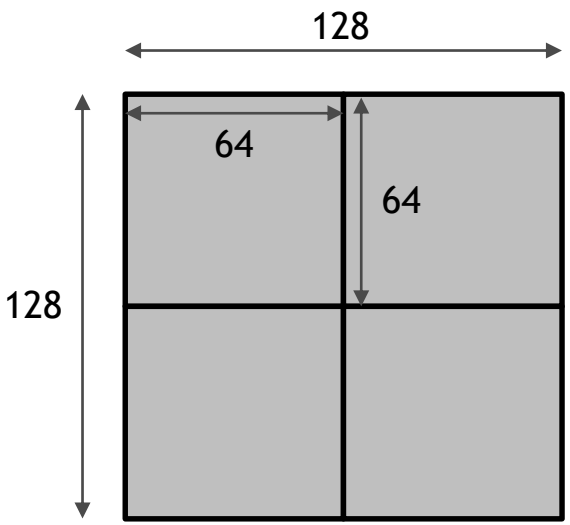



(Tesla V100-DGXS-16GB, cuBLAS 10.1)

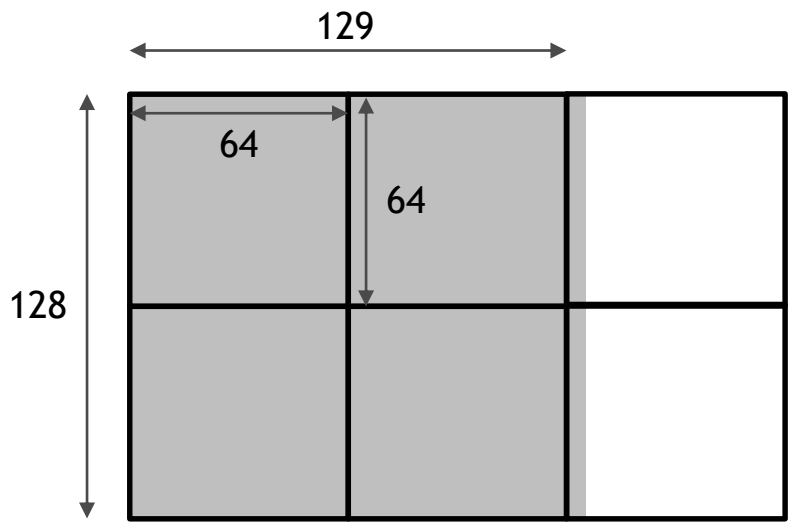
# PARALLELIZATION: TILE QUANTIZATION

## Dimensions quantize to tile boundaries

When the problem size does not cleanly divide into tiles, performance is lost




best case
4/4 tiles used
100% utilization

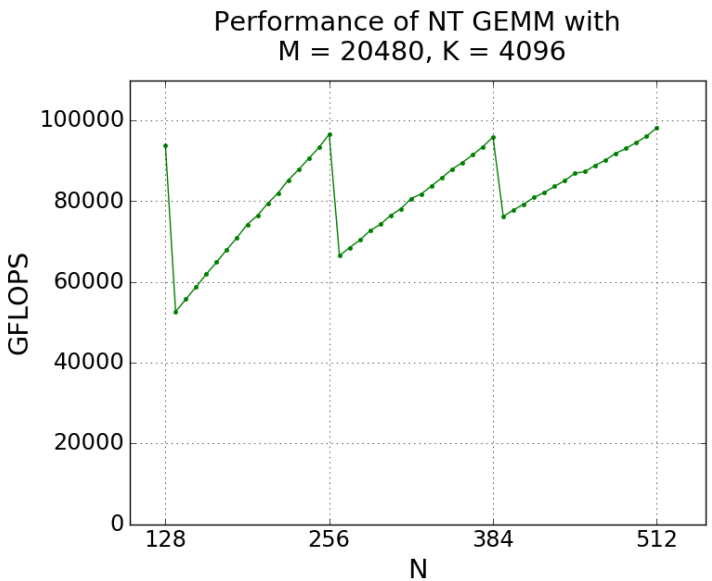


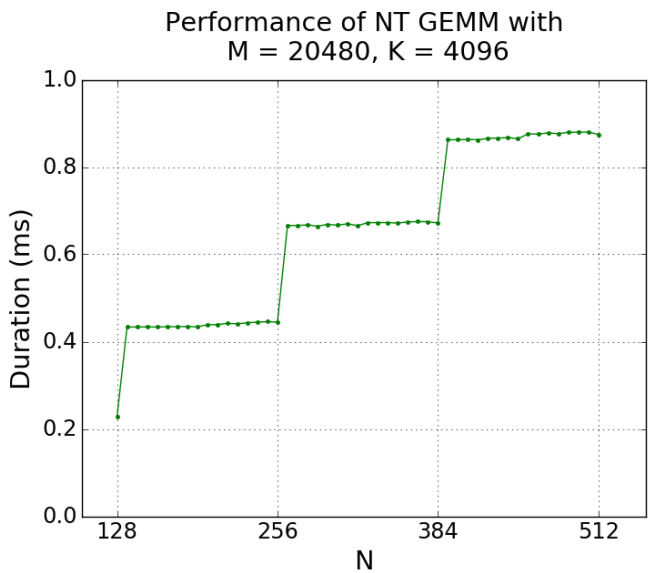

not-so-great case
~4/6 tiles used
67% utilization

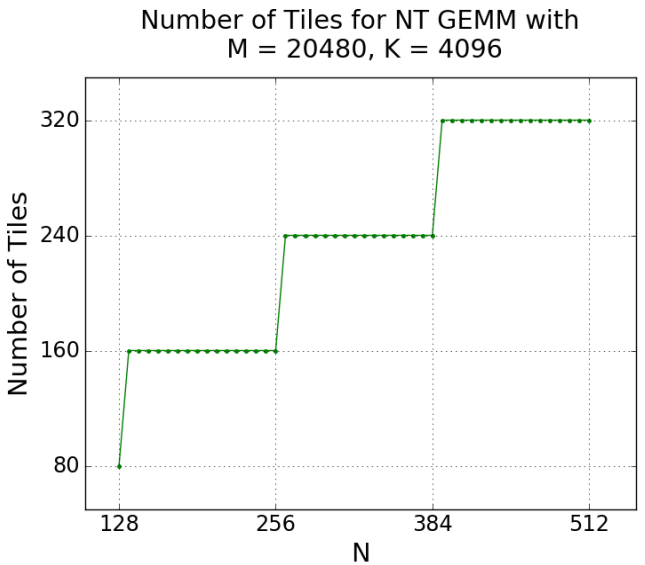
# PARALLELIZATION: TILE QUANTIZATION

## Dimensions quantize to tile boundaries

When the problem size does not cleanly divide into tiles, performance is lost



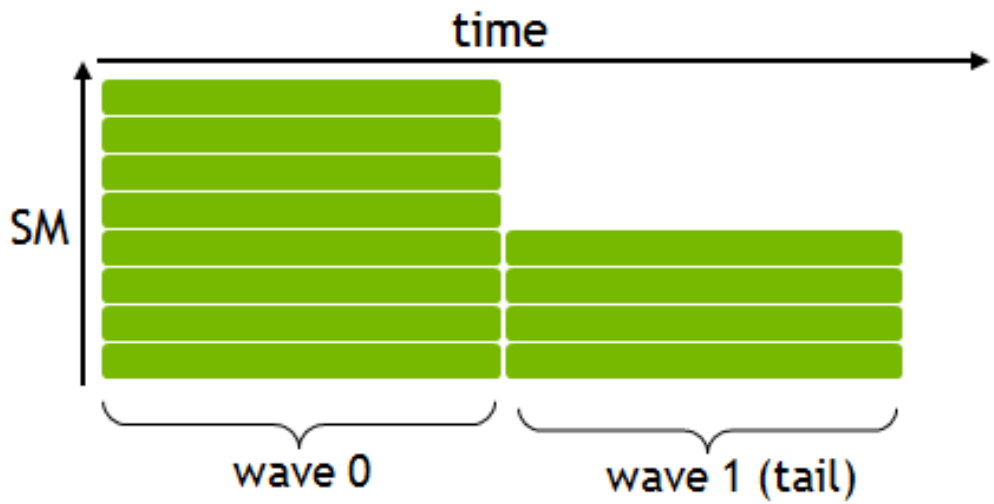




**Choosing dimensions to be multiples of 64 minimizes tile quantization (cuBLAS 10.1)**

# PARALLELIZATION: WAVE QUANTIZATION

## Number of tiles quantizes to the GPU size

Tiles are assigned to SMs, so performance is ideal when number of tiles is a multiple of SM count

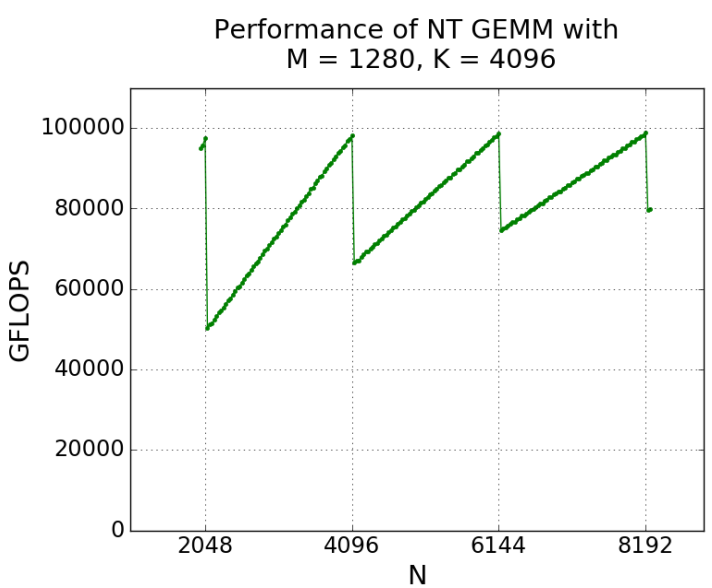


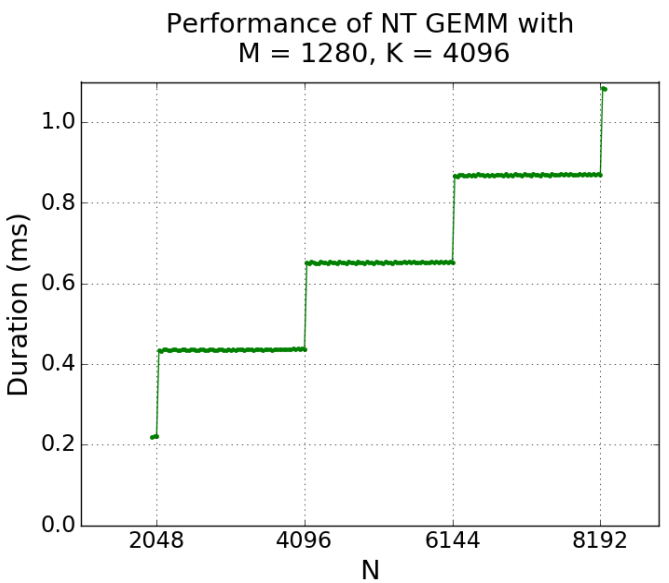

Example with 12 tiles on an 8-SM GPU, assuming 1 tile/SM
Second wave runs at 50% utilization
Overall computation runs at 75% utilization

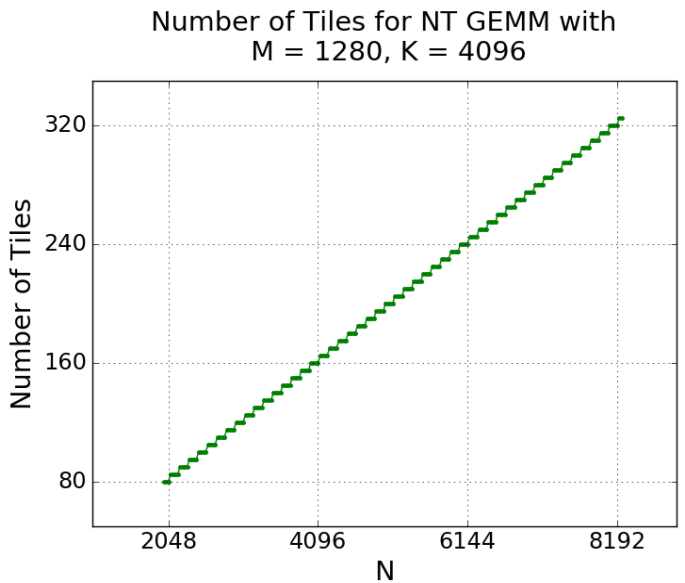
# PARALLELIZATION: WAVE QUANTIZATION

## Number of tiles quantizes to the GPU size

Tiles are assigned to SMs, so performance is ideal when number of tiles is a multiple of SM count




**It is useful to check the number of thread blocks created (by calculation or nvprof/nsight)**

# PARALLELIZATION: TILE EFFICIENCY

## Larger tiles are more bandwidth efficient, larger K amortizes overhead

Tiles are just smaller GEMMs – same data reuse principles

When tile's M and N are smaller …

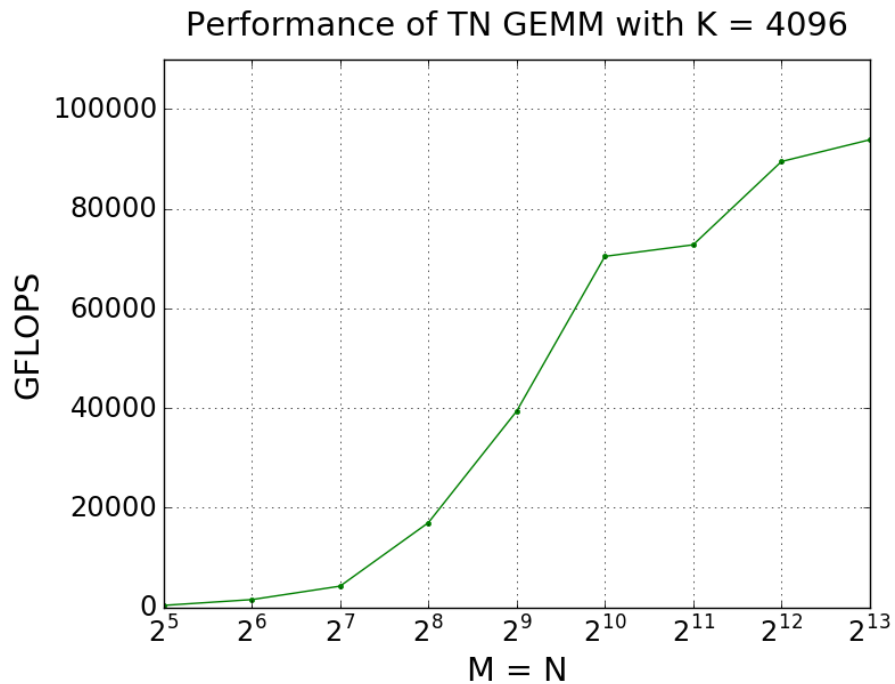
… less data reuse is captured in the tile

… more external bandwidth is required

Also, when tile's K is small …

… setup and teardown overheads dominate

In general, **larger operations perform better**




(Tesla V100-DGXS-16GB, cuBLAS 10.1)

# TENSOR CORE PERFORMANCE GUIDELINES

## If you only remember one slide from this presentation, use this one!

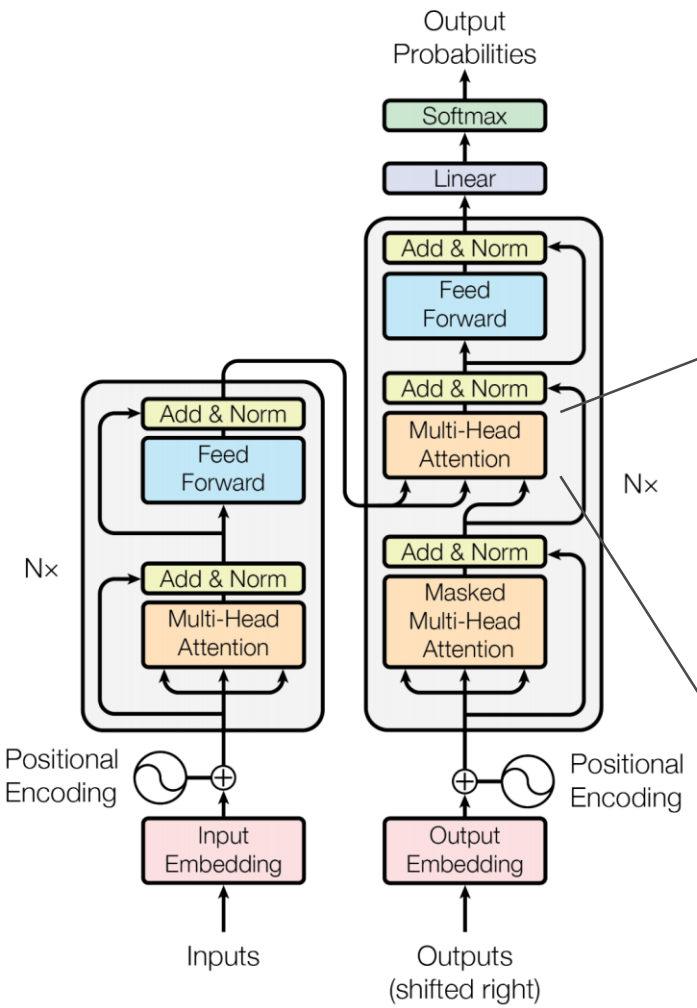
1. Satisfy requirements to enable Tensor Cores
    - For linear layers: input size, output size, batch size need to be multiples of 8 (FP16) / 16 (INT8)
    - For convolutions: input and output channel counts need to be multiples of 8 (FP16) /16 (INT8)
2. Ensure good Tensor Core GEMM efficiency
    - Choose the above dimensions as multiples of 64/128/256
    - (if the total number of tiles is small) Ensure that the tile count is a multiple of the SM count
3. Be aware of bandwidth limited regimes
    - If any GEMM dimension is 128 or smaller, the operation is likely bandwidth limited

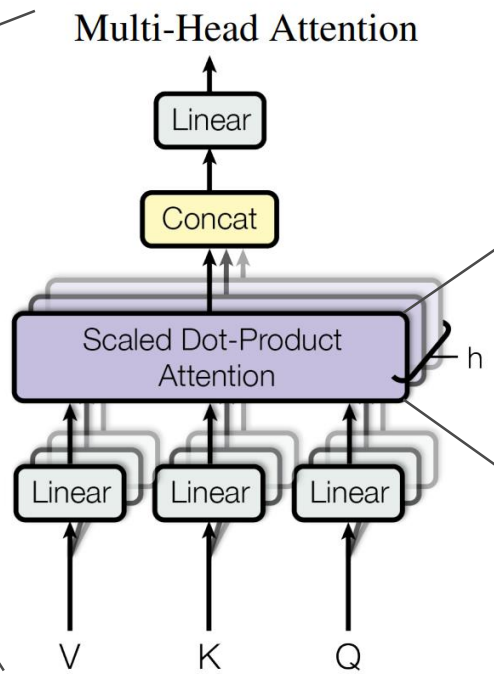
# CASE STUDY: TRANSFORMER

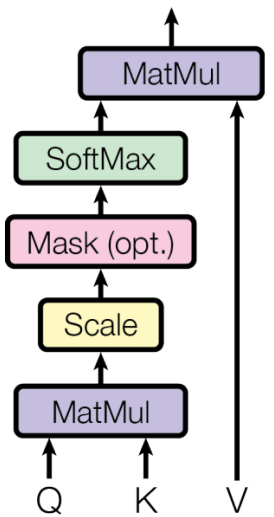
# CASE STUDY: TRANSFORMER

## From “Attention is all you need”

Transformers perform neural machine translation without suffering from RNN dependencies



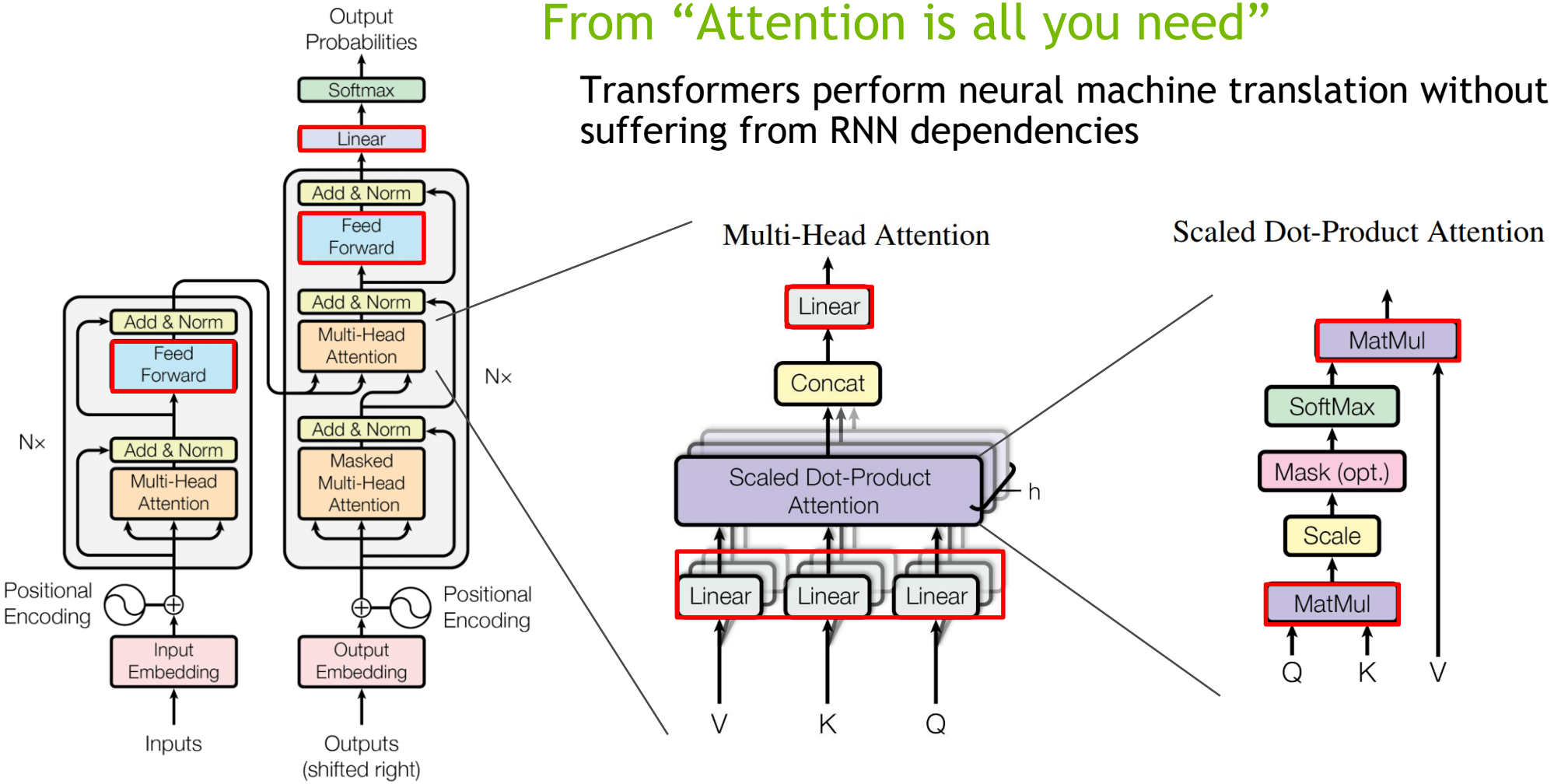



# CASE STUDY: TRANSFORMER

## From “Attention is all you need”

Transformers perform neural machine translation without suffering from RNN dependencies

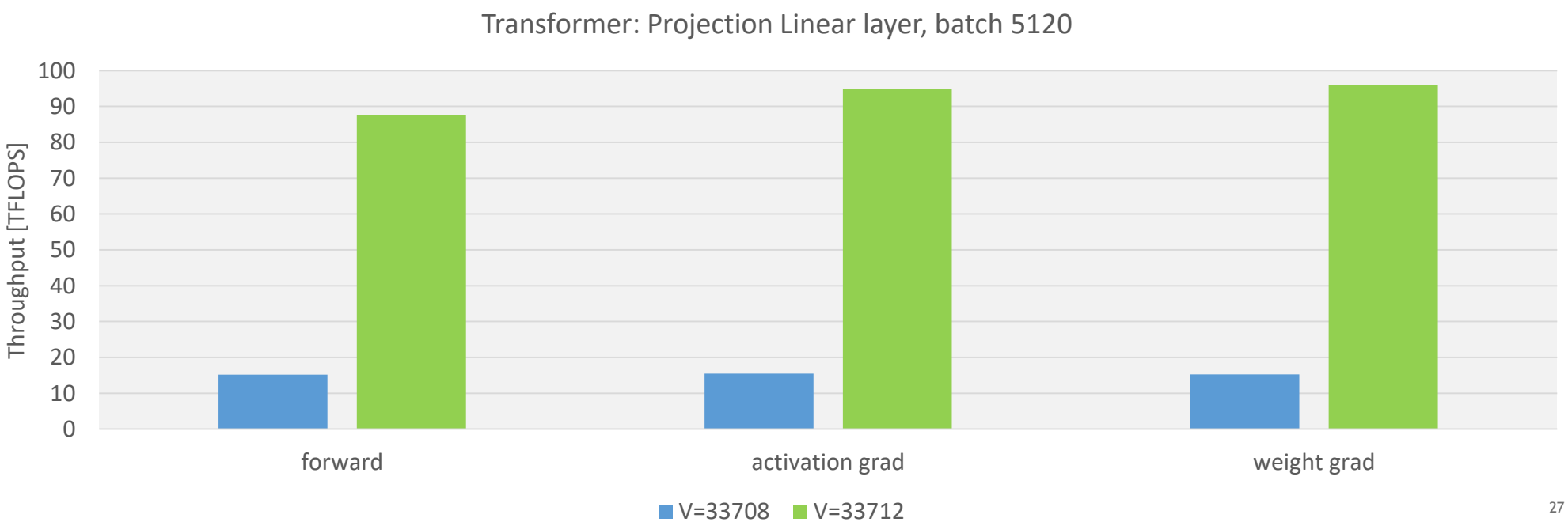


# CASE STUDY: TRANSFORMER

## From “Attention is all you need”

Step 1: Pad vocabulary to multiple of 8 to ensure TC usage in projection layer

Vocabulary size maps to M dimension in projection layer



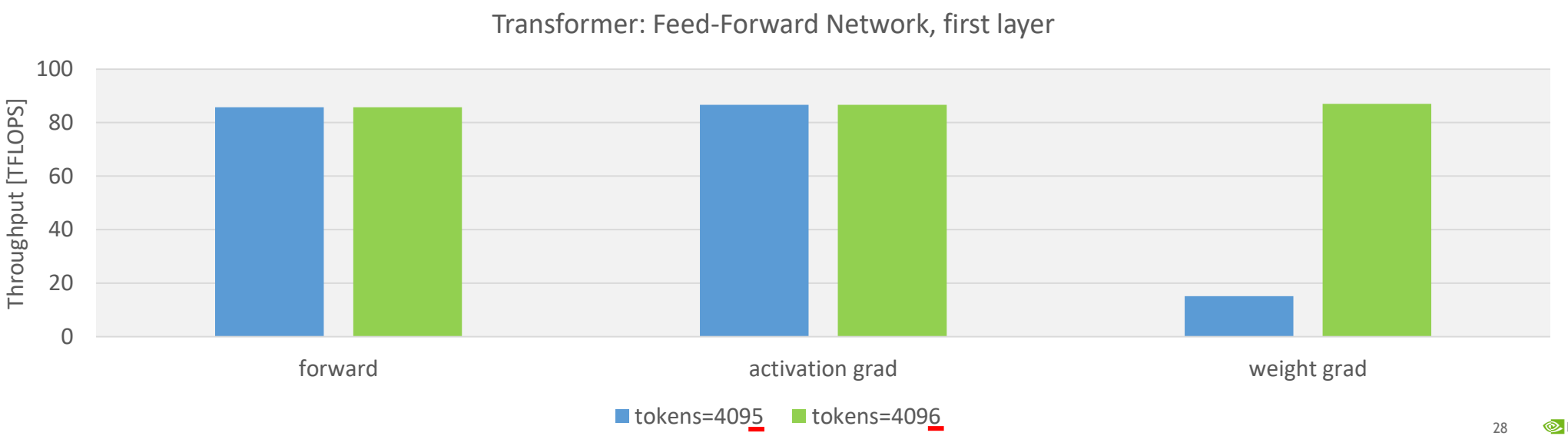
# CASE STUDY: TRANSFORMER

## From “Attention is all you need”

Step 2: Pad input sequence data to multiple of 8 to ensure TC usage in all other layers

Sequence length maps to M/N dimensions in attention layers

Sequence length * number of sentences maps to N dimension in most layers

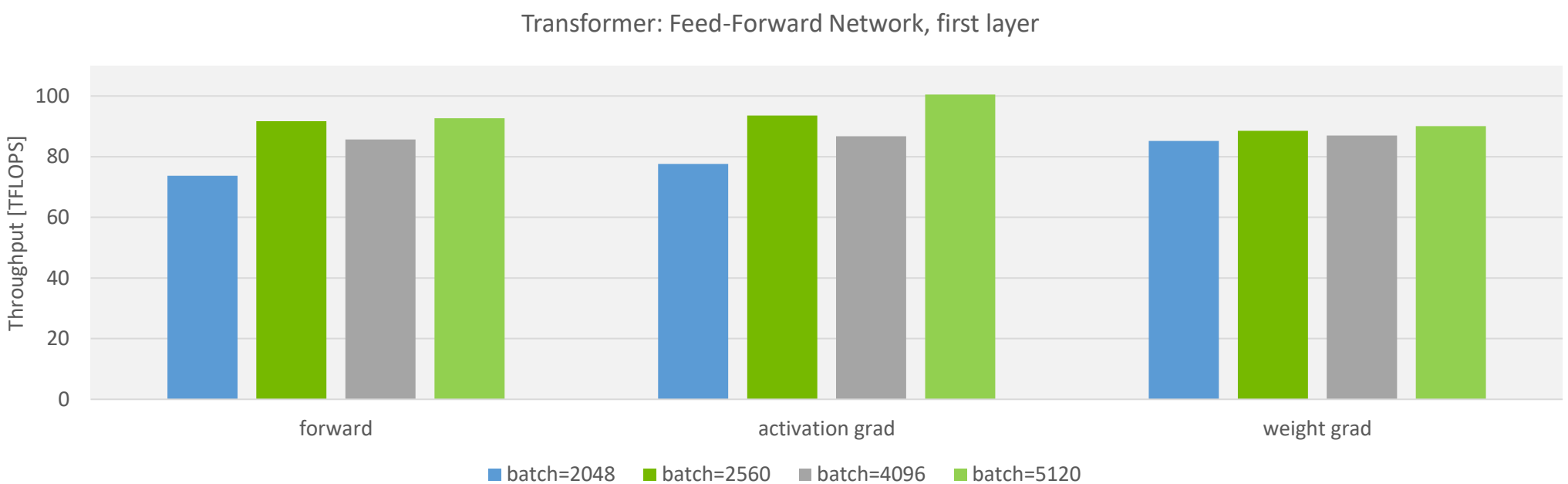


# CASE STUDY: TRANSFORMER

## From “Attention is all you need”

Step 3: Choose token count per batch such that tile count is multiple of SM count (80 here)

E.g. 5120 instead of 4096, 2560 instead of 2048, …



# SUMMARY

# SUMMARY: TENSOR CORE GUIDELINES

Tensor Core GPUs provide considerable deep learning performance

Following a few simple guidelines can maximize delivered performance

- Ensure key dimensions are multiples of 8 (FP16) or 16 (INT8)
- Choose dimensions to avoid tile and wave quantization where possible
- Up to a point, larger dimensions lead to higher efficiency

Visit the permanent online version of this guide (ETA early April)

https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html

# RESOURCES

# TENSOR CORES

## For more information

Volta V100 whitepaper

Turing whitepaper

Mixed-precision training guide

Tensor Core technology webpage

Programming Tensor Cores blog post

# DNN OPERATION MAPPING TO GEMM

## All pass mappings

| Operation | Phase | GEMM "M" | GEMM "N" | GEMM "K |
|---|---|---|---|---|
| FC/Linear | Forward | Output features | Batch size | Input features |
| | Data grad | Input features | Batch size | Output features |
| | Weight grad | Input features | Output features | Batch size |
| Conv | Forward | Batch x iHeight x iWidth | Output channels | Input channels x fHeight x fWidth |
| | Data grad | Batch x iHeight x iWidth | Input channels | Output channels x fHeight x fWidth |
| | Weight grad | Input channels x fHeight x fWidth | Output channels | Batch x iHeight x iWidth |

# TENSOR CORE THROUGHPUTS

## On Volta and Turing GPUs (except TU11x), MACs/SM/CLK

| | CUDA Cores | | | | Tensor Cores | | | |
|---|---|---|---|---|---|---|---|---|
| GPU | FP64 | FP32 | FP16 | INT8 | FP16 | INT8 | INT4 | INT1 |
| Volta | 32 | 64 | 128 | 256 | 512 | | | |
| Turing | 2 | 64 | 128 | 256 | 512 | 1024 | 2048 | 8192 |

# CONVOLUTION DATA LAYOUTS

## With Tensor Cores, NHWC layout is faster than NCHW layout

4D tensor data can be laid out two ways

- “channel-first” or NCHW
- “channel-last” or NHWC

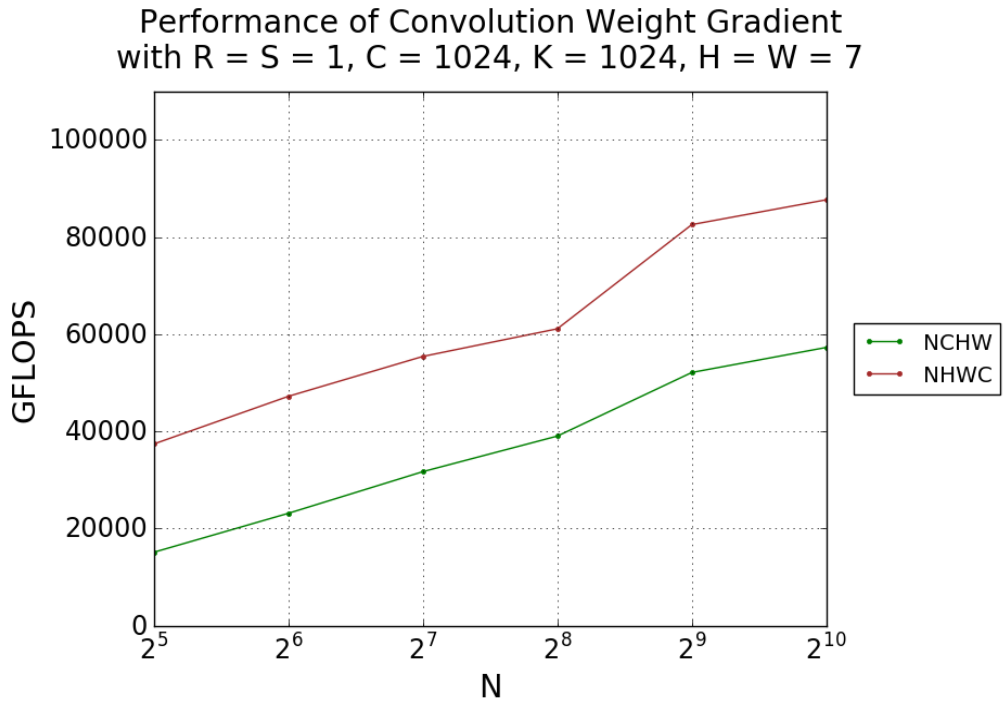
TC convolutions natively process NHWC tensors

- NCHW data incurs an extra transpose

Native NHWC support in MxNet and TF (via XLA)

- PyTorch support in development

**Enable NHWC layout when possible**




(Tesla V100-DGXS-16GB, cuBLAS 10.1)


NVIDIA