

# Efficient Deep Learning Systems

## Course introduction

Max Ryabinin

# What's this about?

- 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 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?

Do I utilize my  
resources to the fullest?

How to navigate 100s of experiments?

How to avoid bugs in my pipeline?

~~Is my model useful?~~

~~Is my model good enough?~~

Is performance good enough for my use case?




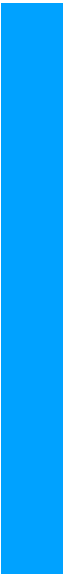

How do I ensure the model is maintainable?

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
  - 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. <i>(You are here)</i> Intro, basics of GPU architecture & benchmarking |    | <b>Systems &amp; better training 1</b> |
| 2. Experiment tracking & versioning, testing & debugging                  |    | <b>Basically, MLOps</b>                |
| 3. Profiling DL pipelines, tricks for efficient training                  |    | <b>Systems &amp; better training 2</b> |
| 4. OS recap, distributed ML recap   |   | <b>Distributed training</b>            |
| 5. Data-parallel training, All-Reduce, torch.distributed intro            |   |  |
| 6. Memory-efficient training, model parallelism                           |   |  |
| 7. Basics of web service deployment                                       |  | <b>Deployment in production</b>        |
| 8. Deploying neural networks: software side                               |   |  |
| 9. Optimizing models for inference  |   |  |

# Logistics

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

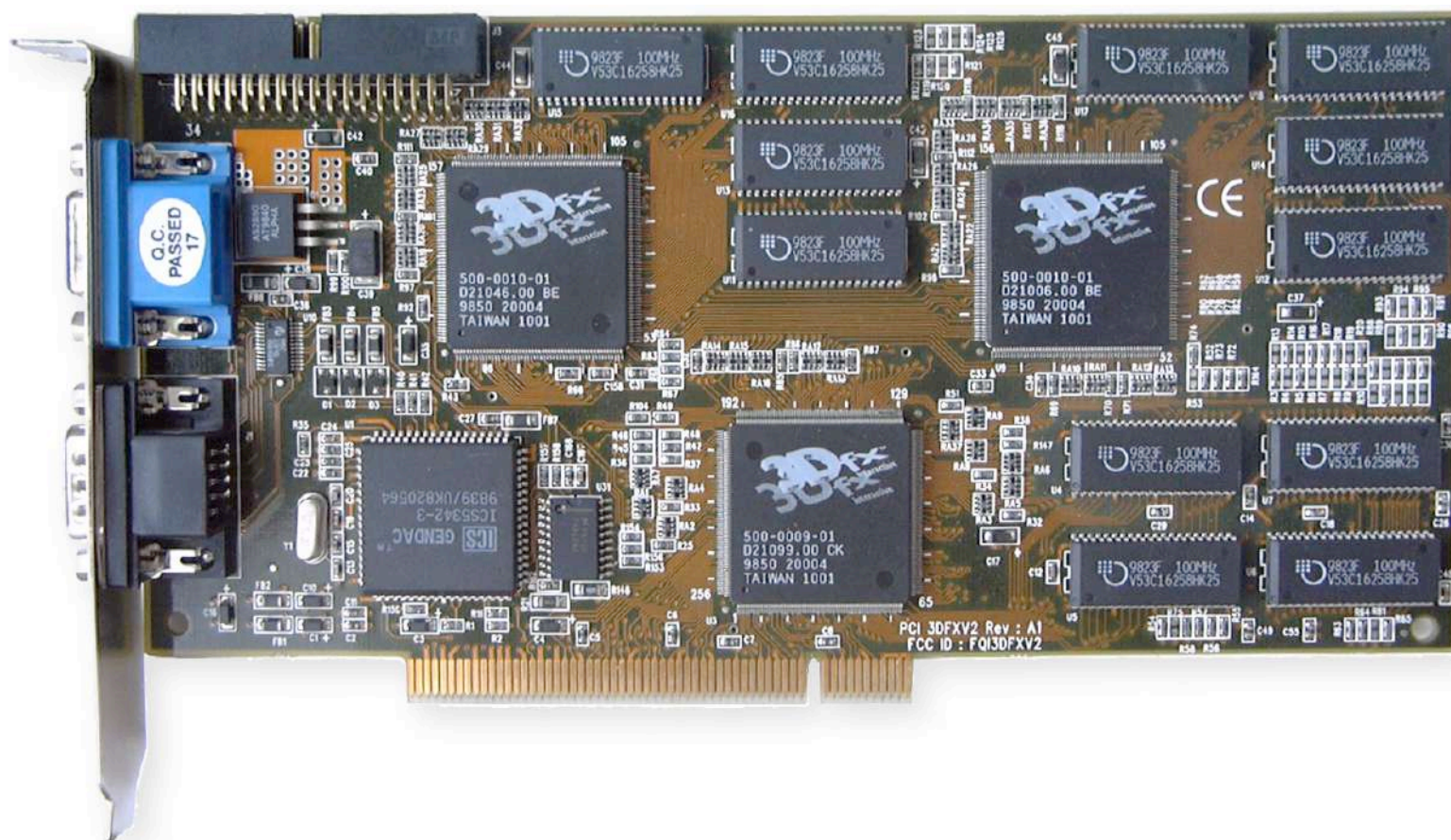
# Grading

- 3 assignments:
  1. Fast and reliable pipelines (2 parts)
  2. Distributed training (3 parts)
  3. Deployment (3 parts)
- Each assignment consists of sub-assignments given each week (except\* this one)
- Final grade:  $G_{total} = 0.2G_1 + 0.4G_2 + 0.4G_3$



# 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



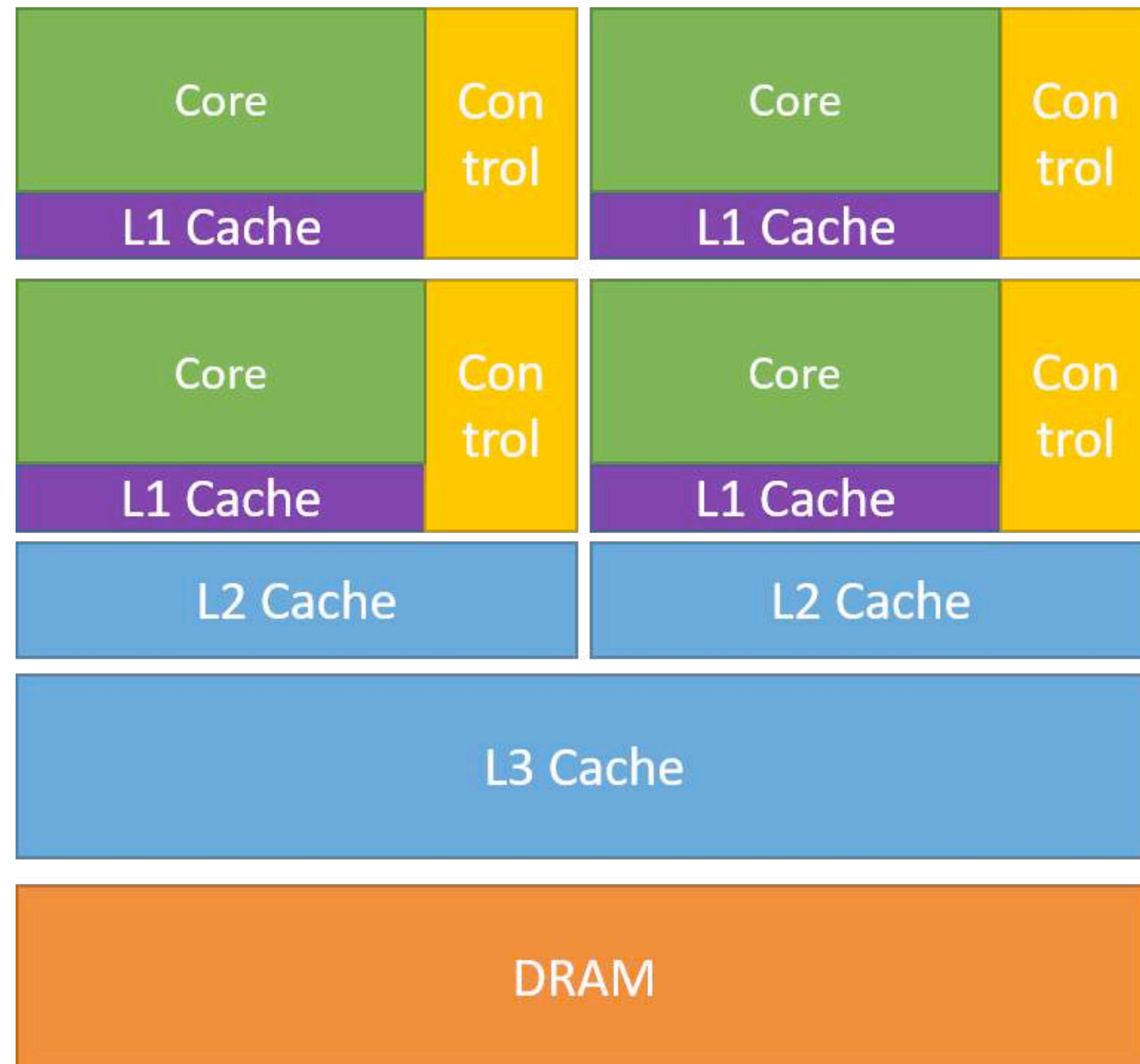
**3dfx Voodoo2: 12MB RAM**



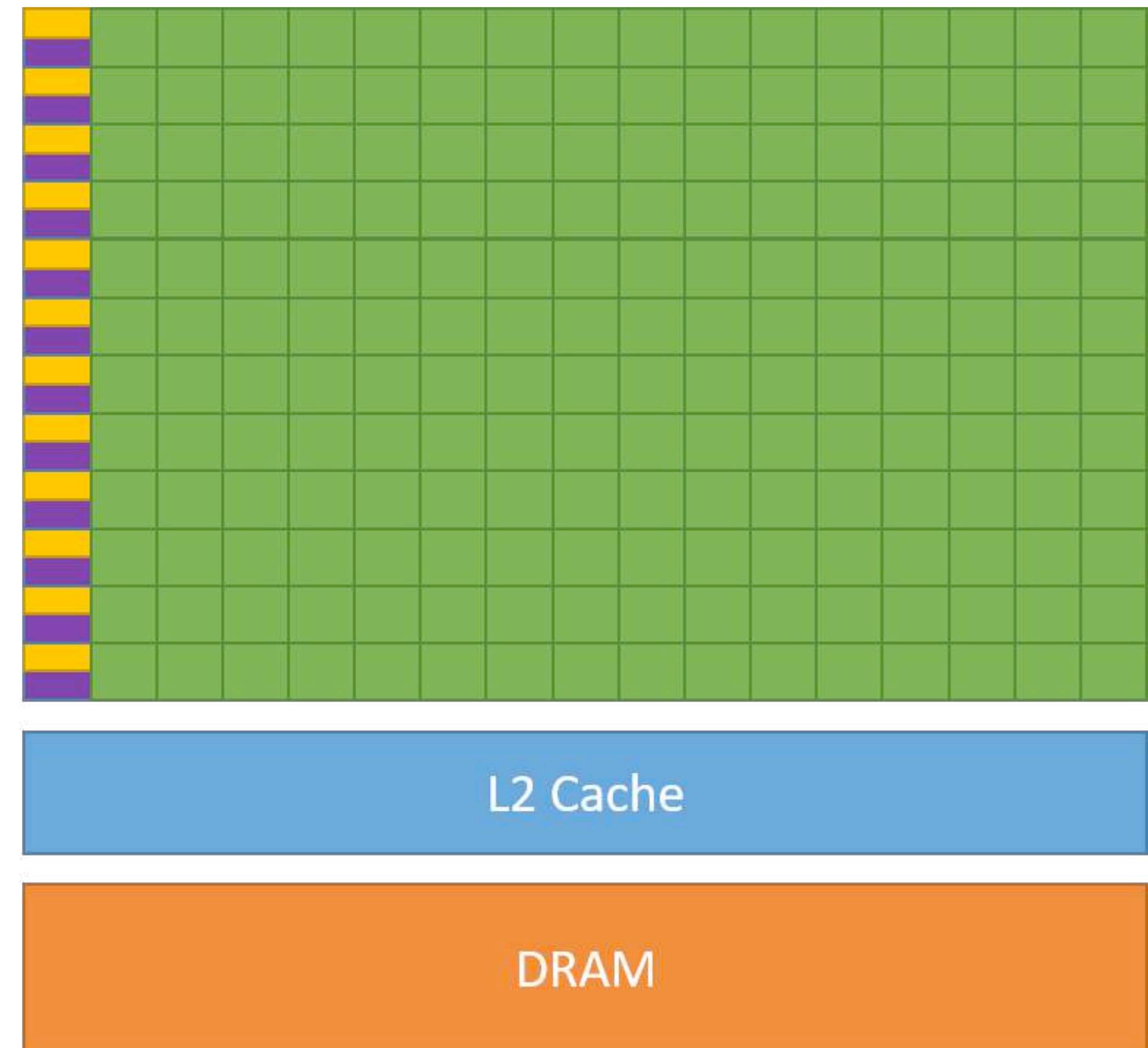
**NVIDIA H200: 141GB RAM**



# GPU architecture: a brief overview



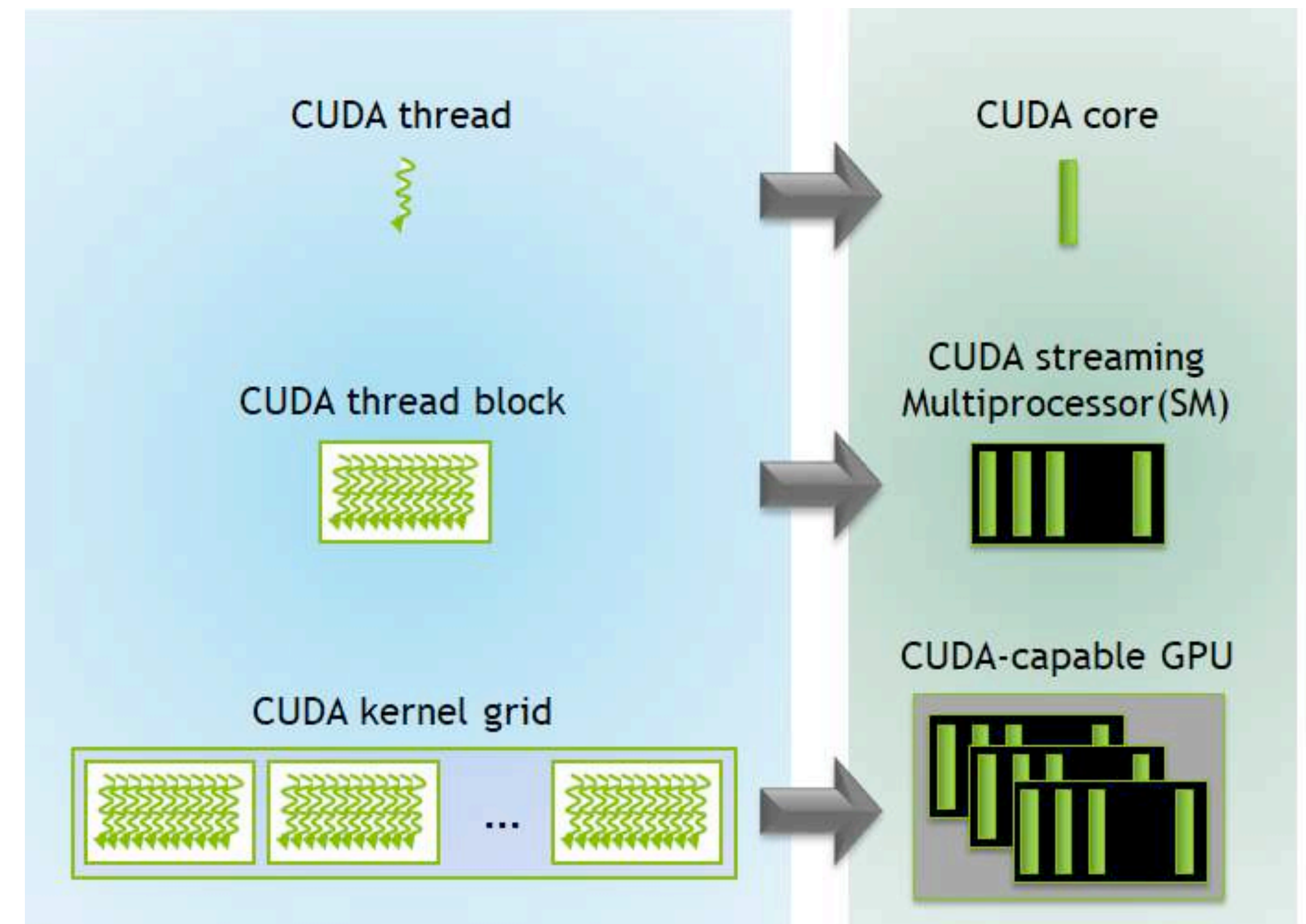
CPU



GPU

# CUDA computation model

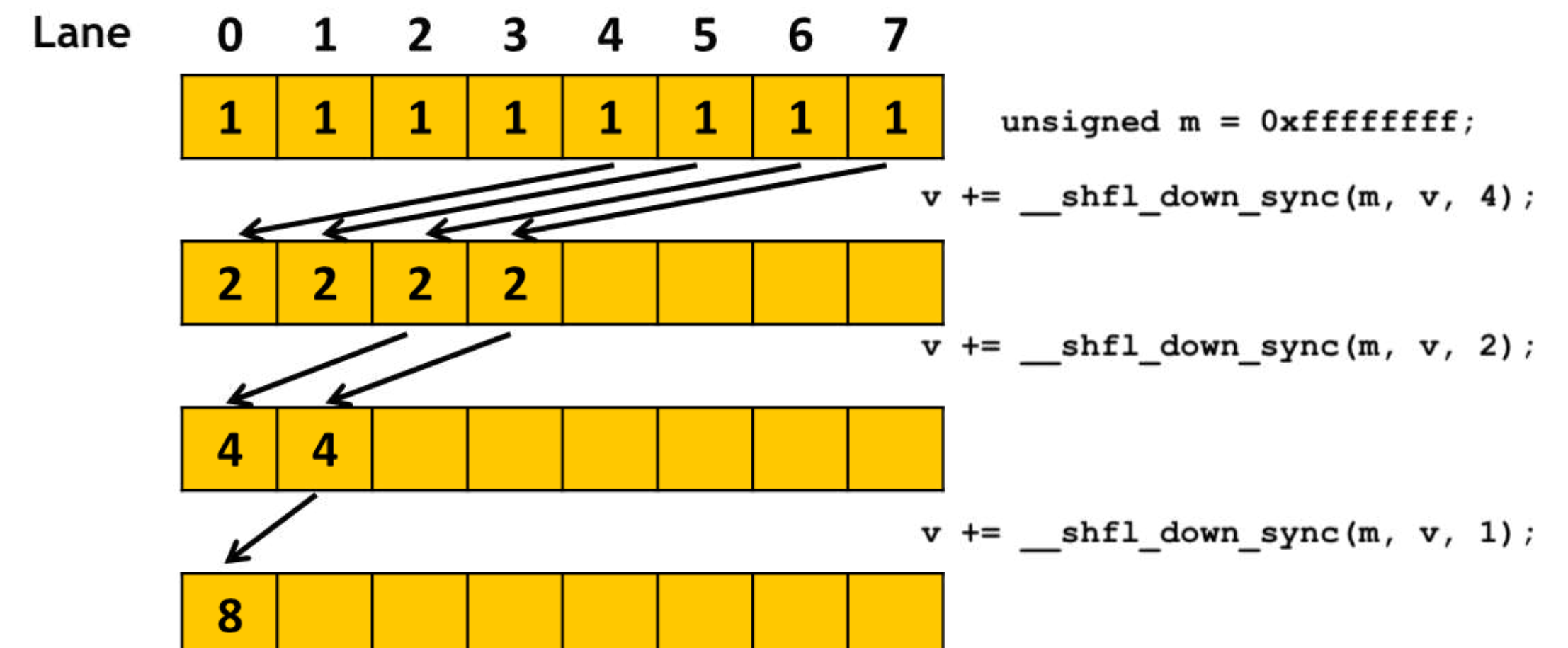
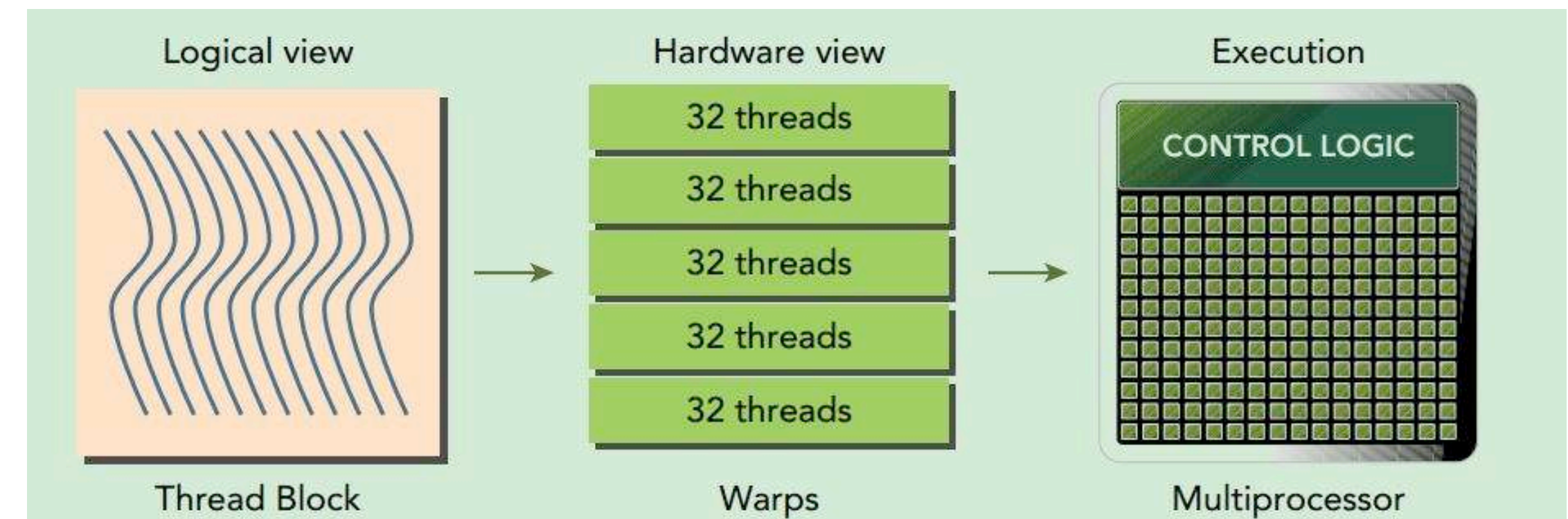
- In CUDA, we launch kernels from the host that are executed in parallel on the device
- Kernels are executed by threads grouped in thread blocks of limited size
- A GPU is composed of multithreaded Streaming Multiprocessors (SMs) that are assigned different thread blocks
- Multiple thread blocks are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





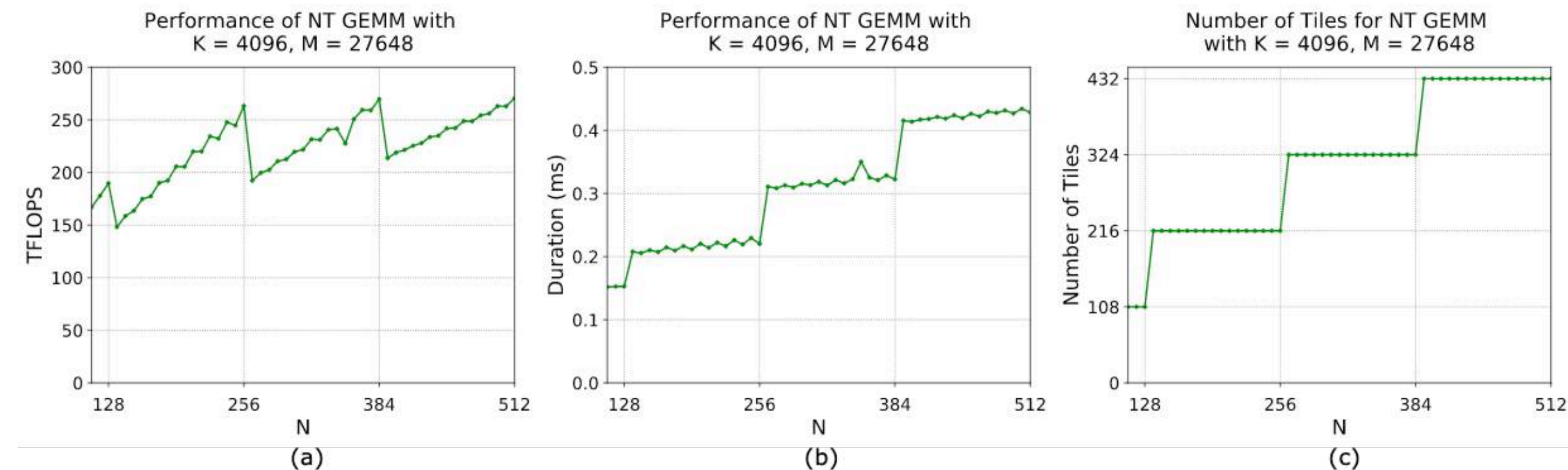
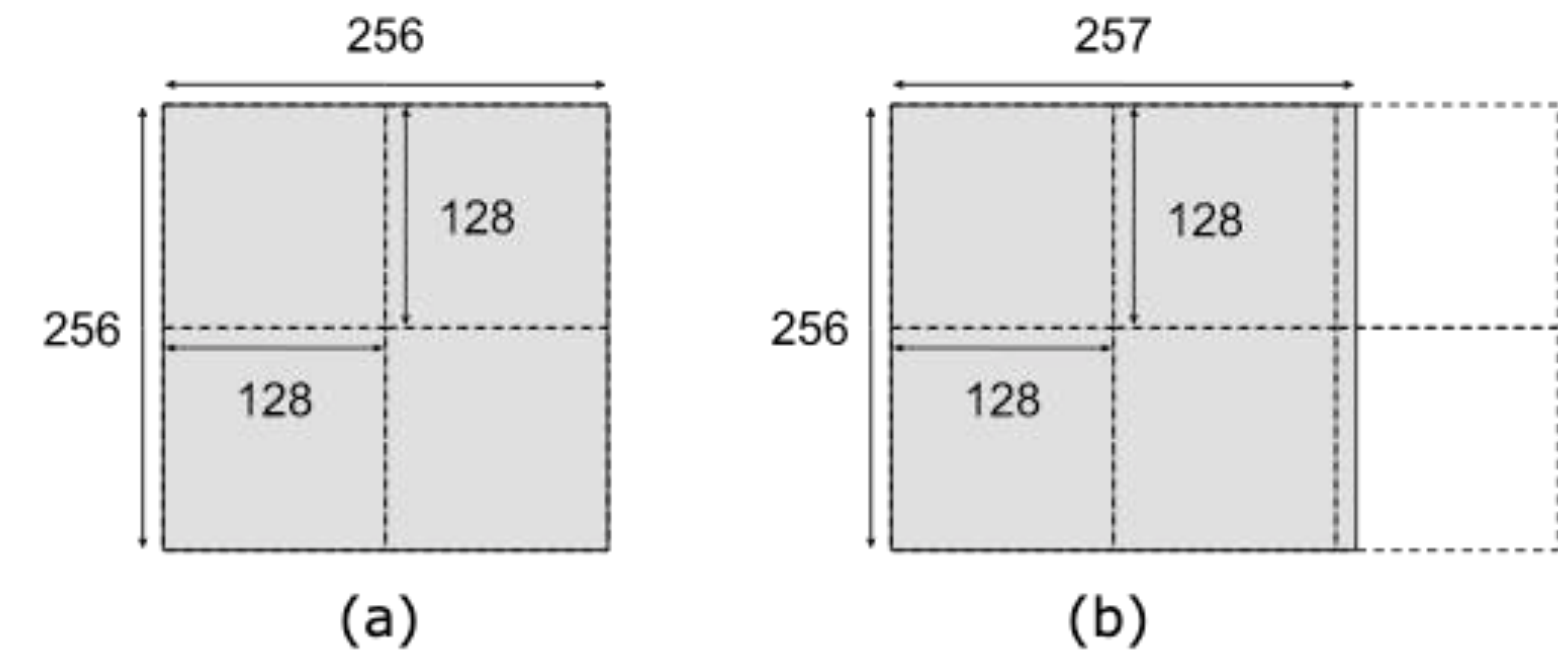
# GPU computations: hardware side

- 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



# Why does all of this matter?

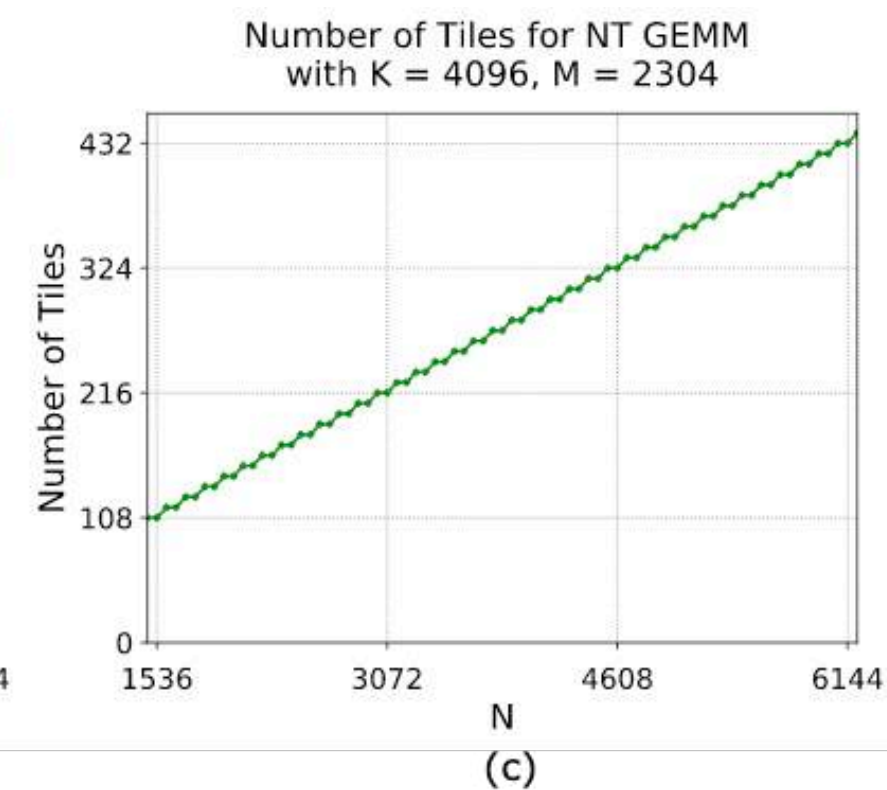
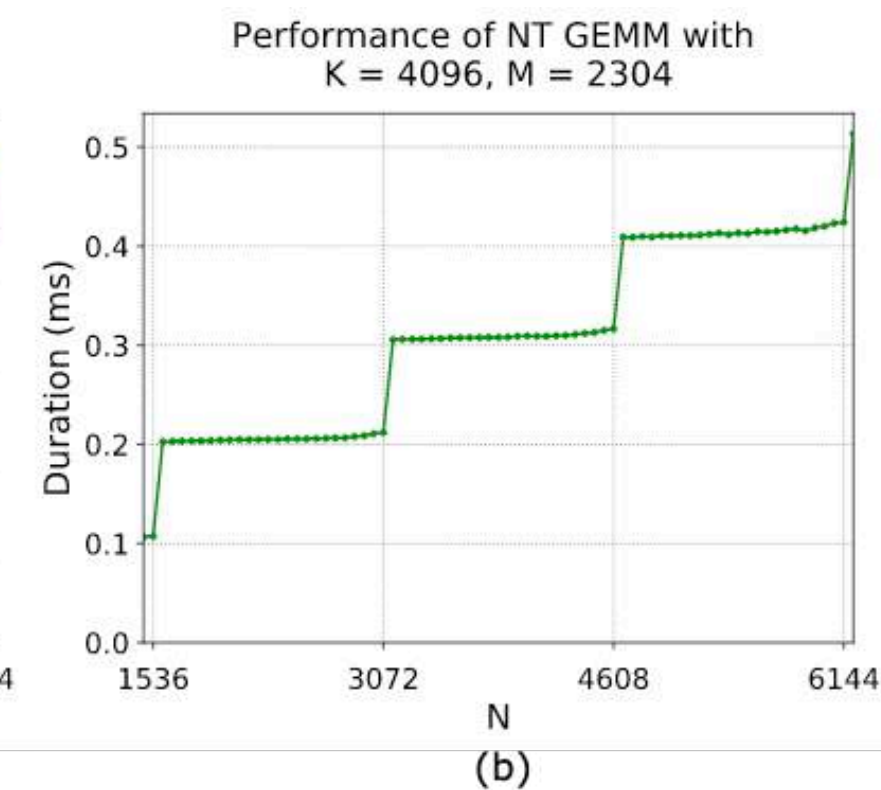
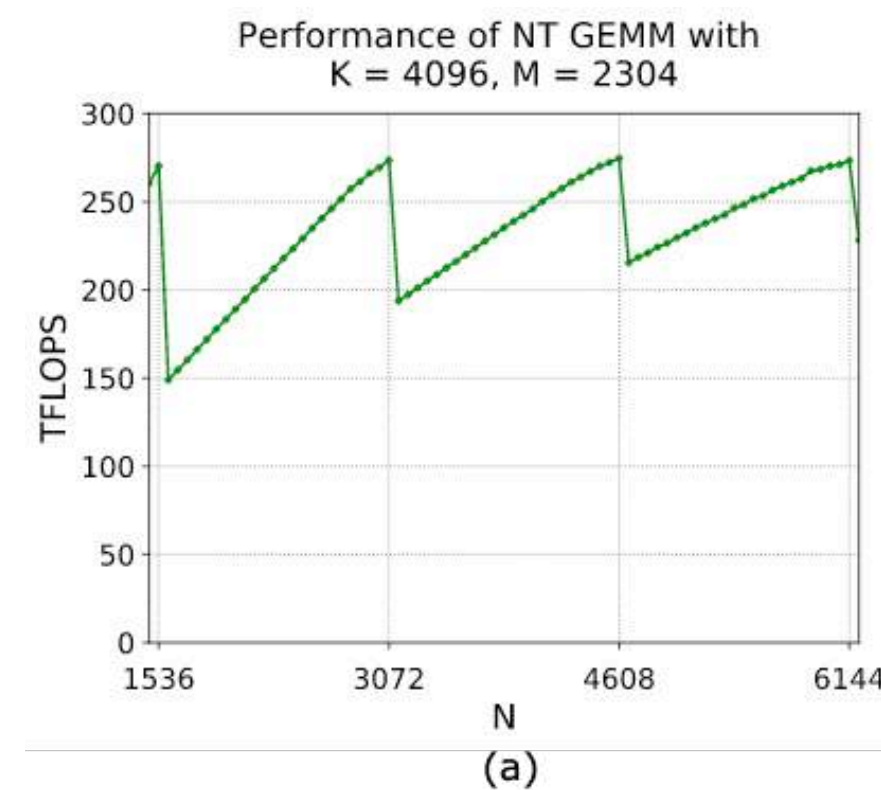
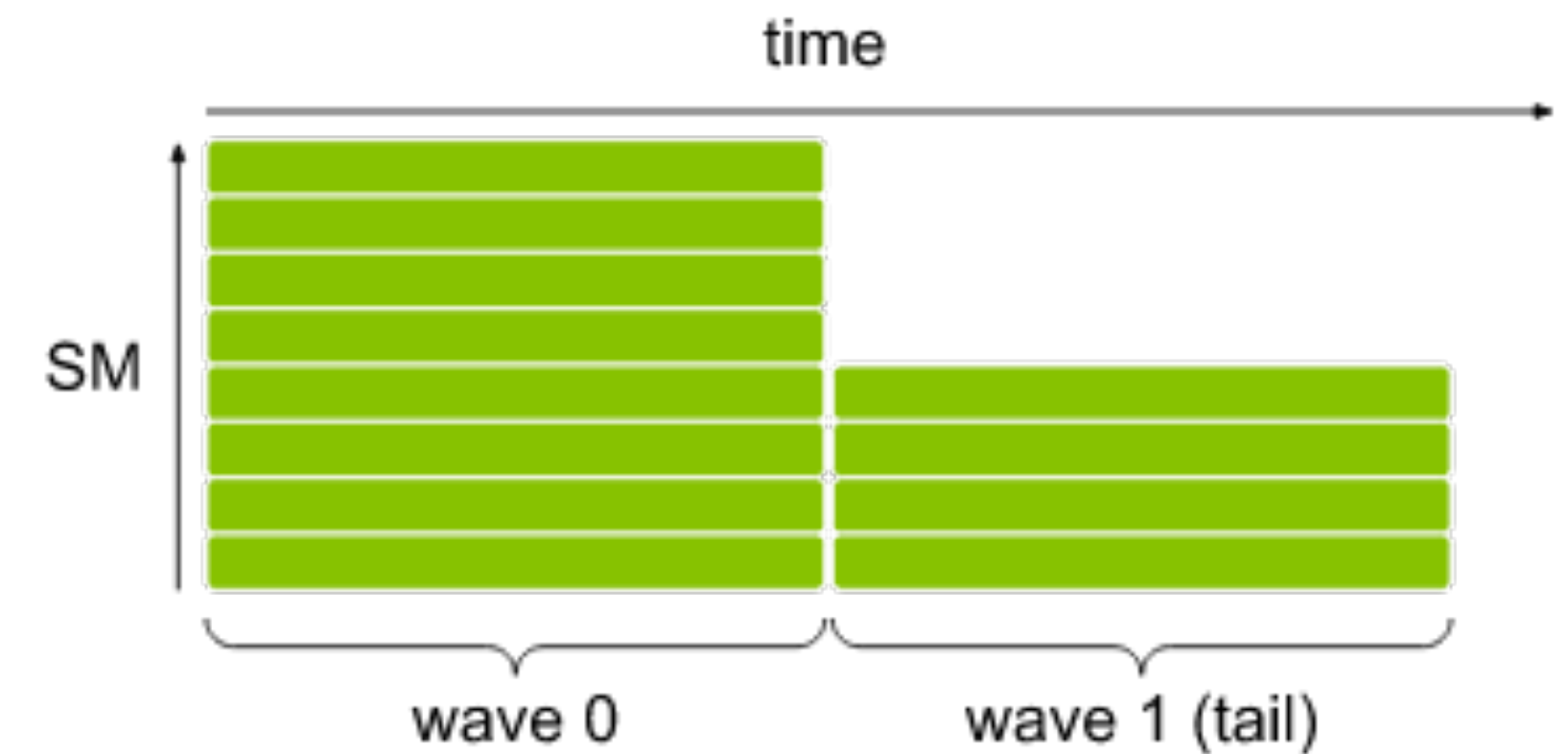
- The most popular operation in DL is matrix multiplication
- Executing this in parallel can have two potential effects when dividing the work
- **Tile Quantization:** matrix size is not divisible by the thread block tile size





# Why does all of this matter?

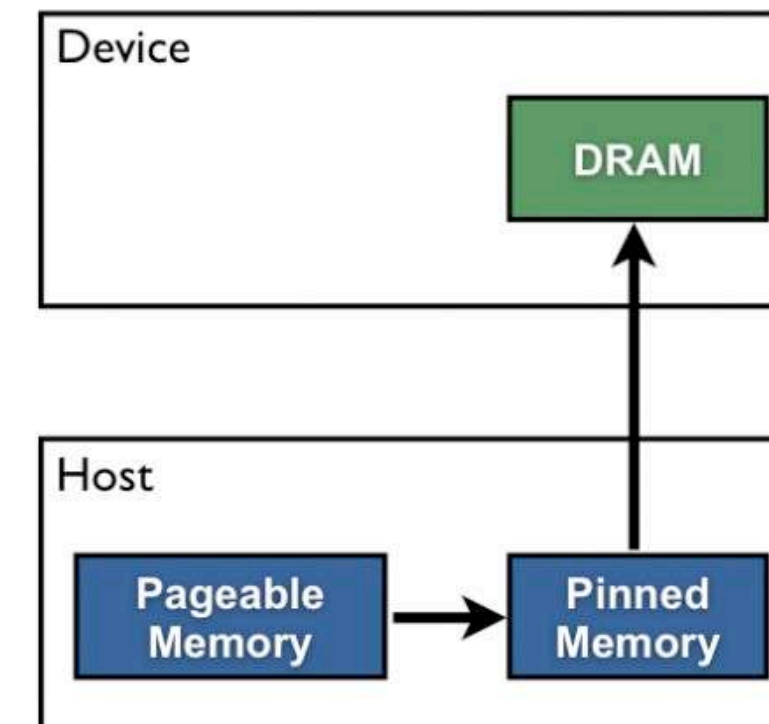
- The most popular operation in DL is matrix multiplication
- Executing this in parallel can have two potential effects when dividing the work
- **Tile Quantization:** matrix size is not divisible by the thread block tile size
- **Wave Quantization:** total number of tiles is quantized to the number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



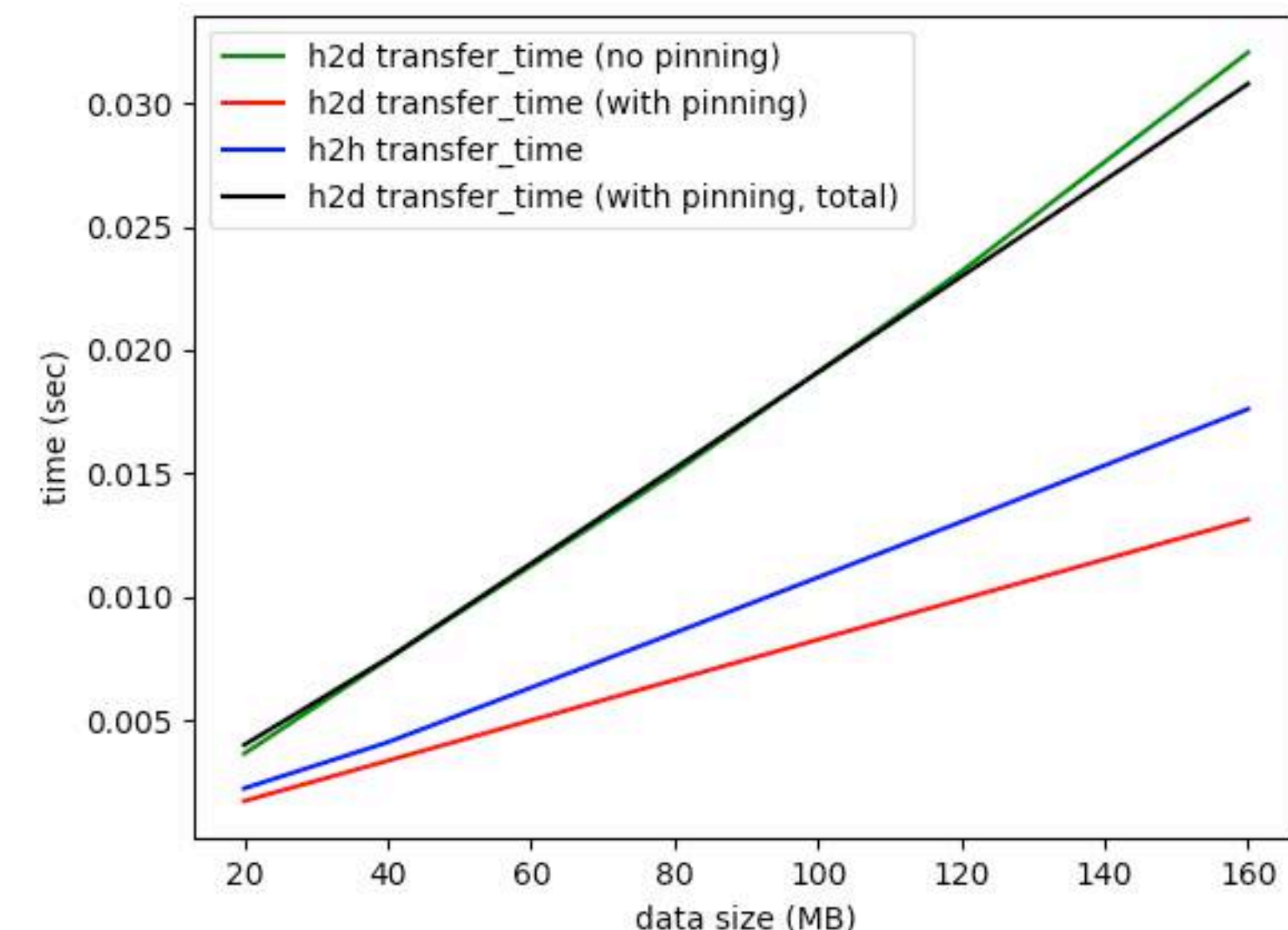
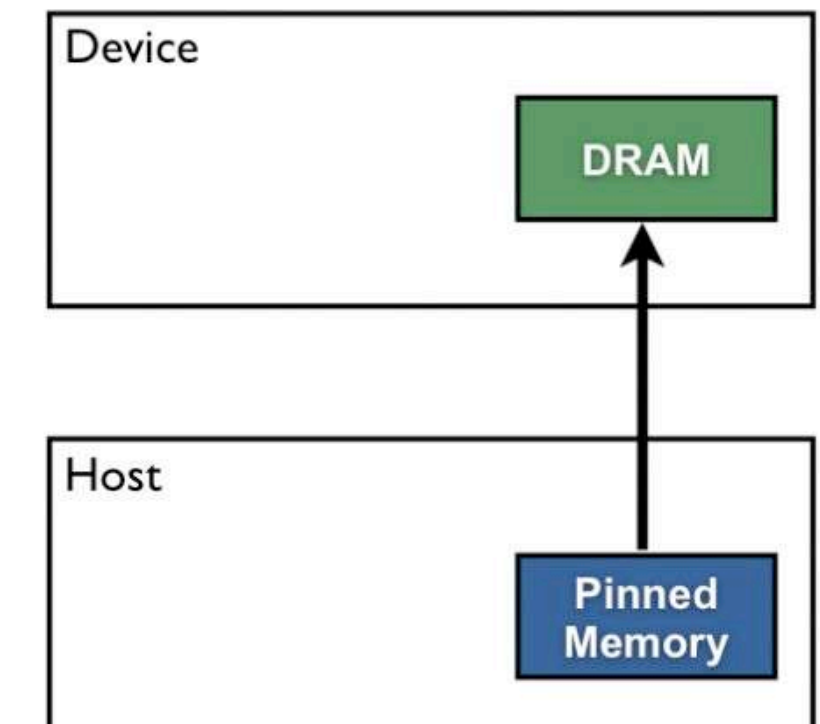
# Memory access

- 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!

*Pageable Data Transfer*



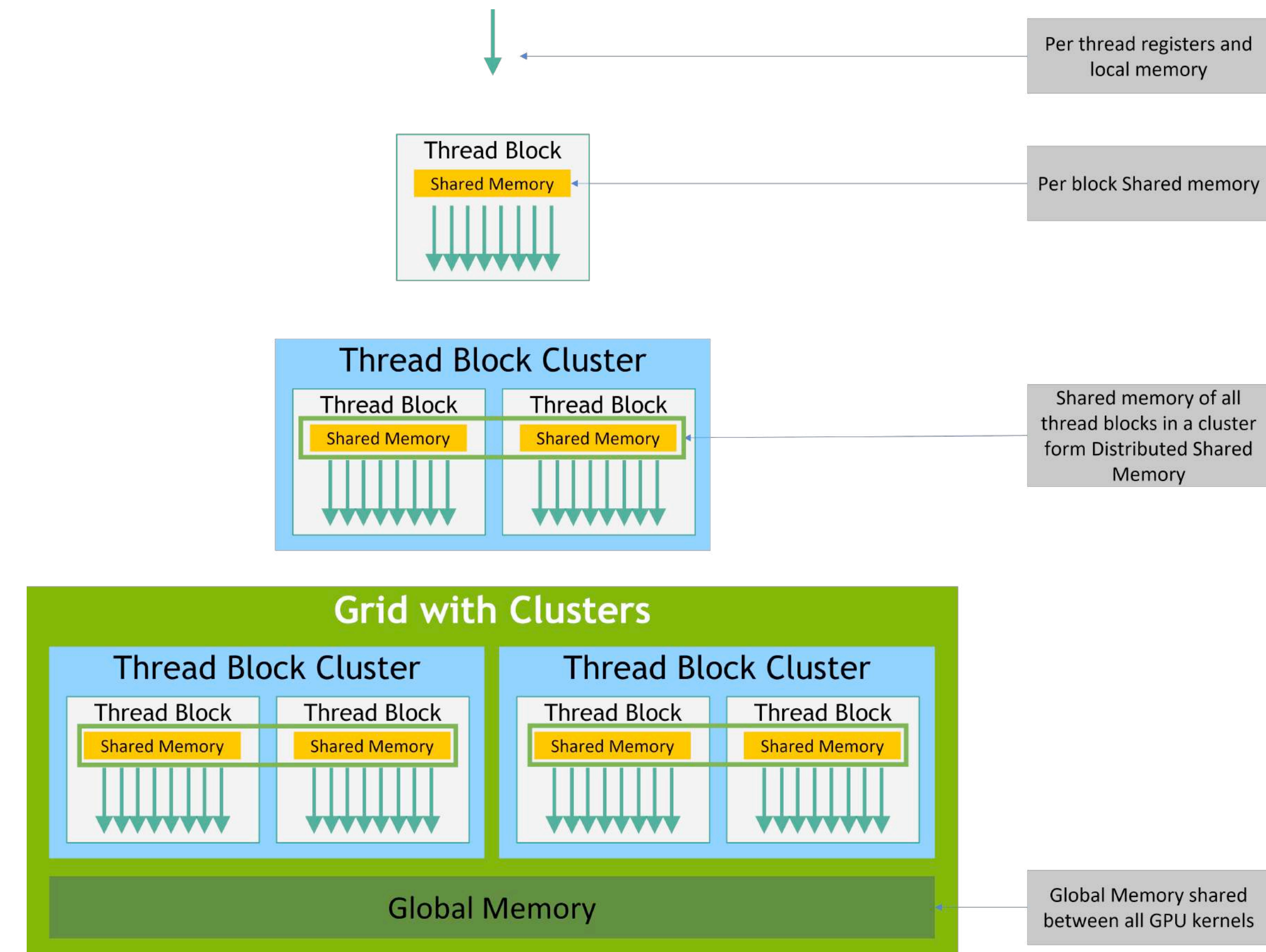
*Pinned Data Transfer*





# 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!



# 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)

## Sequential Version



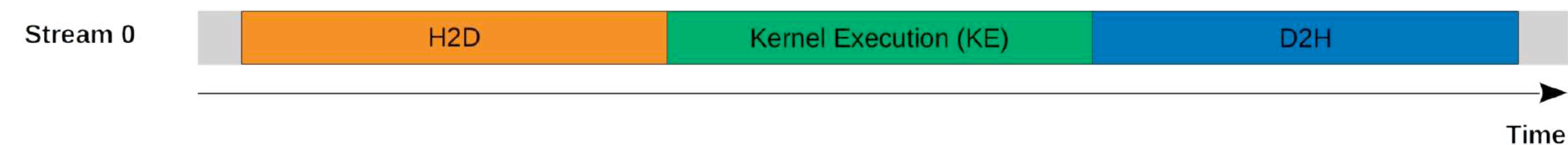
## Asynchronous Version 1



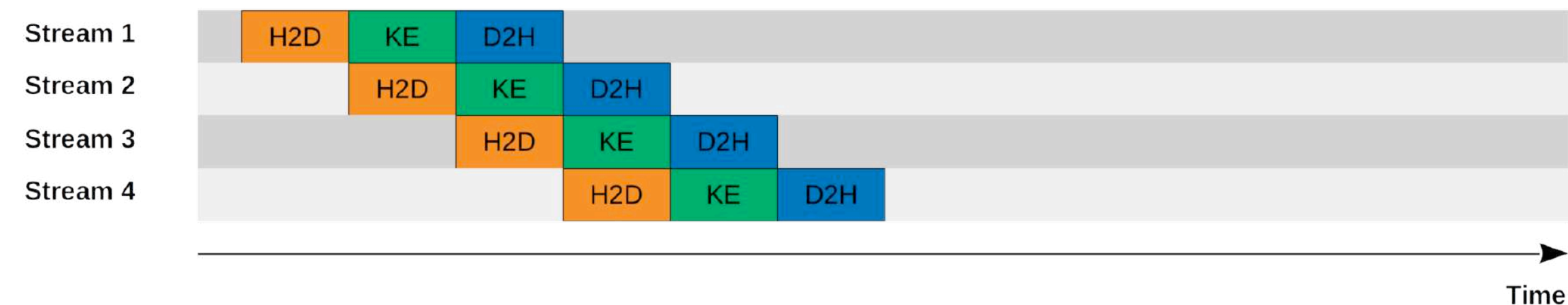
## Asynchronous Version 2



## Serial Model



## Concurrent Model



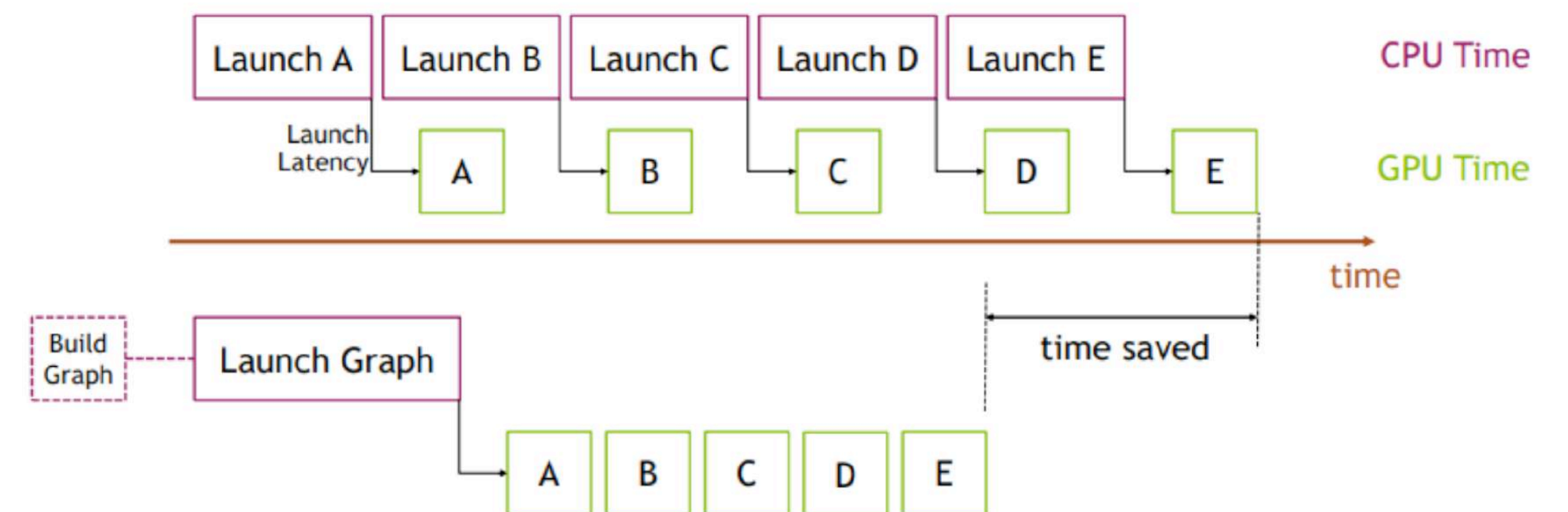
# DL specifics

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	<code>cudnn.benchmark = False</code> (the default)	<code>cudnn.benchmark = True</code>	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!**