Efficient Deep Learning Systems
Course introduction

Max Ryabinin

2026

Why are we here?

- DL as a field is getting mature:
  - Neural networks are becoming more and more widespread in practice
  - Scaling trends everywhere (model size, dataset size, coauthor list size)
  - .ipynb-based development is no longer viable even for basic stuff :)
- Training and running large models is necessary for frontier-level tasks
- Engineering knowledge is now **essential** for SOTA research
- For practical applications, performance and maintainability are key factors

Bird's eye view of DL

Training → Inference

How to achieve the best quality?

~~Is my model useful?~~

~~Is my model good enough?~~

How to run the experiments quickly enough?

Is performance good enough for my use case?

Do I utilize my resources to the fullest?

How do I ensure the model can serve the demand?

How many resources do I need in the first place?

…How do I not run out of money in the process? :)

Goal of the course

- Most DL courses do not cover practical details and overall systems:
  - Small code changes can make your training/inference much faster
  - Deployment of trained networks, both on their own and as a part of a larger system
  - Streamlined maintenance by treating ML models like any other code (testing, versioning, etc.)
- Knowledge about this is scattered around the Internet and unstructured
- We want to give you these useful bits of practical knowledge!
- …no bleeding-edge methods or last-week papers (with some exceptions)

Plan

1. *(You are here)* Intro, basics of GPU architecture & benchmarking
2. Profiling DL pipelines, techniques for efficient training
3. Data-parallel training, All-Reduce, torch.distributed intro
4. Memory-efficient training, model parallelism
5. Sharded data parallel training, optimizations for distributed training
6. Large-scale training arithmetics
7. Basics of web service deployment
8. Inference optimizations: systems
9. Inference optimizations: algorithms

Training pipelines and performance

Large-scale training

Deployment in production

Logistics

- Lectures&seminars: every Wednesday, 18:00 – 21:00, via Zoom
- Course repo: github.com/mryab/efficient-dl-systems
- YSDA LMS for submitting assignments
- Channel with announcements: see HSE FCS wiki/course page in LMS
- Resources: Yandex Cloud VM + DataSphere (HSE), YSDA GPUs + DataSphere (YSDA)

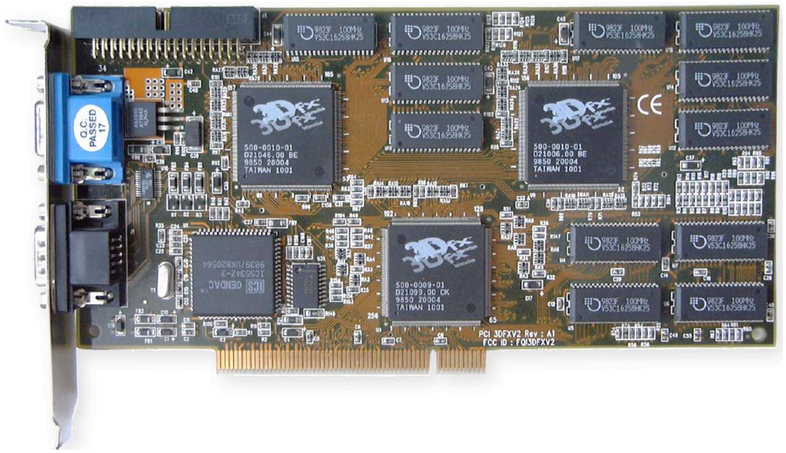
Grading

- 3 assignments:
    1. Fast pipelines (1 part)
    2. Large-scale training (4 parts)
    3. Deployment (3 parts)

- Each assignment consists of sub-assignments given each week (except this one)
- Final grade: $G_{total} = 0.1G_1 + 0.5G_2 + 0.4G_3$

GPU architecture: a brief overview

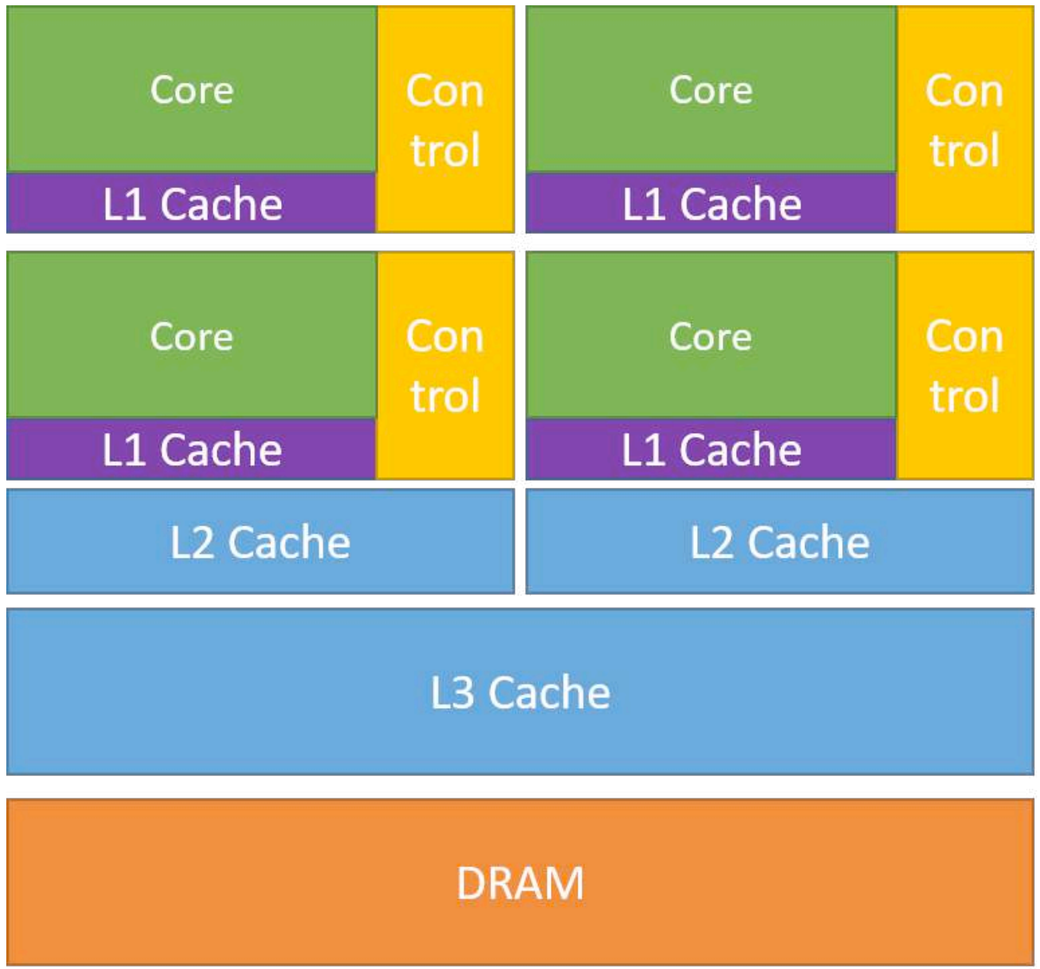
- As the name suggests, originally used for graphics
- Highly parallel execution model: objects can be rendered simultaneously
- Since ~2007, simple GPGPU API started to appear (CUDA, OpenCL, Metal)
- GPU-trained AlexNet/DanNet sparked the DL revolution in early 2010s



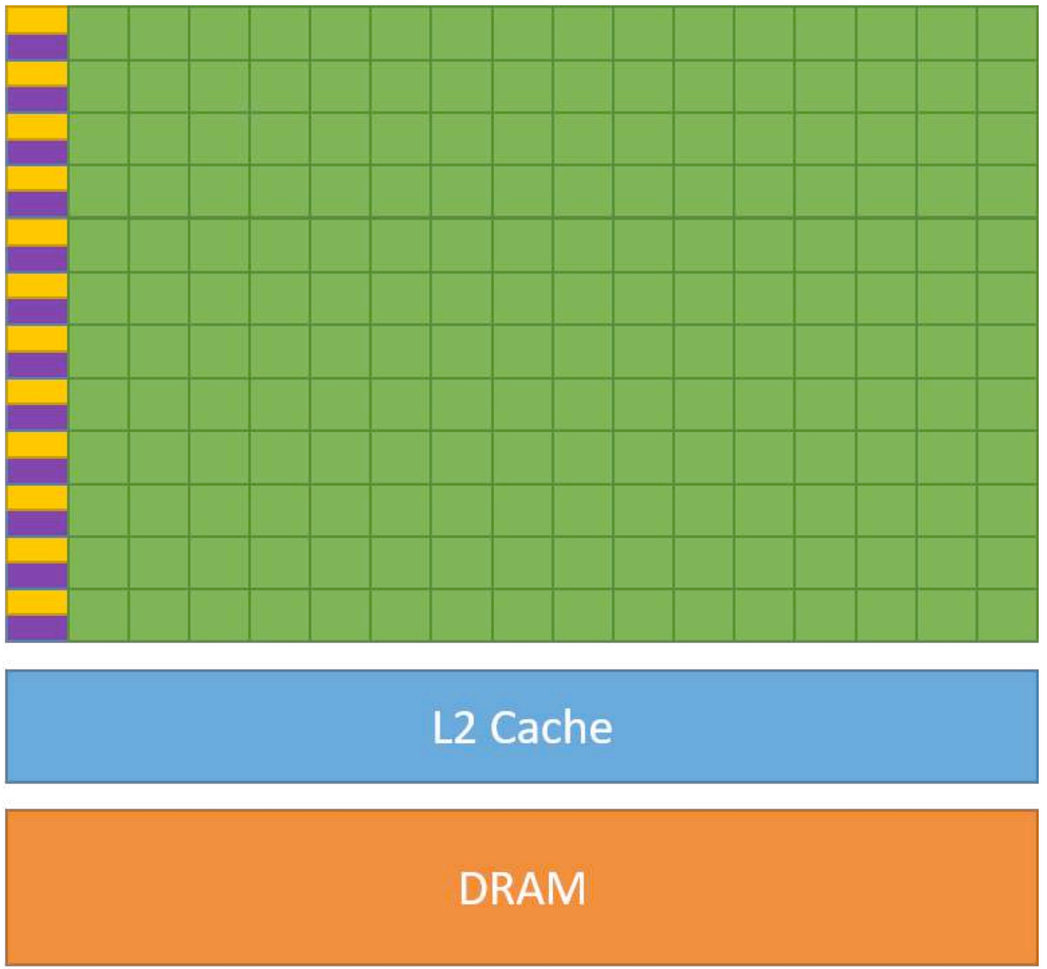
3dfx Voodoo2: 12MB RAM

NVIDIA GB200 NVL72: 186GB RAM

GPU architecture: a brief overview



CPU

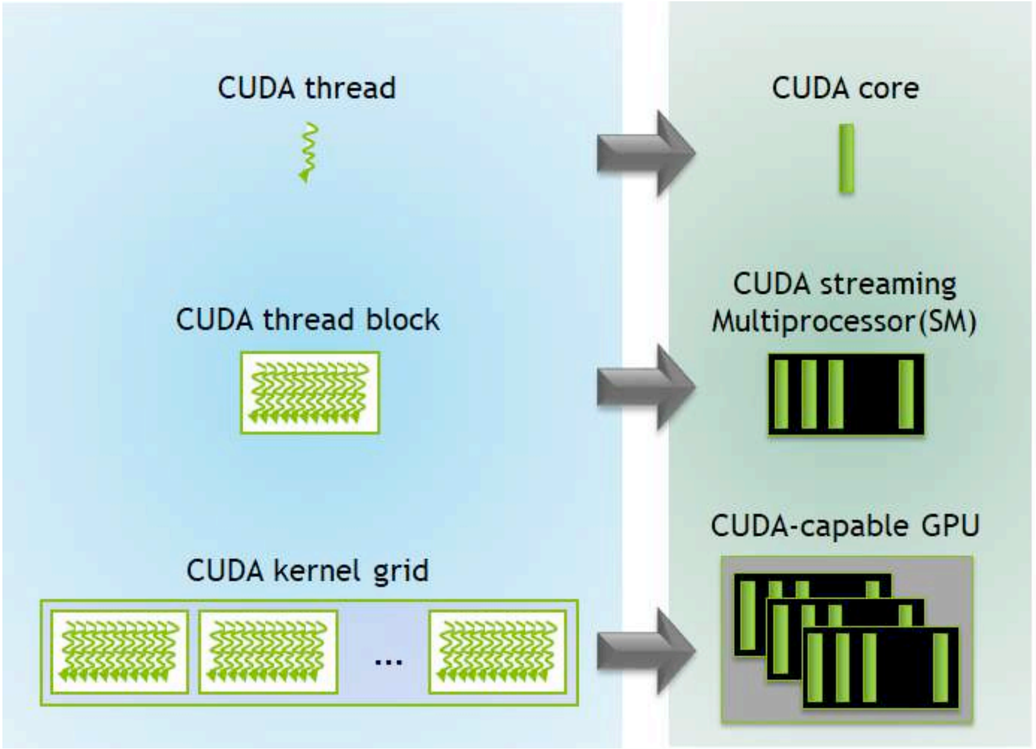


GPU

docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA computation model

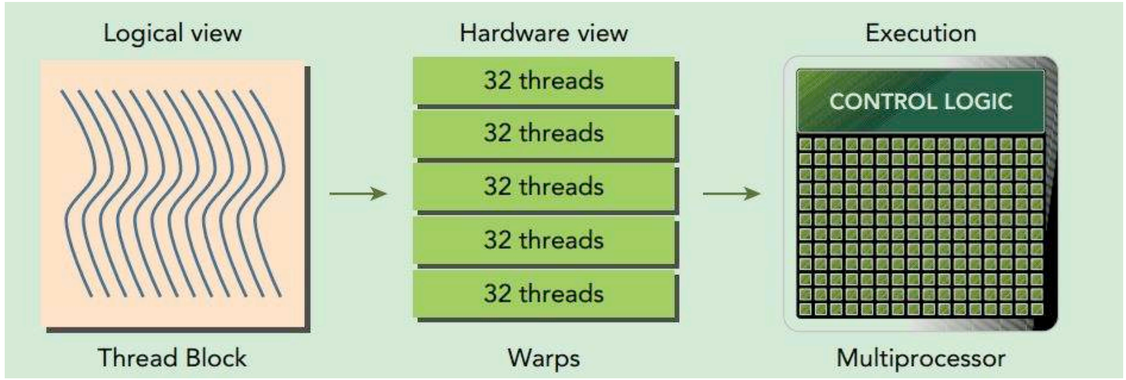
- In CUDA, we launch kernels from the host that are executed in parallel on the device
- Kernels are executed by threads grouped in thread blocks of limited size
- A GPU is composed of multithreaded Streaming Multiprocessors (SMs) that are assigned different thread blocks
- Multiple thread blocks are arranged in grids (can be 1D, 2D, or 3D)

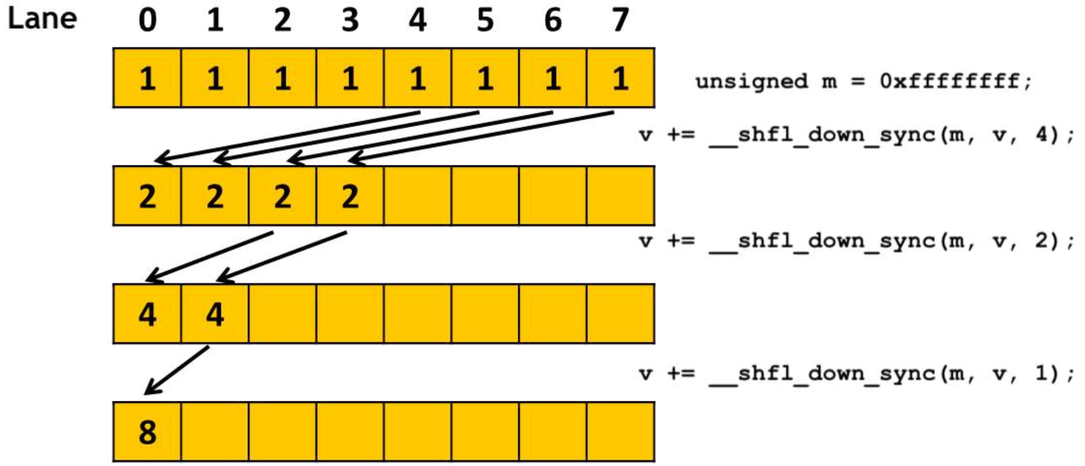


developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

GPU computations: hardware side

- SIMT (Single Instruction, Multiple Thread)
- On a physical level, threads are executed in groups of 32 called warps
- A warp executes one instruction at a time:
  in case of branching, all paths need to be taken
- This does not affect correctness but has major performance implications
- Warp-level primitives can be leveraged for parallel computation

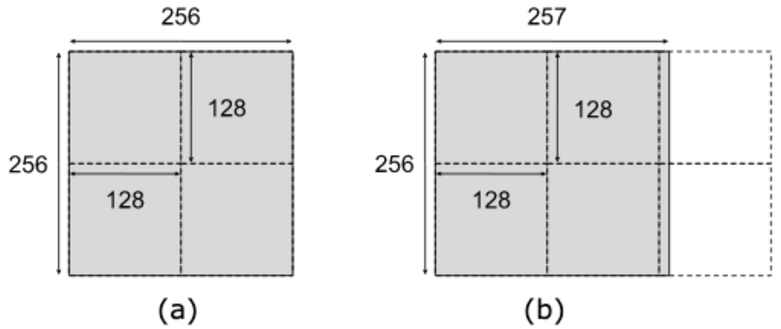


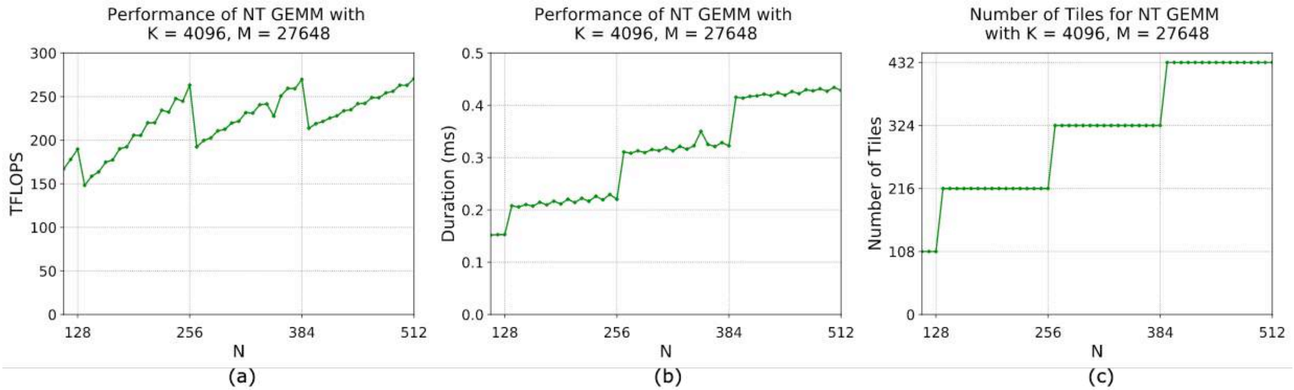


developer.nvidia.com/blog/using-cuda-warp-level-primitives

Why does all of this matter?

- The most popular operation in DL is matrix multiplication
- Executing this in parallel can have two potential effects when dividing the work
- **Tile Quantization**: matrix size is not divisible by the thread block tile size

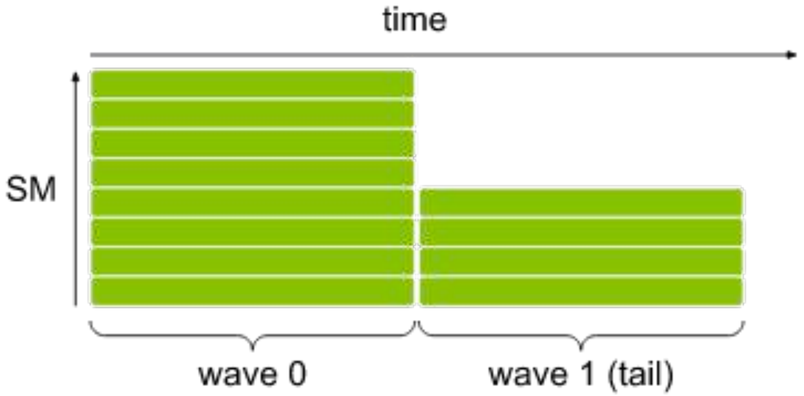


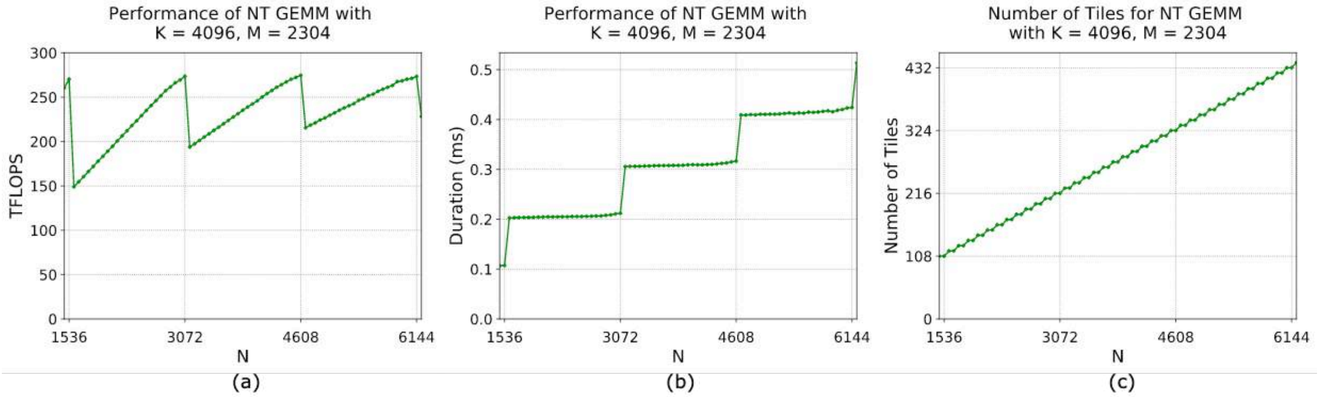


docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Why does all of this matter?

- The most popular operation in DL is matrix multiplication
- Executing this in parallel can have two potential effects when dividing the work
- **Tile Quantization**: matrix size is not divisible by the thread block tile size
- **Wave Quantization**: total number of tiles is quantized to the number of SMs
- Both effects can be quite noticeable for small or irregular shapes!

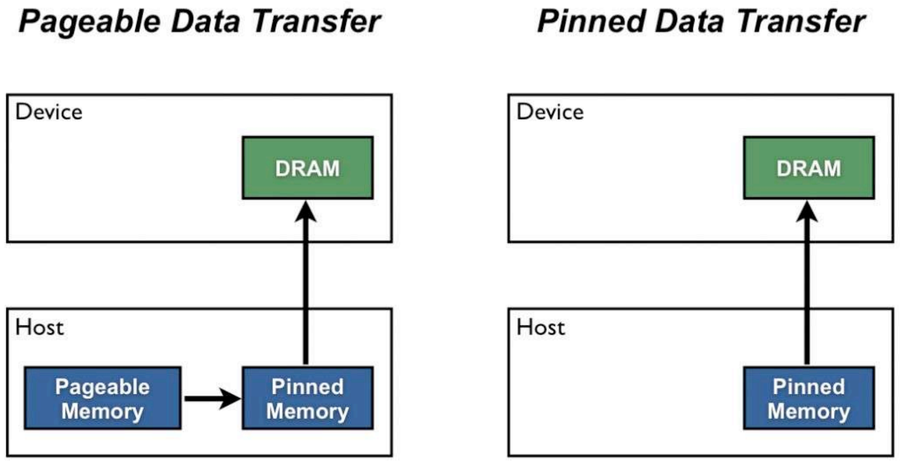


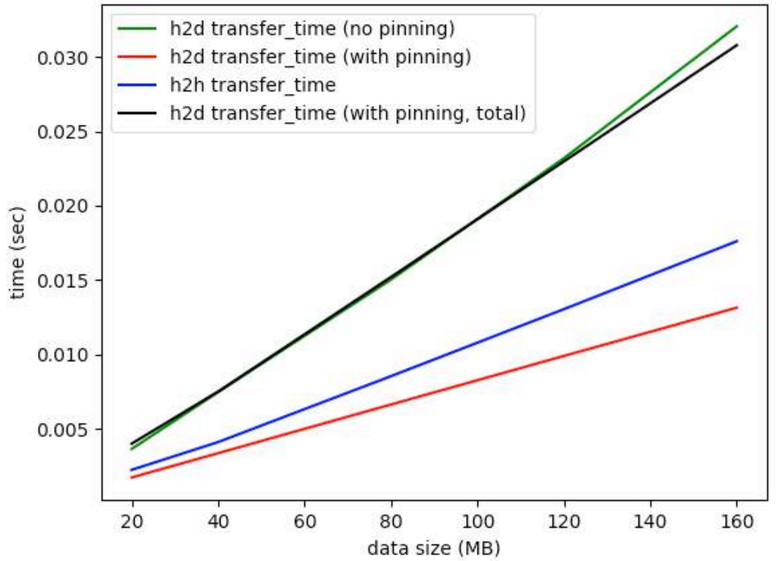


docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Memory access

- GPU has a separate memory unit (called device memory)
- Need to copy from host memory and back (PCIe 4.0 x16 — 32GB/s peak)
- Memory transfer is often a bottleneck
- Pinned (page-locked) memory access is much faster

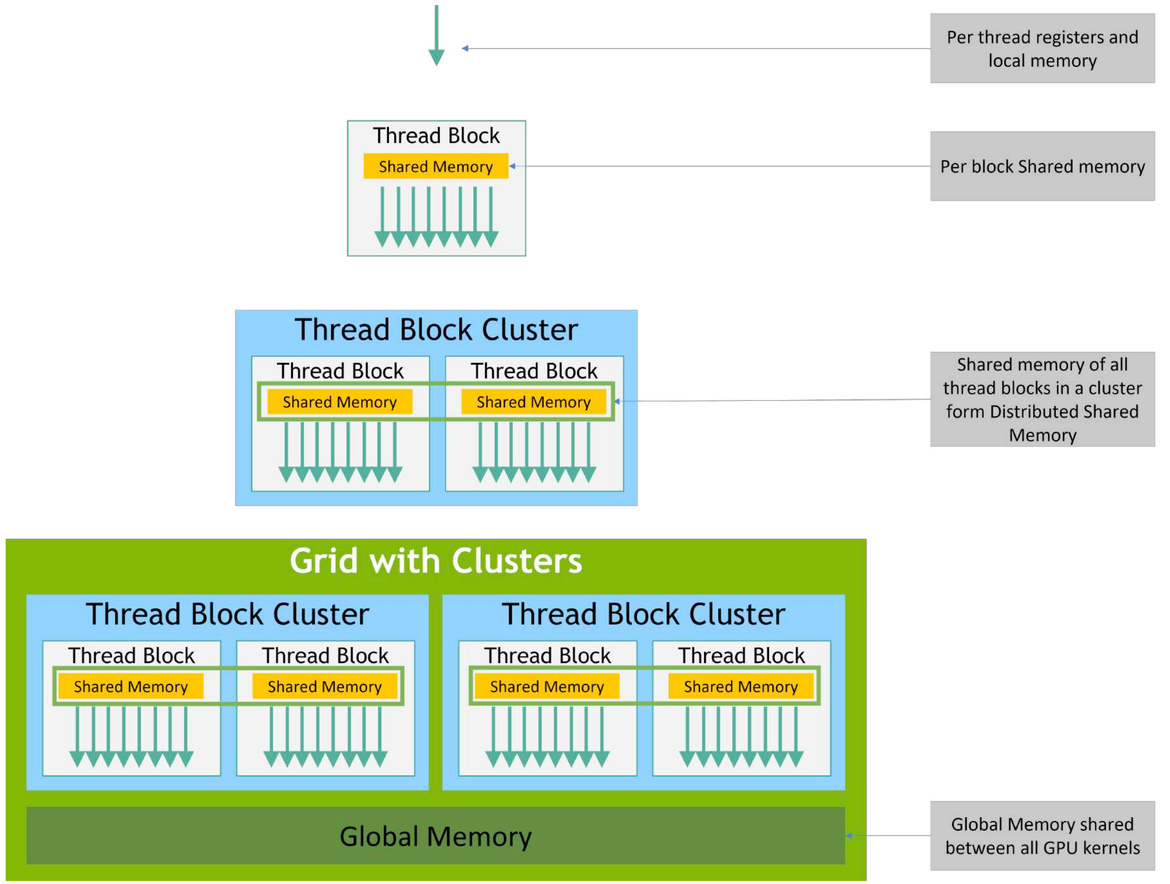




https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

Memory access

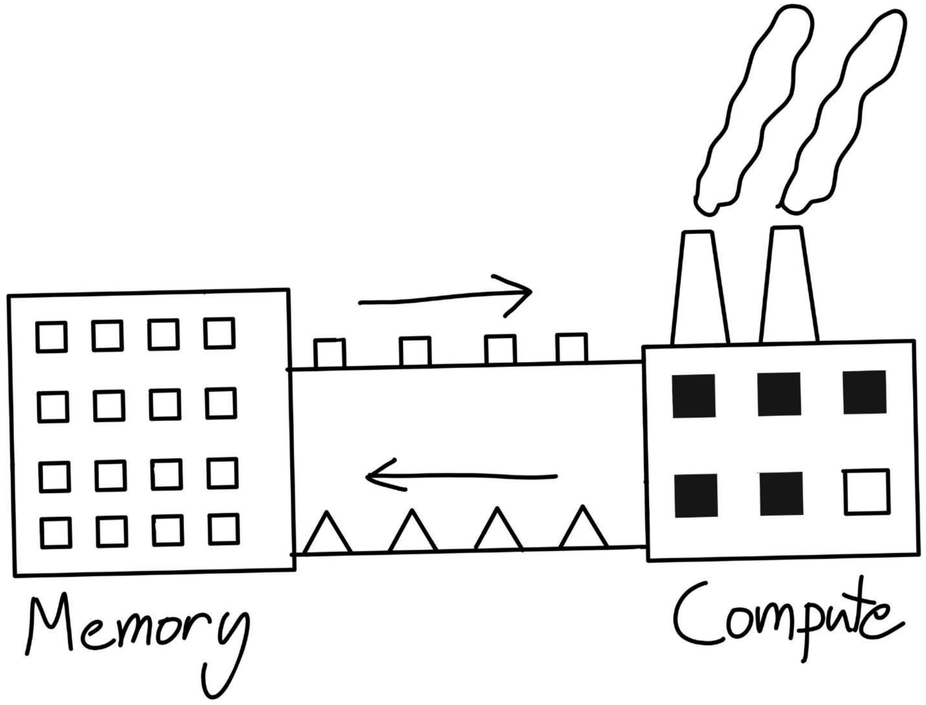
- GPU has a separate memory unit (called device memory)
- Need to copy from host memory and back (PCIe 4.0 x16 — 32GB/s peak)
- Memory transfer is often a bottleneck
- Pinned (page-locked) memory access is much faster
- Memory hierarchy is a thing, just like on CPUs!



https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

GPU Performance 101

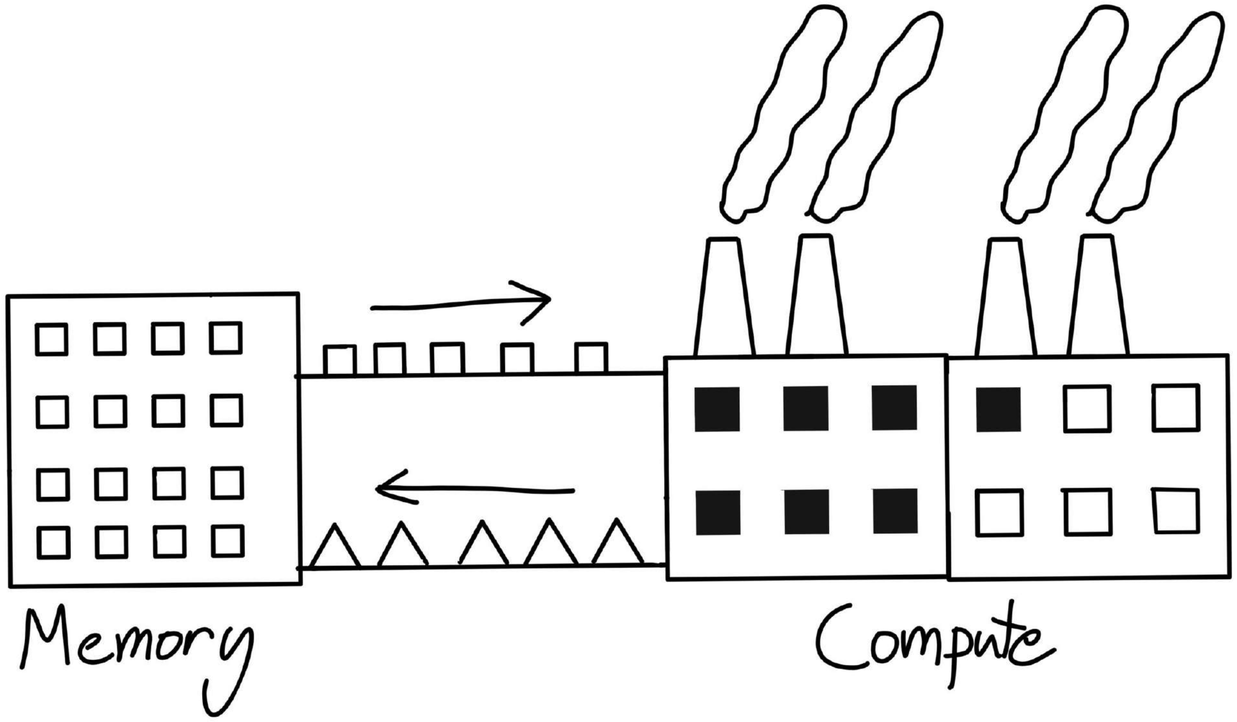
- GPUs are **exceptionally** good at number crunching
- However, the memory bandwidth of GPUs themselves hardly keeps up



horace.io/brrr_intro

GPU Performance 101

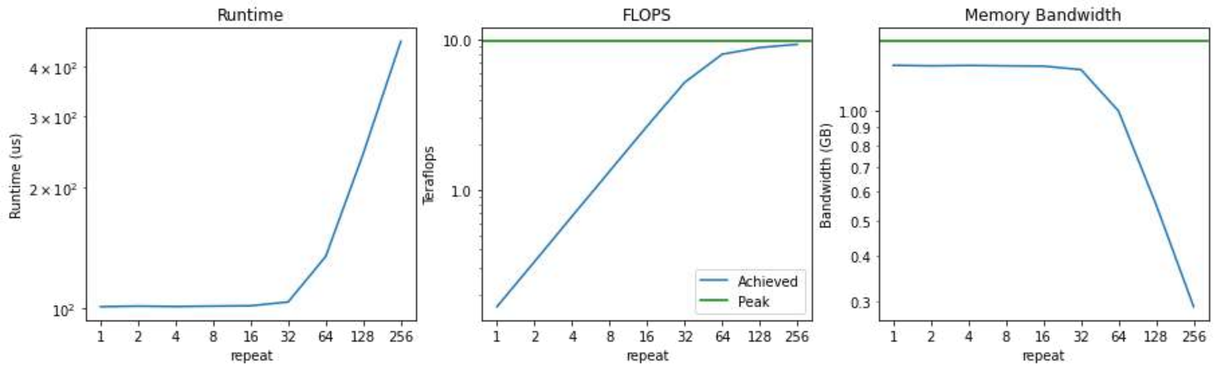
- GPUs are **exceptionally** good at number crunching
- However, the memory bandwidth of GPUs themselves hardly keeps up



horace.io/brrr_intro

GPU Performance 101

- GPUs are **exceptionally** good at number crunching
- However, the memory bandwidth of GPUs themselves hardly keeps up



horace.io/brrr_intro

GPU Performance 101

- GPUs are **exceptionally** good at number crunching
- However, the memory bandwidth of GPUs themselves hardly keeps up
- The key optimizations are often about **minimizing or hiding data movement!**
- The principle applies both at the GPU level and the multi-GPU level
- Recent accelerators introduce asynchronous/specialized memory copies to alleviate that problem

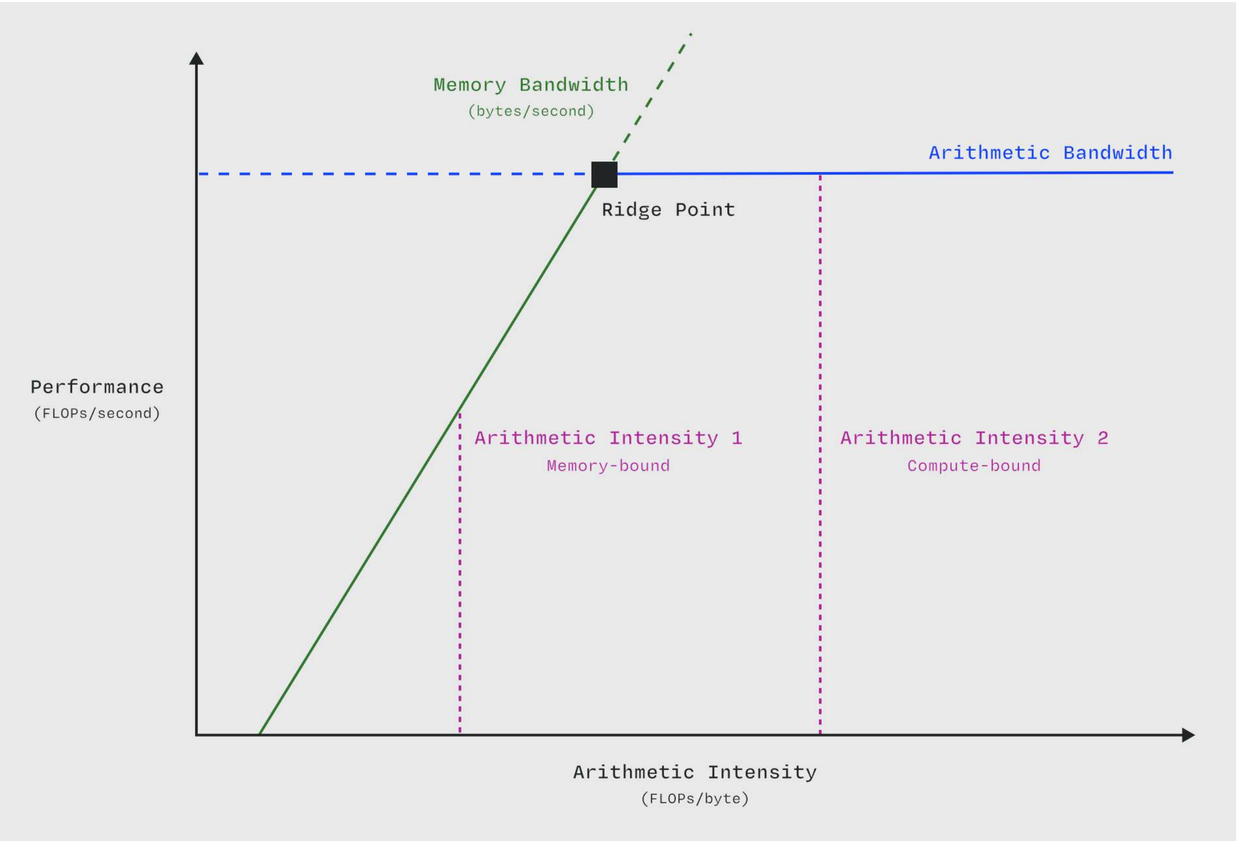
Table 1. Proportions for operator classes in PyTorch.

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

horace.io/brrr_intro

The Roofline Model™

- How to find where the bottleneck is?
- Usually, the tradeoff is **between the data transfer and compute operations**
- Arithmetic intensity: $\frac{\text{Computation FLOPs}}{\text{Transferred bytes}}$
- Roofline: plot peak achievable FLOPs against the arithmetic intensity

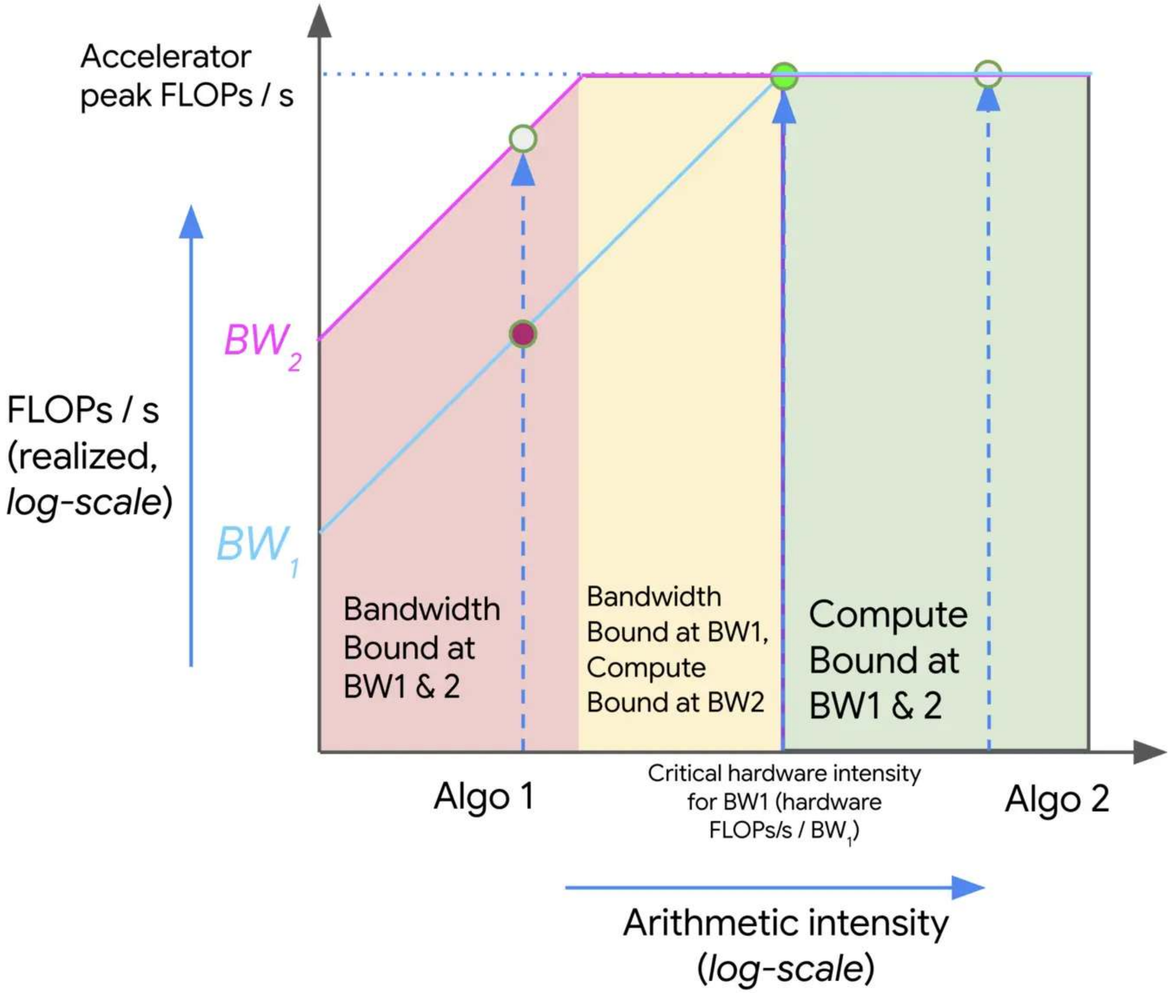


jax-ml.github.io/scaling-book/roofline

modal.com/gpu-glossary/perf/roofline-model

The Roofline Model™

- How to find where the bottleneck is?
- Usually, the tradeoff is **between the data transfer and compute operations**
- Arithmetic intensity: $\frac{\text{Computation FLOPs}}{\text{Transferred bytes}}$
- Roofline: plot peak achievable FLOPs against the arithmetic intensity
- You can use this to optimize both the code and input shapes for your algorithms!

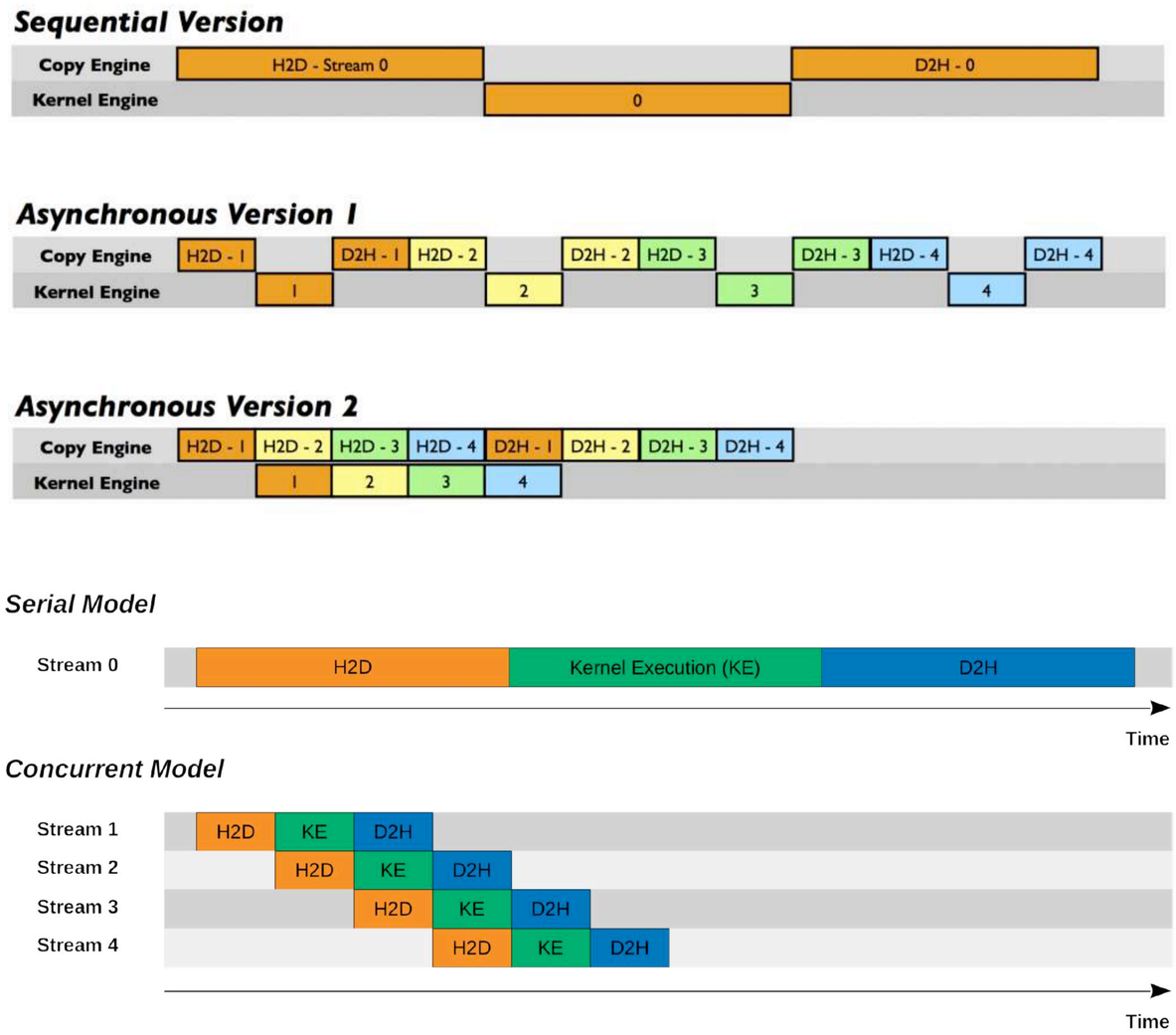


jax-ml.github.io/scaling-book/roofline

modal.com/gpu-glossary/perf/roofline-model

Asynchronous execution

- By default, CUDA kernel calls and device transfers are asynchronous
- You can send several kernels and wait for results
- Recent versions of CUDA offer better concurrency mechanisms (streams, graphs)



https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/

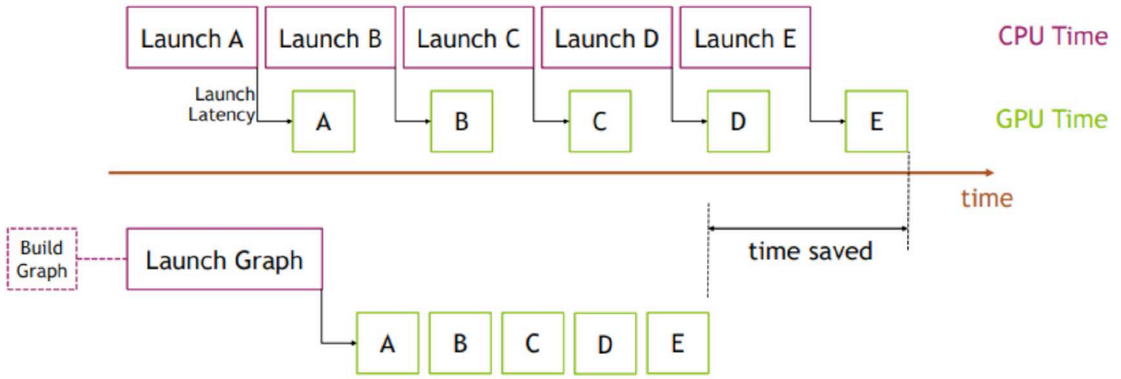
DL specifics

With PyTorch as an example:

- Kernel execution is asynchronous, which hides the latency of Python
- Be careful when benchmarking though!
- Calling Tensor.item() triggers a D2H copy
- Allocated memory is not released immediately to simplify caching
- torch.backends.cudnn.benchmark=True
- CUDA streams, graphs etc. are available in latest releases

nn.Conv2d with 64 3x3 filters applied to an input with batch size = 32, channels = width = height = 64.

Setting	`cudnn.benchmark = False` (the default)	`cudnn.benchmark = True`	Speedup
Forward propagation (FP32) [us]	1430	840	1.70
Forward + backward propagation (FP32) [us]	2870	2260	1.27



Measuring performance

- Benchmarking is a key step of understanding your bottlenecks and measuring the impact of optimizations
- Basically, just run the code several times or measure large workloads
- Can be done via %timeit or timeit.Timer (mind the synchronization)
- Due to possible side-effects (preallocation, caching), warmup and randomization are often necessary
- In PyTorch, you can use torch.utils.benchmark
- **Don't overoptimize!**
  - …at least before you find the true bottleneck