TENSOR CORE PERFORMANCE: THE ULTIMATE GUIDE

Valerie Sarge, Michael Andersch

NVIDIA

OUTLINE

Understanding performance limits: math and memory

Our recommendations for getting the most out of your GPU

- Enable Tensor Cores
- Understand the calculations being done
- Choose dimensions to fill the GPU efficiently
- Pick the best implementation for your situation

(…and see the guide for more!)

https://docs.nvidia.com/deeplearning/performance/index.html

HARDWARE ACCELERATION FOR ML

Why Tensor Cores?

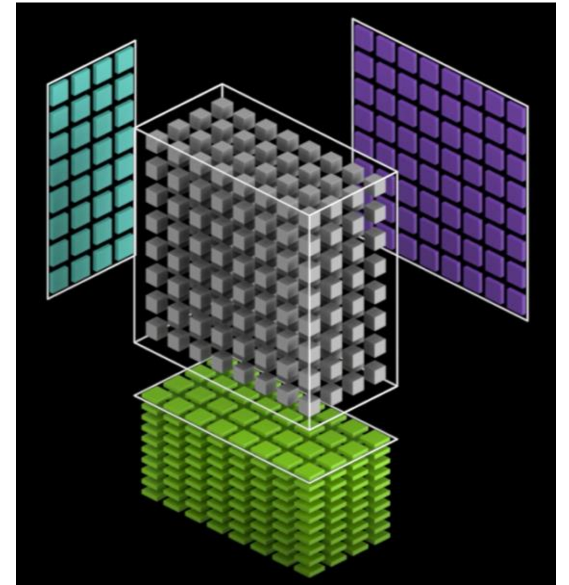
Tensor Cores are specialized hardware for deep learning

- Perform matrix multiplies quickly
- Tensor Cores are available on Volta, Turing, and NVIDIA A100 GPUs

NVIDIA A100 GPU introduces Tensor Core support for new datatypes (TF32, Bfloat16, and FP64)

Deep learning calculations benefit, including:

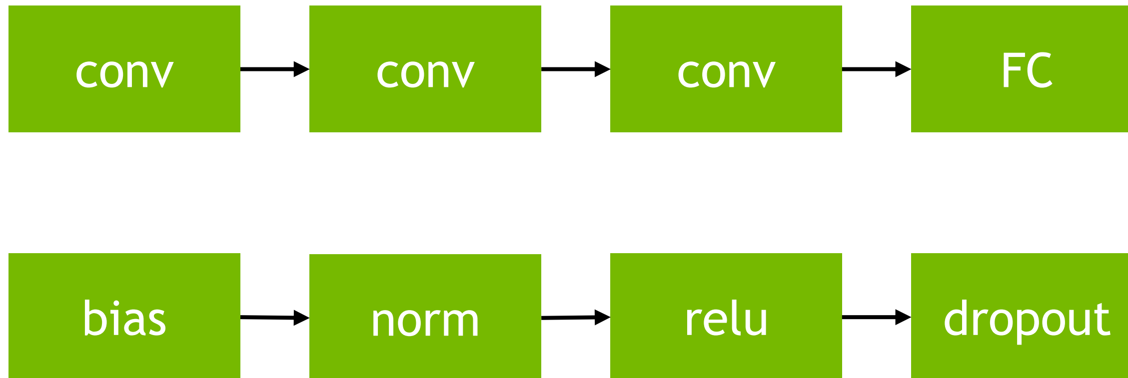
- Fully-connected / linear / dense layers
- Convolutional layers
- Recurrent layers



UNDERSTANDING
PERFORMANCE

WHAT LIMITS PERFORMANCE?

Math vs Memory (vs Latency)

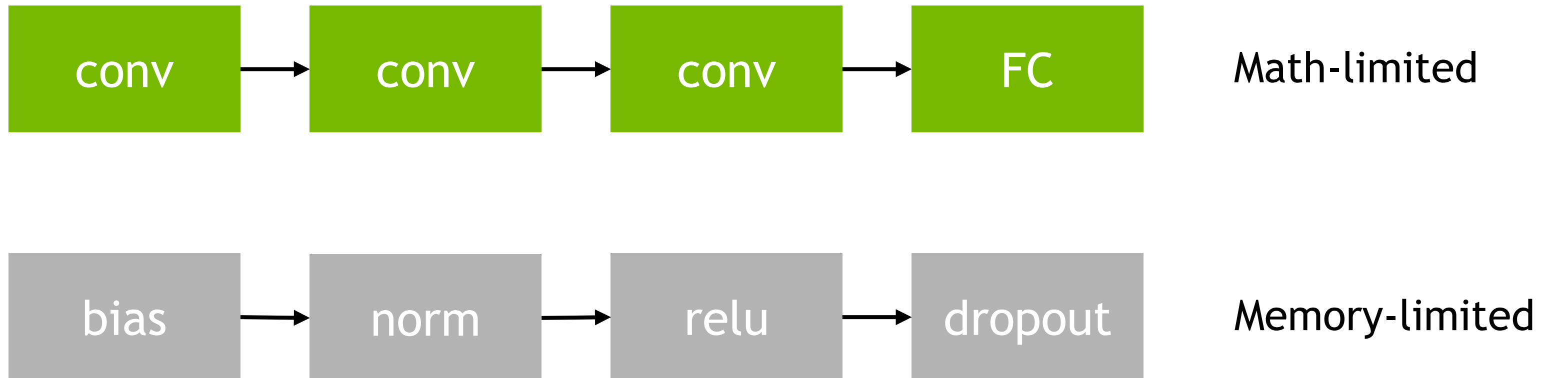


Which series of operations is most likely to effectively use Tensor Cores?

NVIDIA

WHAT LIMITS PERFORMANCE?

Math vs Memory (vs Latency)



Math-heavy ops (like convolutional, fully-connected, and recurrent layers) tend to be limited by calculation speed and thus **benefit from Tensor Cores**

Those with less calculation (like bias, normalization, activation, and dropout layers) tend to be limited by memory access speed and thus **do not benefit from Tensor Cores**

WHAT LIMITS PERFORMANCE?

GPU Basics

Calculation is done in parallel on streaming multiprocessors (SMs)

A100: **19.5 dense TFLOPS** for FP32, no Tensor Cores

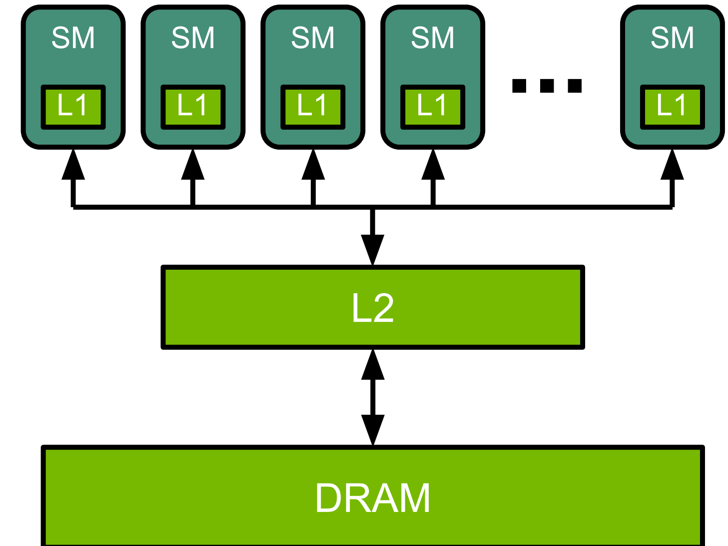
156 dense TFLOPS for TF32, with Tensor Cores

312 dense TFLOPS for FP16, with Tensor Cores

Data and instructions are accessed from DRAM through the shared L2 cache

A100: **1.555 TB/s** from DRAM

L2 cache is faster, but space is limited



NVIDIA

ARITHMETIC INTENSITY

At the operation level

Math-limited if time spent on math is greater than time spent on fetching from memory

$$T_{math} > T_{mem}$$

$$\frac{\#\ ops}{math\ bandwidth} > \frac{\#\ bytes}{mem\ bandwidth}$$

Or equivalently, if

$$\frac{\#\ ops}{\#\ bytes} > \frac{math\ bandwidth}{mem\ bandwidth}$$

NVIDIA.

ARITHMETIC INTENSITY

For a fully-connected layer

For FP16 inference with a FC layer with 4096 inputs and outputs and N_{batch} batch size:

$$Arithmetic\ Intensity = \frac{\#\ ops}{\#\ bytes} = \frac{(N_{out} \cdot N_{batch}) \cdot (N_{in}) \cdot (2)}{(N_{in} \cdot N_{batch} + N_{in} \cdot N_{out} + N_{out} \cdot N_{batch}) \cdot \frac{\#\ bytes}{element}}$$

$$= \frac{4096 \cdot N_{batch}}{4096 + 2 \cdot N_{batch}}\ FLOPS/B$$

By comparison, for A100 (still with FP16):

$$GPU\ FLOPS/B\ ratio = \frac{math\ bandwidth}{mem\ bandwidth} = \frac{312\ TFLOPS}{1.555\ TB/s} = 201\ FLOPS/B$$

NVIDIA.

ARITHMETIC INTENSITY

At the operation level

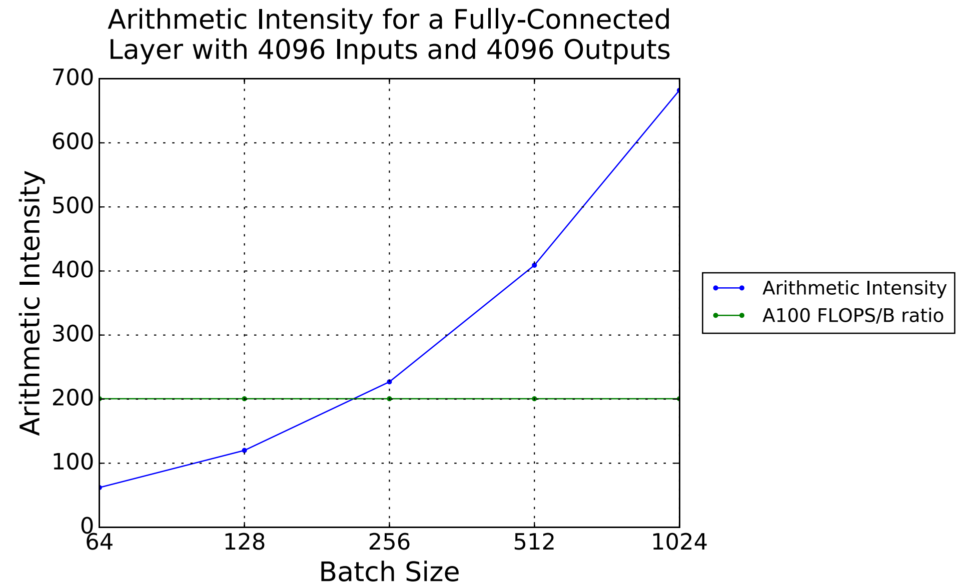
Inference with this fully-connected layer can be limited by either math or memory, depending on batch size

Math-limited when:

$$\frac{\#\ ops}{\#\ bytes} > \frac{math\ bandwidth}{mem\ bandwidth}$$

or

$$\frac{4096 \cdot N_{batch}}{4096 + 2 \cdot N_{batch}} > 201\ FLOPS/B$$



NVIDIA.

ARITHMETIC INTENSITY

At the operation level

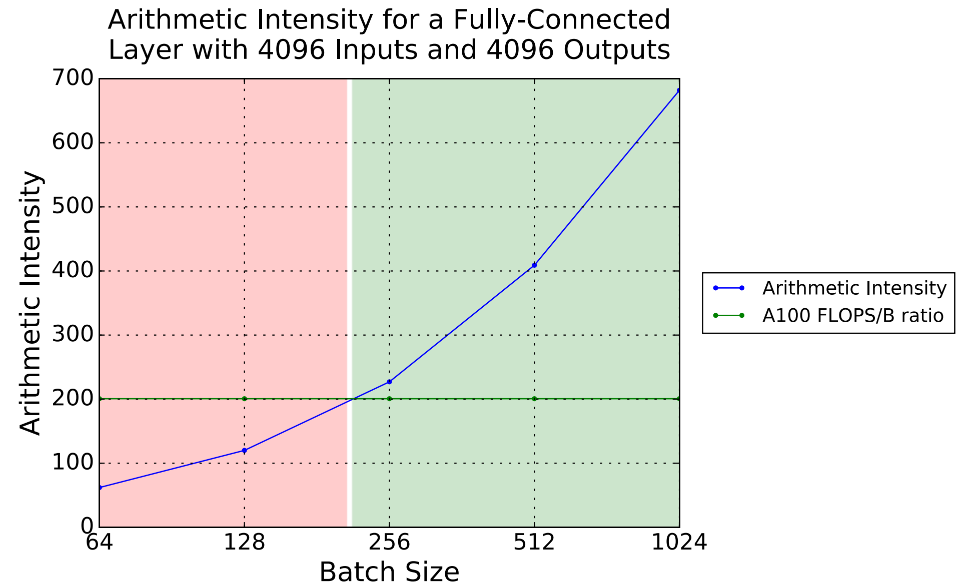
Inference with this fully-connected layer can be limited by either math or memory, depending on batch size

Math-limited when:

$$\frac{\# \ ops}{\# \ bytes} > \frac{math \ bandwidth}{mem \ bandwidth}$$

or

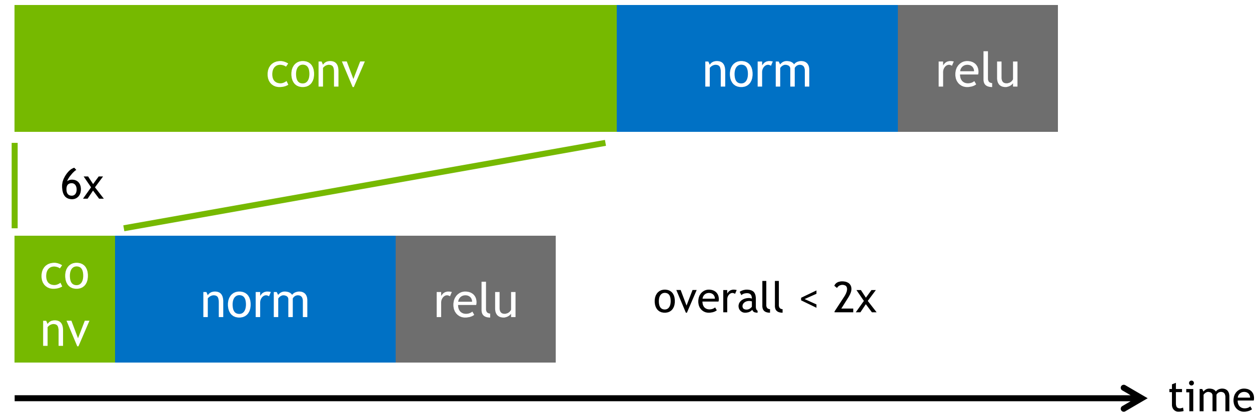
$$\frac{4096 \cdot N_{batch}}{4096 + 2 \cdot N_{batch}} > 201 \ FLOPS/B$$



NVIDIA.

END-TO-END PERFORMANCE

At the model level



Amdahl's law: speeding up one part of an end-to-end computation can only have so much effect on total execution time

We focus on calculation-heavy layers because they often comprise a majority of execution time

But it's also important to be aware of time spent on memory-limited layers!

NVIDIA.

KEY IDEAS SO FAR

GPUs perform calculations in parallel with SMs and access memory through a shared L2 cache

- Calculation speed is best when math can be split evenly between SMs
- Memory speed is best when data is reused from the L2 cache

Models are composed of math-limited and memory-limited operations

- Tensor Cores effectively speed up math-limited ops
- (There are options for speeding up memory-limited ops as well!)

Arithmetic intensity is a good first-order estimate of whether an op is math-limited

End-to-end performance is affected by both math-limited and memory-limited ops

GETTING THE MOST OUT OF YOUR GPU

MATRIX MULTIPLY ABSTRACTION

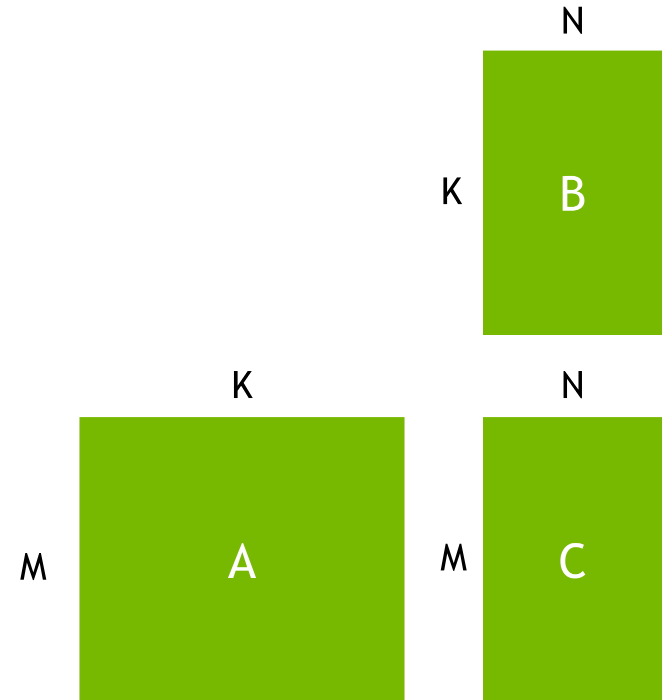
How do we represent DL operations?

Tensor Cores accelerate dot-product operations

- Fully-connected layers
- Convolutional layers
- Recurrent layers

These can also be thought of as matrix multiplies

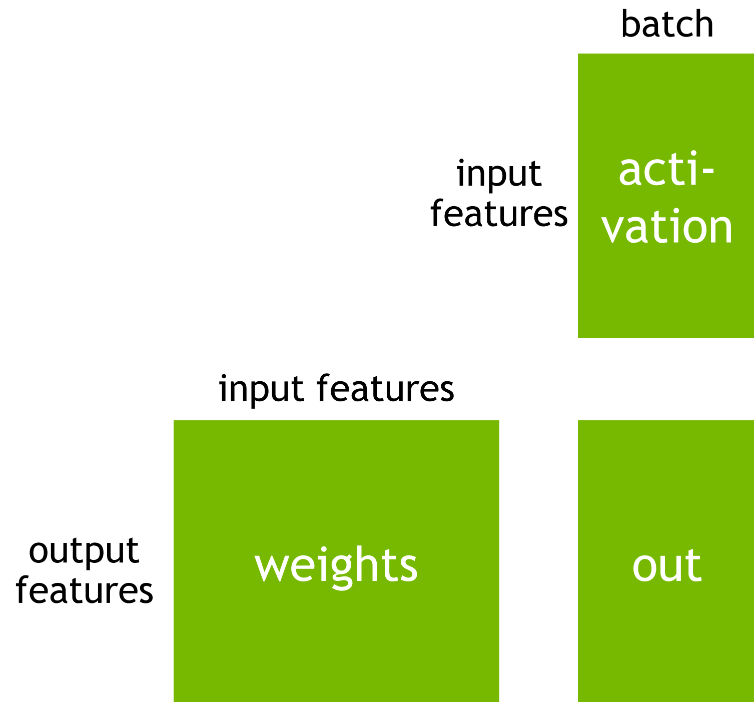
- Often not literally
- “Implicit” matrix multiplies; math is equivalent to a matrix multiply, but input and output matrices are not explicitly created in memory



NVIDIA.

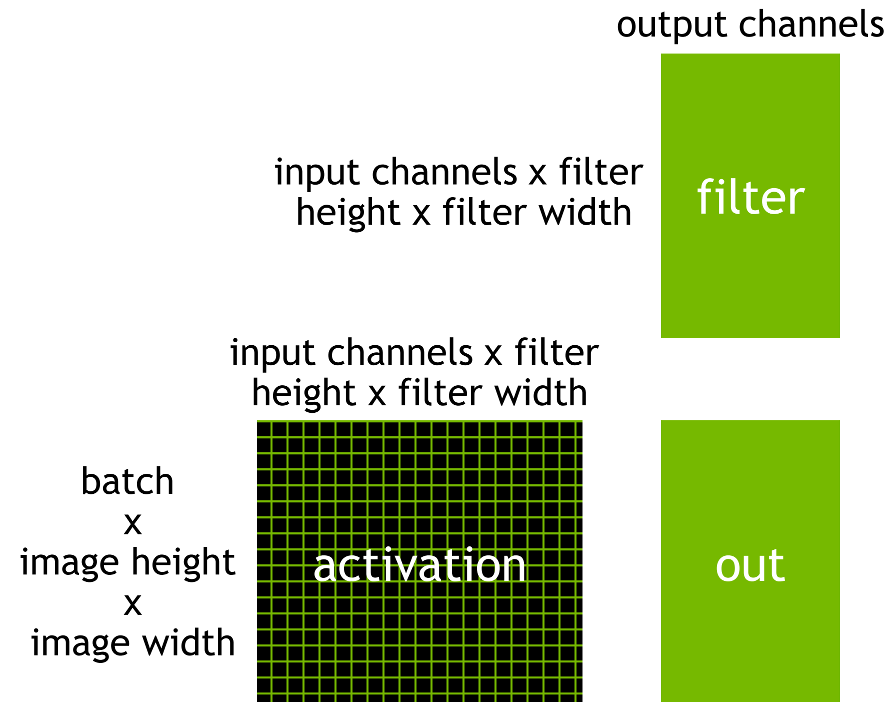
MATRIX MULTIPLY ABSTRACTION

For fully-connected and convolutional layers



Fully Connected / Dense / Linear

(PyTorch mappings, TensorFlow swaps weights and activations)



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

NVIDIA

ENABLING TENSOR CORES

Alignment and functional requirements

For cuBLAS:

- Performance is better when dimensions (M, N, and K) are multiples of 128 bits
- For cuBLAS 11.0 and higher, Tensor Cores can be used regardless

For cuDNN:

- Performance is better when dimensions (for convolution, input and output channel counts) are multiples of 128 bits
- For cuDNN 7.6.3 and higher, dimensions will be automatically padded to allow Tensor Cores to be enabled

NVIDIA.

ENABLING TENSOR CORES

Alignment and functional requirements

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 better perf with multiples of 16)
FP16	Multiples of 8	Always (but better perf with multiples of 8)
TF32	N/a	Always (but better perf with multiples of 4)

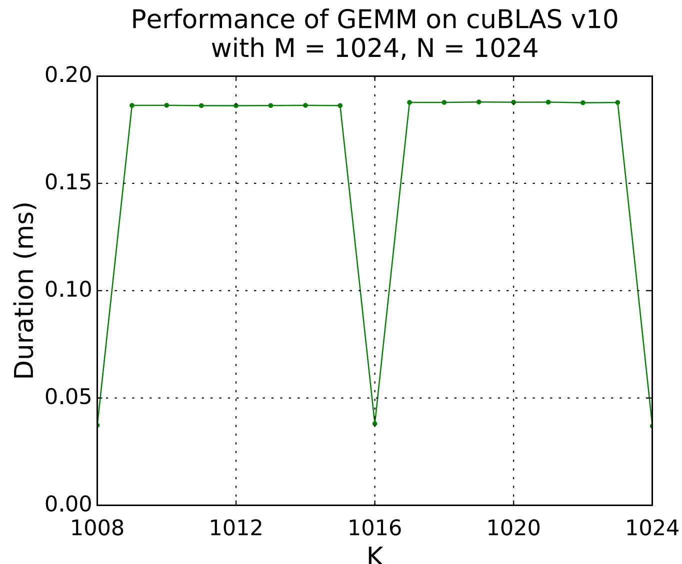
In practice, the requirements can be less strict than this, but following these alignments for all dimensions ensures Tensor Cores are enabled and running efficiently

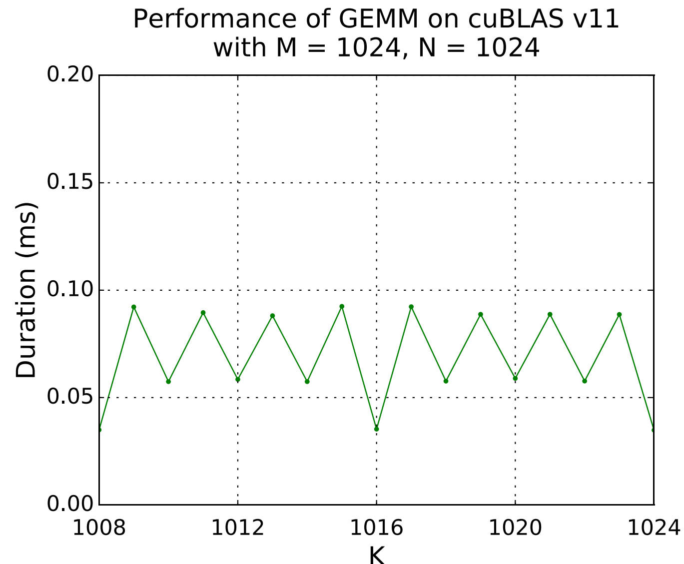
ENABLING TENSOR CORES

Alignment and functional requirements

An example: calculations are fastest (durations are lowest) when K is divisible by 8

GEMMs with K not divisible by 8 show 2-4x speedup here with cuBLAS 11 (leveraging Tensor Cores) compared to cuBLAS 10





FP16 precision. Tesla V100-DGXS-16GB.

AM I USING TENSOR CORES?

Check kernel names

View kernel names and stats with **nsys profile <application>** (or **nvprof <application>**)

Kernels that use Tensor Cores tend to look like:

volta_fp16_s884cudnn_fp16_...

turing_fp16_s1688gemm_fp16_...

ampere_h16816gemm_...

ampere_xmma_implicit_gemm_f16f16f16_..._16x8x16_...

cutlass_tensorop_f16_s1688gemm_f16_...

This isn’t universal- some kernels have names that don’t follow these forms- but it’s useful as a first check

NVIDIA

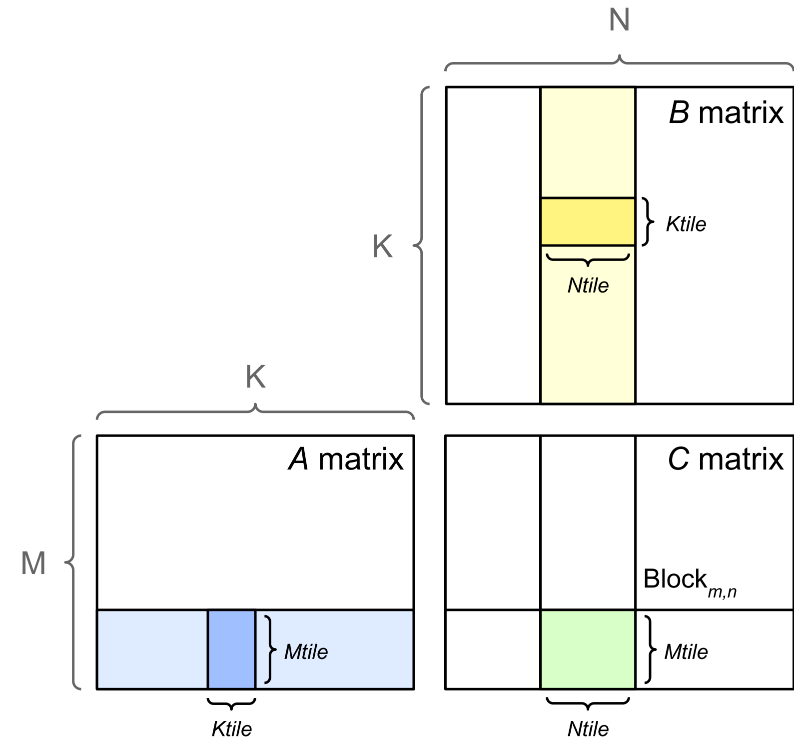
EFFICIENT PARALLELIZATION

How do GPUs split up operations?

Threads are grouped into thread blocks that work cooperatively

Thread blocks produce tiles of the output matrix

Tiled outer product approach: sum partial products over the K dimension to complete the output tile



NVIDIA

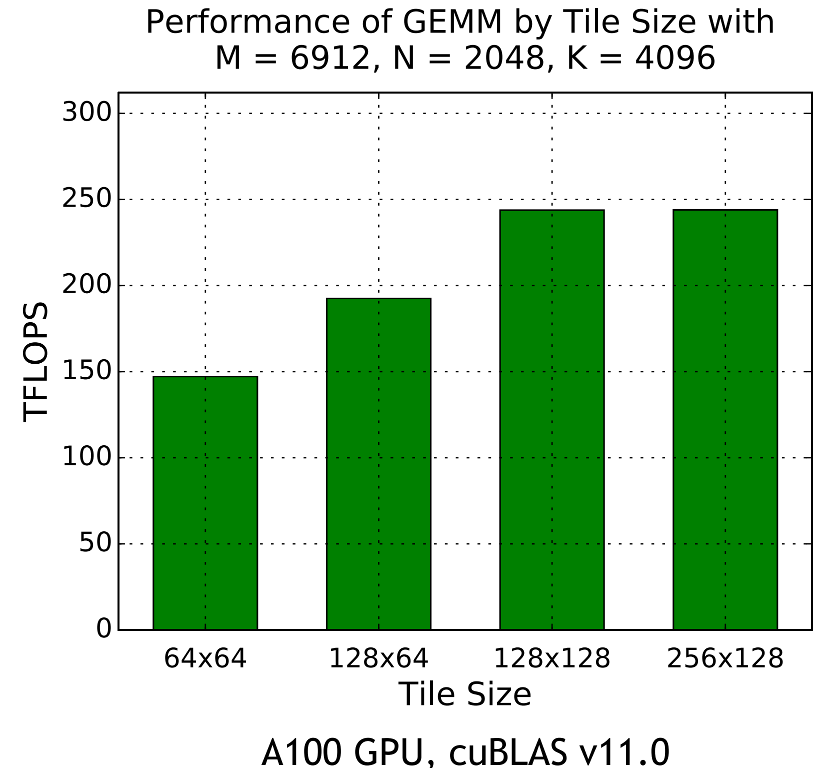
EFFICIENT PARALLELIZATION

How do GPUs split up operations?

Tile sizes are typically (but not always) powers of 2

For best efficiency:

- Larger tiles are more efficient
- Dividing the output matrix into tiles evenly means no wasted work



A100 GPU, cuBLAS v11.0

NVIDIA.

EFFICIENT PARALLELIZATION

What do we mean by quantization?

As a consequence of how work is parallelized, some output matrix sizes are more efficient than others

Output matrices are divided into tiles

- Tiles have fixed sizes, so it’s possible that a matrix will not divide evenly into any tiles of any size

Created tiles are divided among SMs on the GPU for calculation

- The number of SMs is fixed, so it’s possible that the tiles will not divide evenly among SMs

When we say “quantization” in this presentation, we’re referring to this effect (rather than any effect relating to precisions)

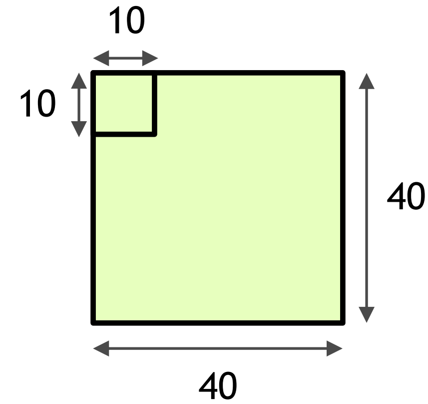
Choosing matrix dimensions with attention to tile size and SM count is a great way to improve performance!

EFFICIENT PARALLELIZATION

The perfect case

Let's talk about a hypothetical GPU with 10x10 tiles and 16 SMs

For a 40x40 matrix:



NVIDIA

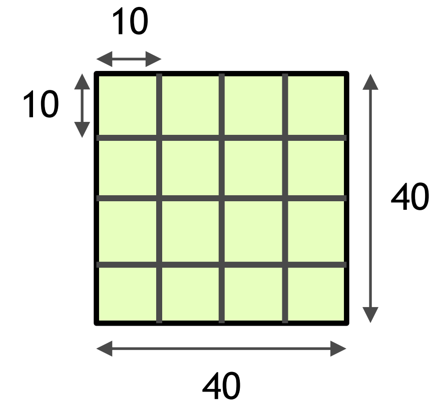
EFFICIENT PARALLELIZATION

The perfect case

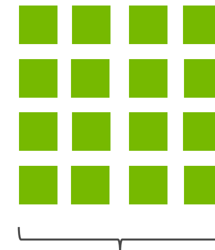
Let's talk about a hypothetical GPU with 10x10 tiles and 16 SMs

For a 40x40 matrix:

- Matrix dimensions are **divisible** by tile size
  - 40 / 10 = 4 tiles exactly on each side
- Number of tiles created is **divisible** by SM count
  - 16 tiles / 16 SMs = 1 tile per SM exactly
- This is a best-case scenario



evenly divisible



evenly divisible

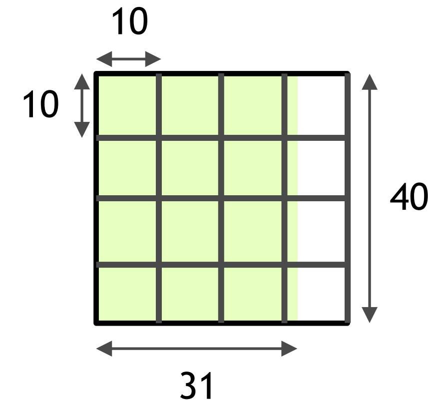
EFFICIENT PARALLELIZATION

Tile quantization

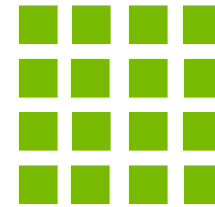
Still on a hypothetical GPU with 10x10 tiles and 16 SMs

Consider instead a 40x31 matrix:

- Matrix dimensions are **not divisible** by tile size
  - 31 / 10 = 3 full tiles plus one mostly-empty tile per column
- Number of tiles created is **divisible** by SM count
  - 16 tiles / 16 SMs = 1 tile per SM exactly
- Work wasted on empty portions of tiles



not evenly divisible



1 full wave, still

evenly divisible

NVIDIA.

EFFICIENT PARALLELIZATION

Wave quantization

Still on a hypothetical GPU with 10x10 tiles and 16 SMs

Consider instead a 40x50 matrix:

Matrix dimensions are **divisible** by tile size

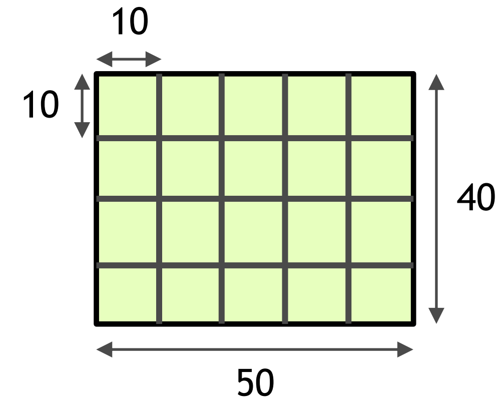
50 / 10 = 5 tiles exactly per column

Number of tiles created is **not divisible** by SM count

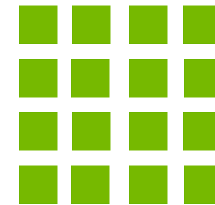
20 tiles / 16 SMs = 16 tiles divided evenly between SMs

+ 4 "tail" tiles processed while 12 SMs are idle

Work wasted while SMs are idle



evenly divisible



not evenly divisible

NVIDIA

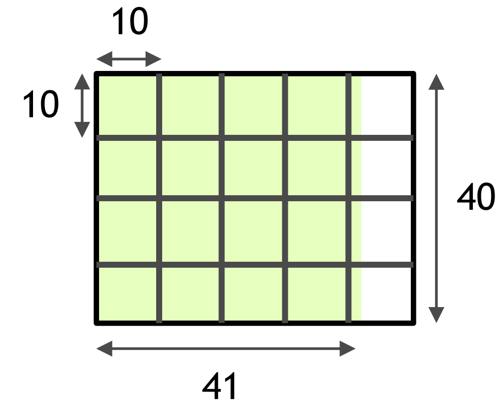
EFFICIENT PARALLELIZATION

Both tile and wave quantization

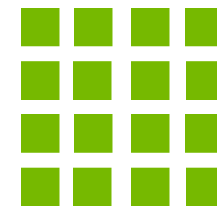
Still on a hypothetical GPU with 10x10 tiles and 16 SMs

Consider instead a 40x41 matrix:

- Matrix dimensions are **not divisible** by tile size
  - 41 / 10 = 4 full tiles + 1 nearly-empty tile per column
- Number of tiles created is **not divisible** by SM count
  - 20 tiles / 16 SMs = 16 tiles divided evenly between SMs
    - + 4 “tail” tiles processed while 12 SMs are idle
- Work wasted on both tile and wave inefficiency



not evenly divisible



1 full wave

1 tail wave (75% idle)

not evenly divisible

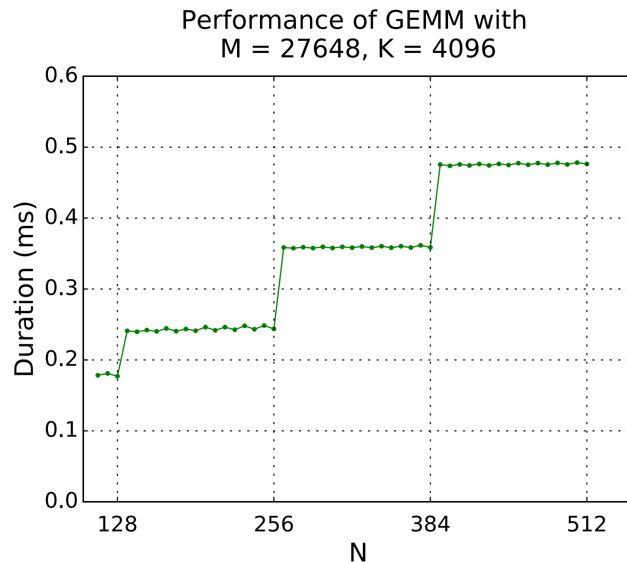
NVIDIA

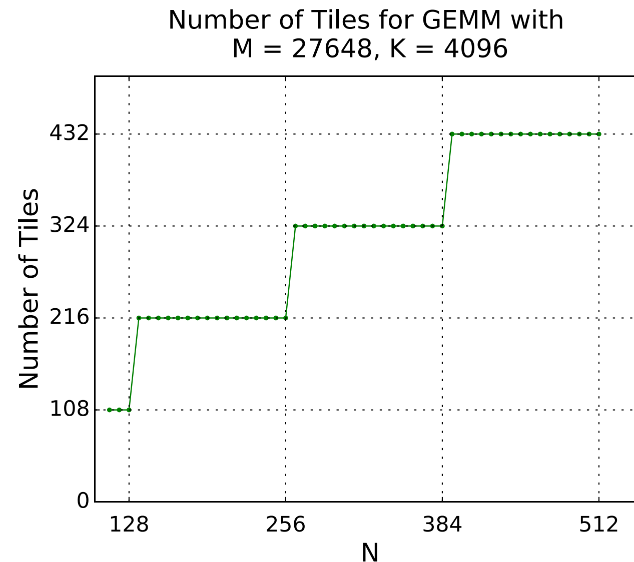
EFFICIENT PARALLELIZATION

Tile quantization example

A concrete example: pick dimensions to be multiples of tile dimensions for best efficiency

(To demonstrate the effect clearly, 256x128 tiles were used for all N; in practice, cuBLAS would select narrower tiles to mitigate performance drop)





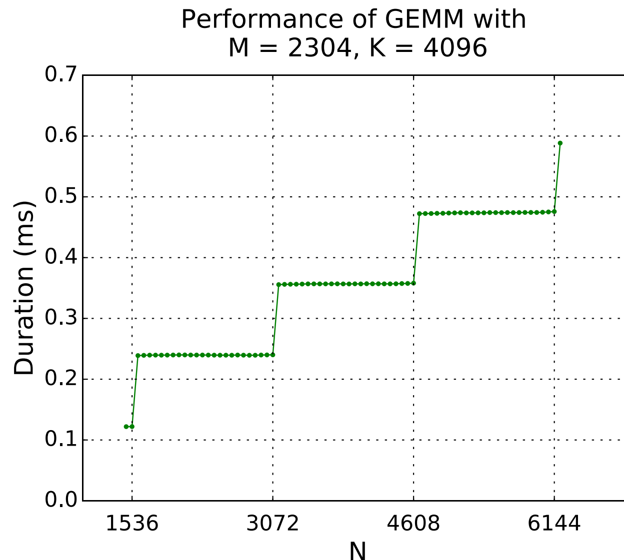
A100 GPU, cuBLAS v11.0

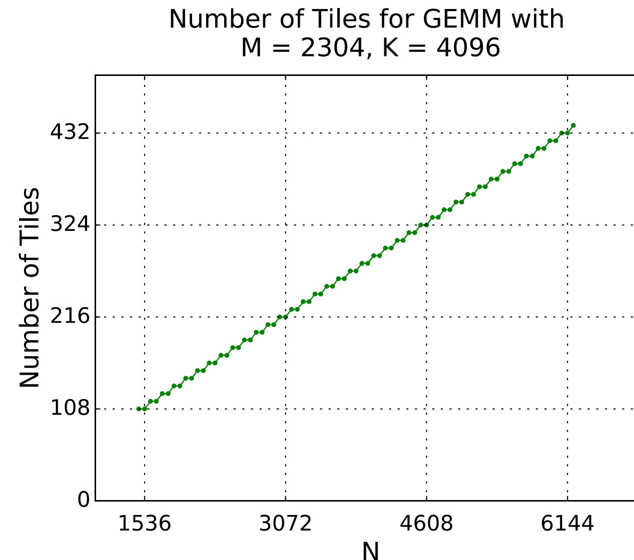
EFFICIENT PARALLELIZATION

Wave quantization example

A concrete example: for best efficiency, make sure the number of tiles is a multiple of the GPU's SM count

(Like the previous example, 256x128 tiles were used for all N)





A100 GPU, cuBLAS v11.0

NVIDIA

EFFICIENT PARALLELIZATION

At work with Transformer

Transformer networks include a block of feed-forward fully-connected layers

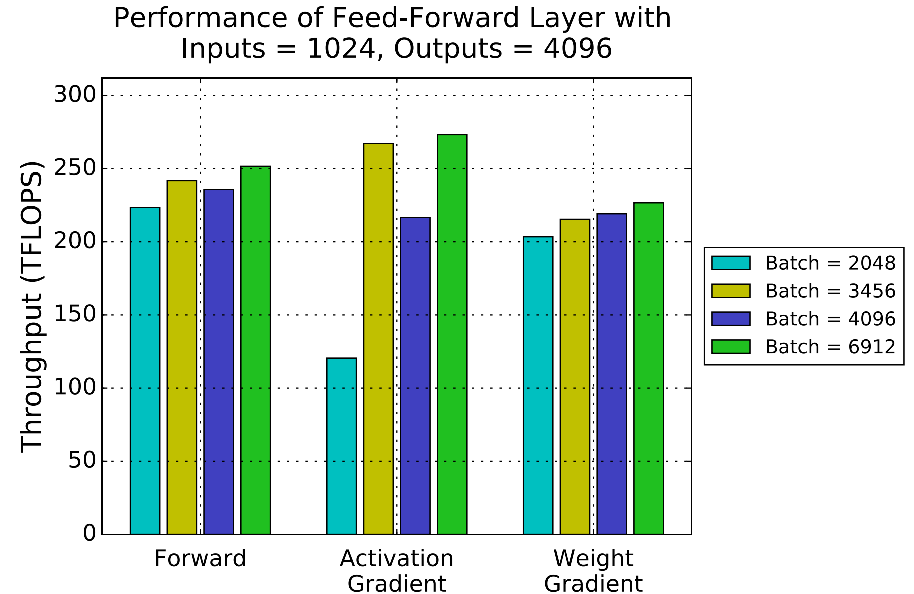
Batches with 3456 or 6912 tokens divide perfectly, assuming 128x128 tiles

$$Tiles = \left\lceil \frac{4096}{128} \right\rceil \times \left\lceil \frac{tokens\ per\ batch}{128} \right\rceil$$

3456 tokens = 864 tiles = 8 full waves

4096 tokens = 1024 tiles = 9 full waves + a 52-tile tail

In fact, training is **more efficient (~7%) with 3456 tokens than it is with larger batches of 4096**



A100 GPU, cuBLAS v11.0

EXPLORE ALTERNATIVE IMPLEMENTATIONS

Choose the right version of an operation for your situation

What about memory-limited layers and other operations where we can’t tweak parameters for better performance?

Persistent implementations hold data in on-chip memory instead of loading it repeatedly

- This speeds up operations that are memory-limited
- (But also requires that data can fit in on-chip memory)

Recurrent layers can use persistent weights

- Non-persistent recurrent layers read each weight from off-chip memory multiple times; if a weight is needed repeatedly, it will be read repeatedly
- Persistent recurrent layers access weights once, then retain them in on-chip memory for further use (although weights must be small enough to fit!)

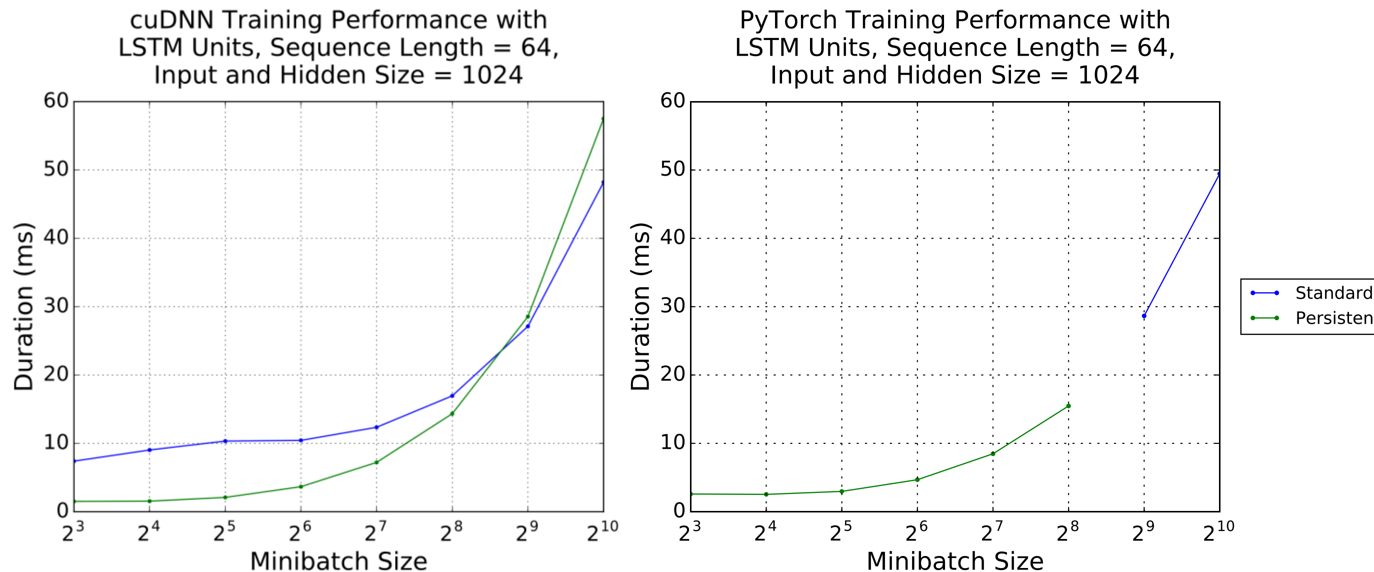
NVIDIA.

EXPLORE ALTERNATIVE IMPLEMENTATIONS

Persistence with recurrent layers

Persistence is advantageous for small problem sizes, where memory is a limiting factor

PyTorch enables persistence automatically when it can benefit performance



Tesla V100-SXM3-32GB GPU, cuDNN v7.6, PyTorch v1.5

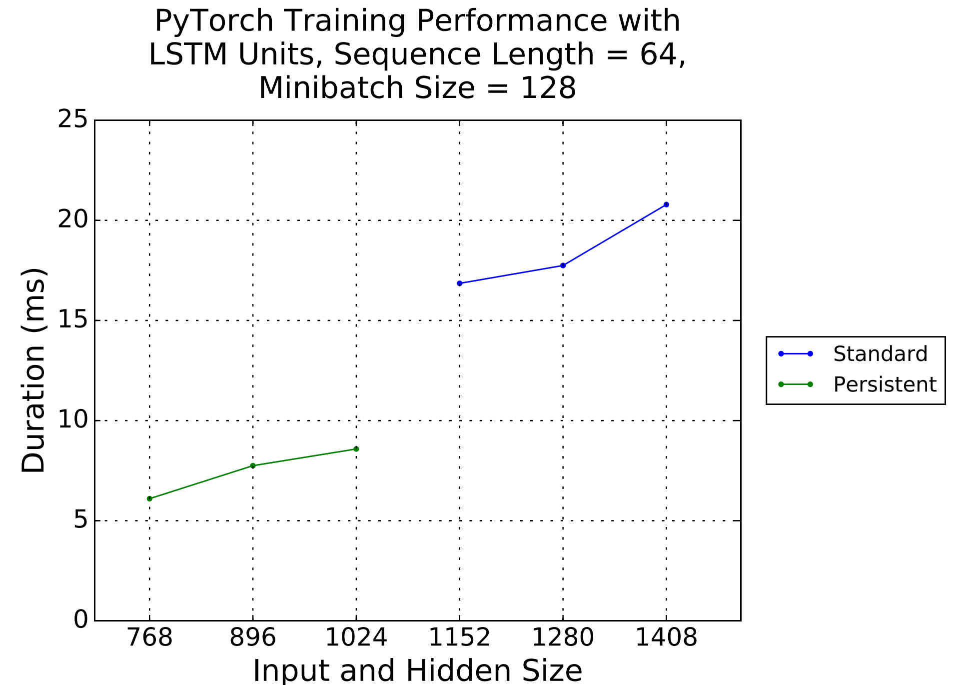
EXPLORE ALTERNATIVE IMPLEMENTATIONS

Persistence with recurrent layers

Persistence can improve performance for recurrent layers; weights are cached on-chip

Here, hidden sizes over 1024 for LSTM layers mean that the weights are too large to be cached, so the non-persistent implementation is used instead

In other words, if minibatch size is small and non-negotiable, choosing hidden size of 1024 or less will allow you to take advantage of persistence



Tesla V100-SXM3-32GB GPU, cuDNN v7.6, PyTorch v1.5

NVIDIA

TENSOR CORES CHEAT SHEET

Make sure Tensor Cores can run efficiently by aligning key dimensions to 128 bits

- For fully-connected layers: input, output, and batch size
- For convolutional layers: input and output channel counts
- For recurrent layers: minibatch size and hidden sizes

Provide enough work to fill the GPU

- Choose key dimensions larger than 256, and at least one **substantially** larger, to be math-limited

Aim for good tile and wave efficiency, especially when at least one dimension is small

- Choose key dimensions to be multiples of 64/128/256
- Ensure the number of tiles is a multiple of the SM count

If you can't follow every guideline, following as many as possible still helps performance!

https://docs.nvidia.com/deeplearning/performance/dl-performance-getting-started/index.html#checklists

NVIDIA.

NVIDIA®