Efficient Deep Learning systems
Course introduction

Max Ryabinin

2022

HSE UNIVERSITY

What's this about?

- DL is maturing as a field:
  - 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 sustainable :)
- Each model is much more than just architecture, loss and even data
- Engineering knowledge becomes handy even 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?~~
Do I utilize my resources to the fullest?		Is its performance sufficient?
How to navigate 100s of experiments?		How do I ensure the model is maintainable?
How to avoid bugs in my pipeline?		How to avoid bugs in my pipeline? pt.2

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
  - Simplified 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. OS recap, distributed ML recap
3. Data-parallel training, All-Reduce, torch.distributed intro
4. Memory-efficient training, model parallelism
5. Profiling DL pipelines, tricks for efficient training
6. Basics of web service deployment
7. Deploying neural networks: software perspective
8. Deploying neural networks: ML perspective
9. Experiment tracking, model versioning
10. Testing and debugging, monitoring and maintenance

Systems & better training 1

Distributed training

Systems & better training 2

Deployment in production

Basically, MLOps

Logistics

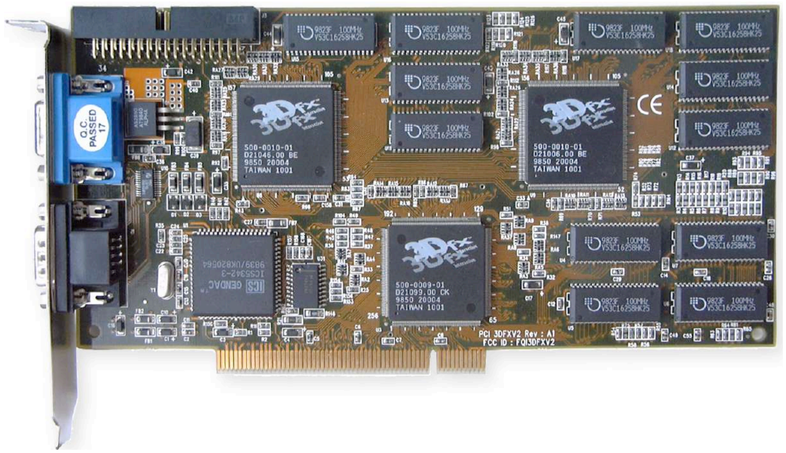
- Lectures&seminars: every Saturday, 13:00 – 16:00, most likely Zoom
- Course repo: github.com/mryab/efficient-dl-systems
- Wiki page just in case
- Channel with announcements: t.me/efficientdl_22
- Chat for discussion/questions: https://t.me/+Cqa8QoxEcPJhZWUy
- Feedback form: https://forms.gle/MkJE7bzoiAQYP9Ay6
- Yandex Cloud for resources (both Compute Cloud and Datasphere) + possibly HSE cluster

Grading

- Anytask: https://anytask.org/course/892 (invites sent out next week)
- 4 assignments:
    1. Distributed training (3 parts)
    2. Profiling&pipelines
    3. Deployment (3 parts)
    4. Monitoring&testing
- 0.3*HW1+0.1*HW2+0.4*HW3+0.2*HW4
- Arithmetic rounding, extra points possible
- All deadlines are hard

GPU architecture: a brief overview

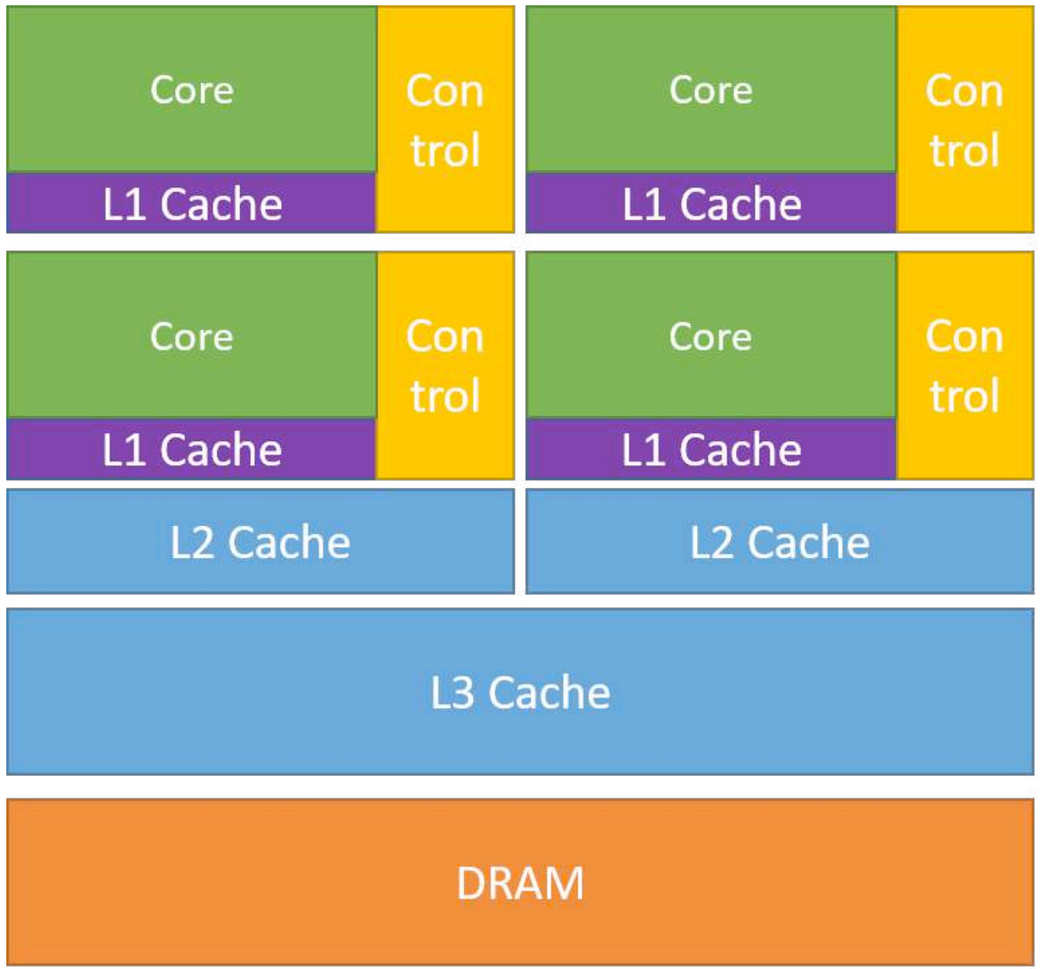
- As the name suggests, originally used for graphics
- Highly parallel execution model: objects can be rendered simultaneously
- Since ~2007, simple GPGPU API started to appear (CUDA, OpenCL, Metal)
- GPU-trained AlexNet/DanNet sparked the DL revolution in early 2010s



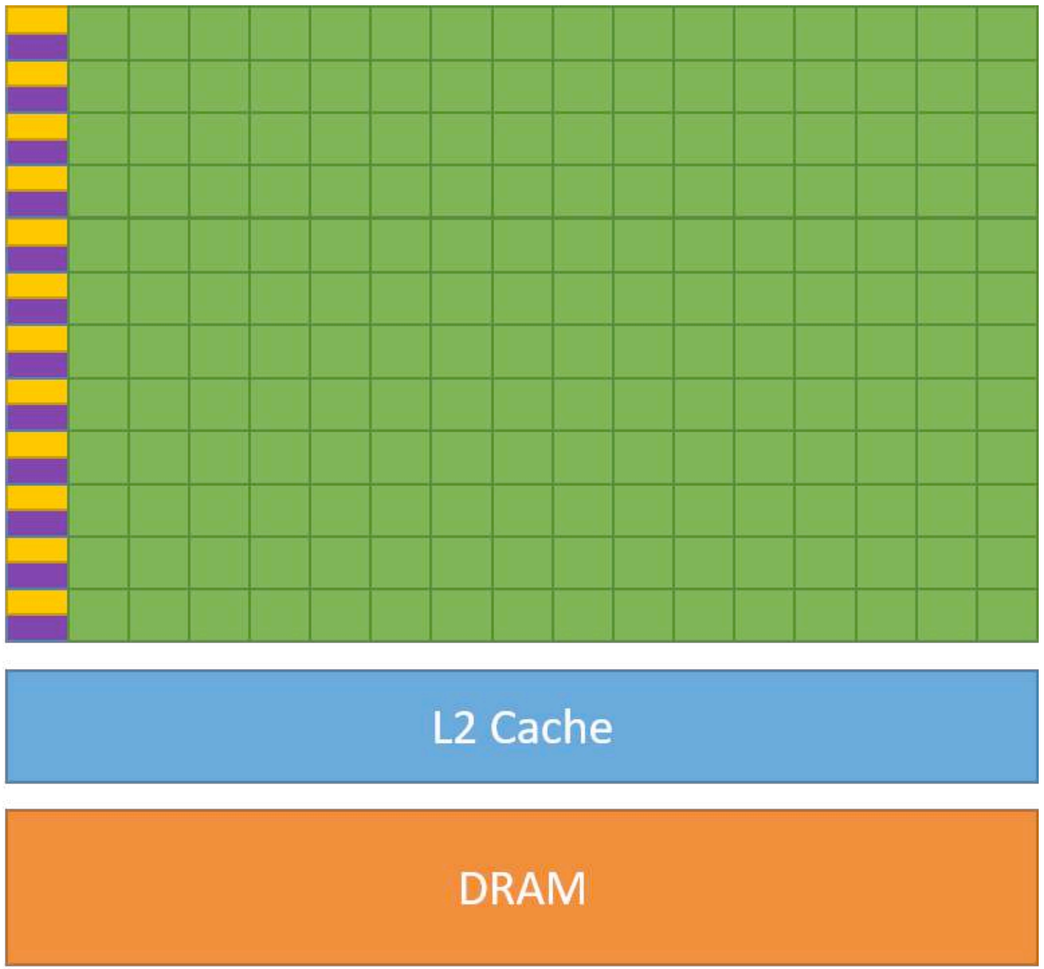
3dfx Voodoo2: 12MB RAM

NVIDIA Tesla A100: 80GB RAM

GPU architecture: a brief overview



CPU

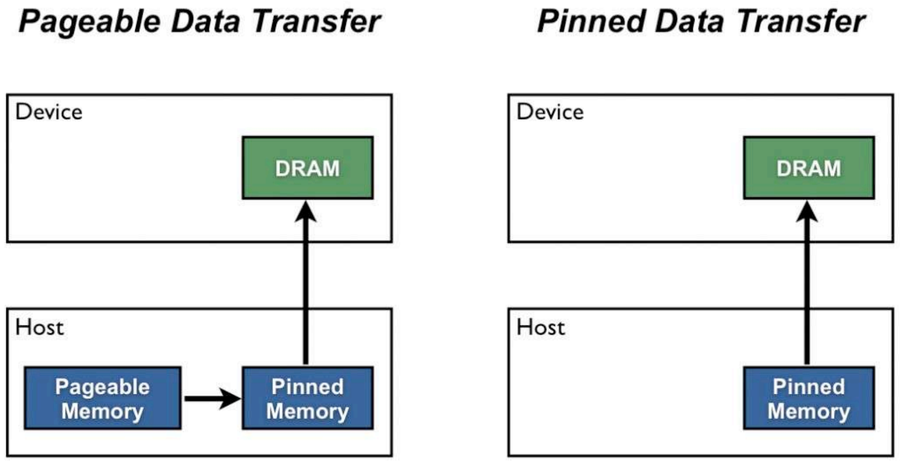


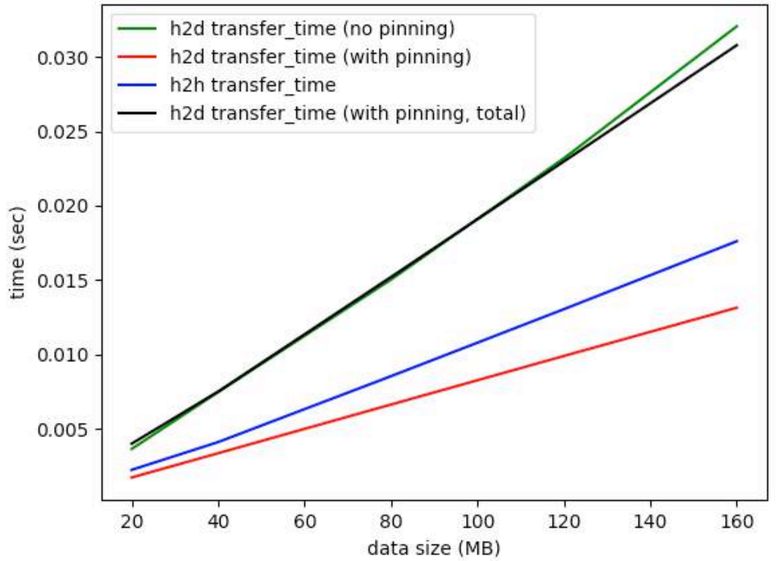
GPU

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

Memory access

- GPU has a separate memory unit (called device memory)
- Need to copy from host memory and back (PCIe 4 x8 — 15GB/s peak)
- Memory transfer can be a bottleneck
- Pinned (page-locked) memory access is much faster

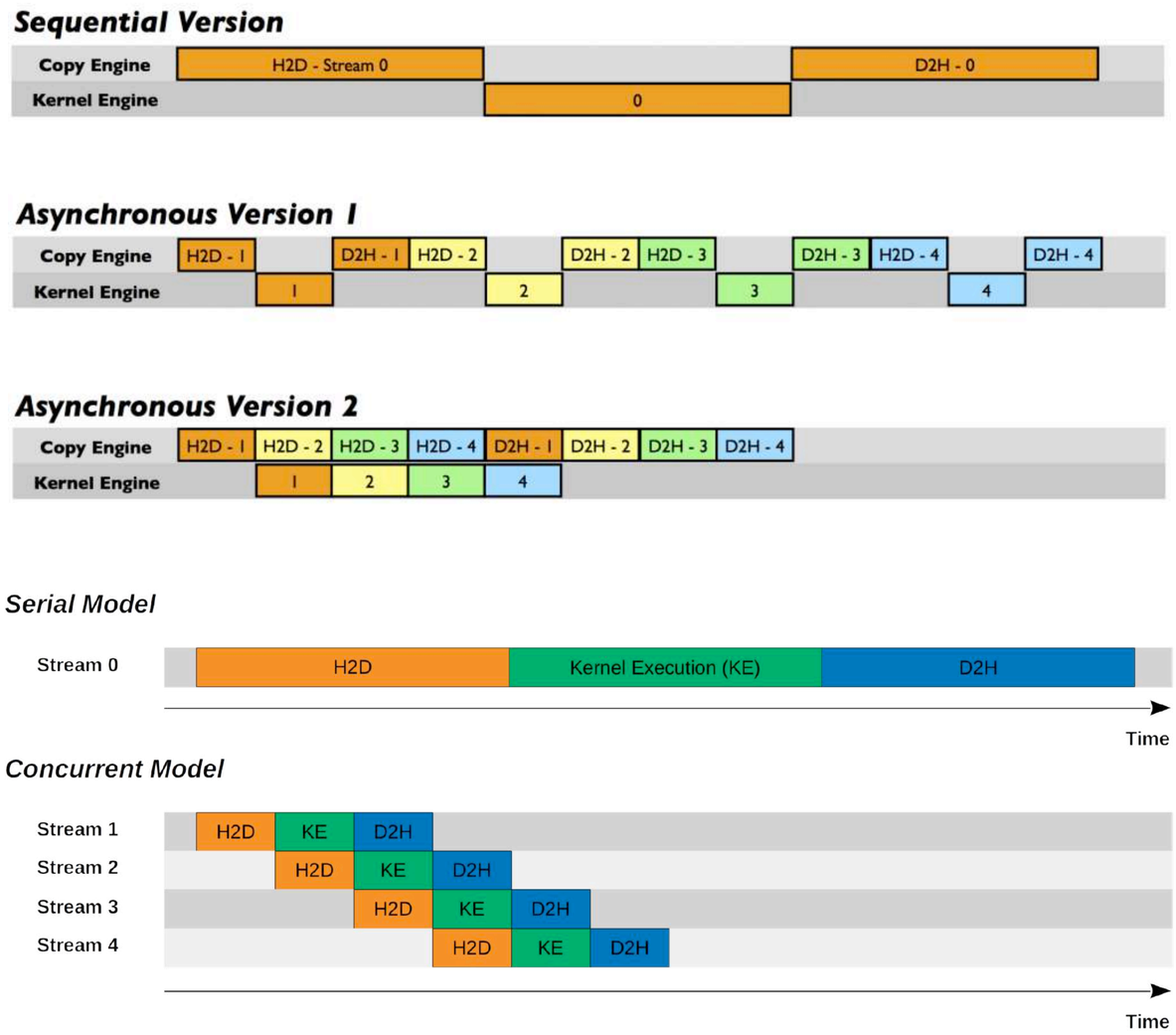




https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

Asynchronous execution

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



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

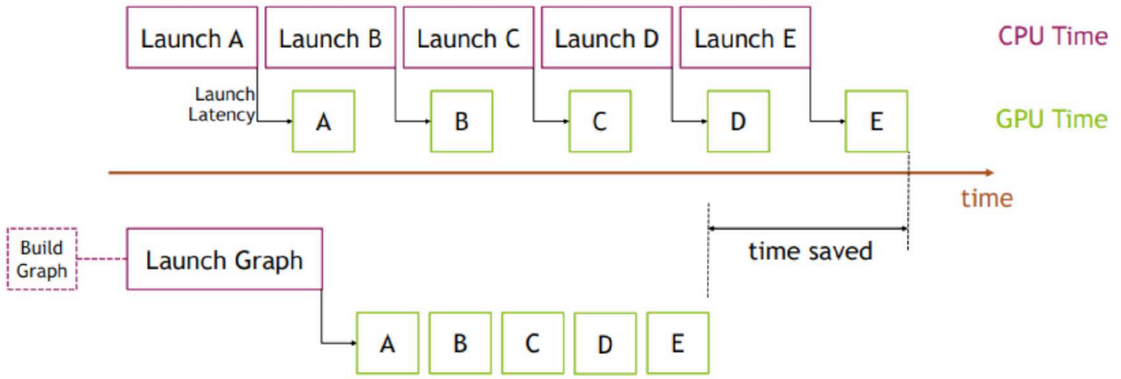
DL specifics

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!**