Essentials of GPU, Deep Learning Bottlenecks

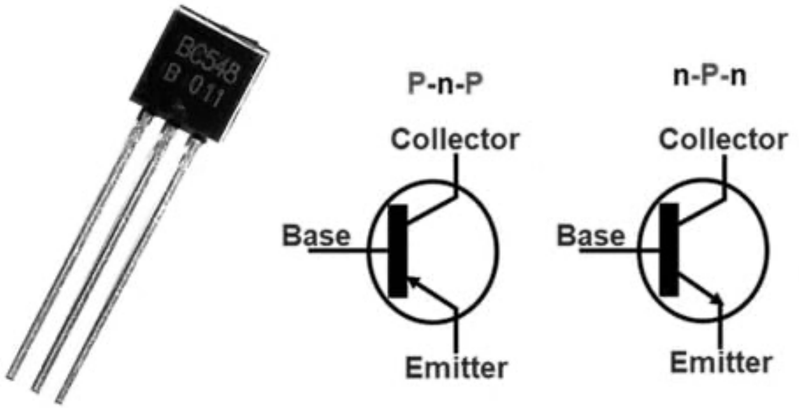
… and intro to benchmarking

Fedor Velikonivtsev

Why don’t we use only CPU?

Transistor

- Electronic switch - controls electric current on a device
- Fundamental building block of all modern chips
- **Process size** - dimension of the smallest feature (e.g. *transistor*) $\Longrightarrow$ can place more transistors on the same area
- More transistors $\Longrightarrow$ faster hardware



Moore’s law

- Number of transistors on a chip doubles every 18 months
- As time goes by, process size decreases
- Number of transistor grows exponentially

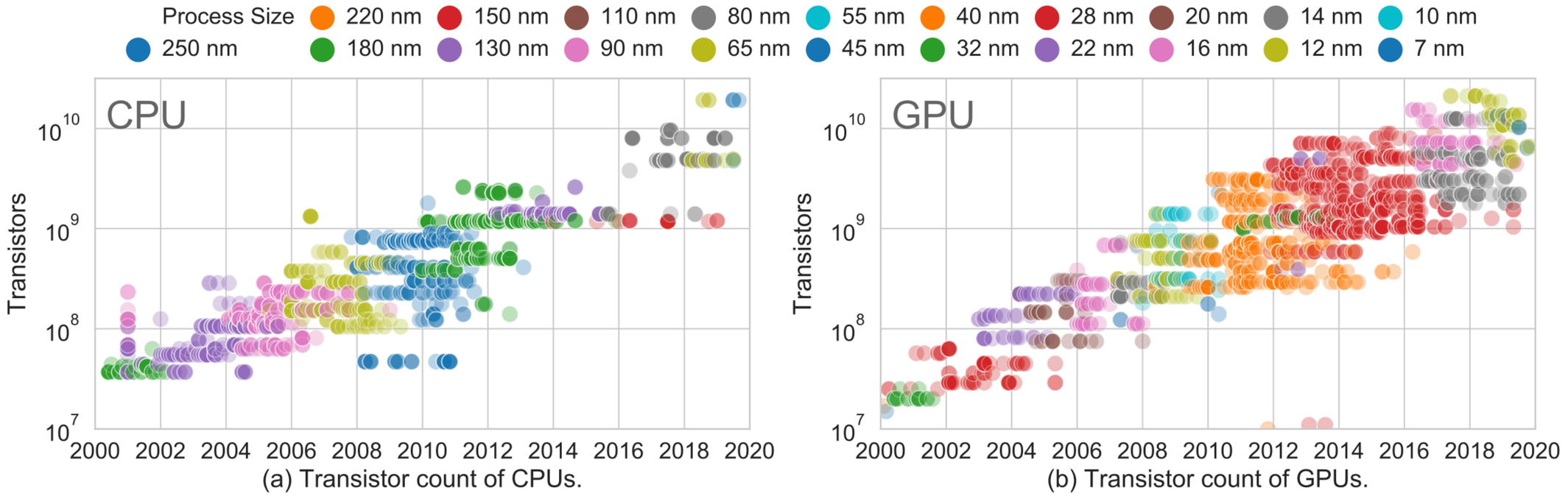


Fig. 1. Moore’s Law is still valid for both CPUs and GPUs.

https://arxiv.org/abs/1911.11313

Dennard Scaling law

- Energy consumption per transistor reduces with decreasing process size **(a)**
- Power usage stays constant in various process sizes **(b)**
- Clock frequency increases (hardware is faster) when you have more transistors per area block **(c)**

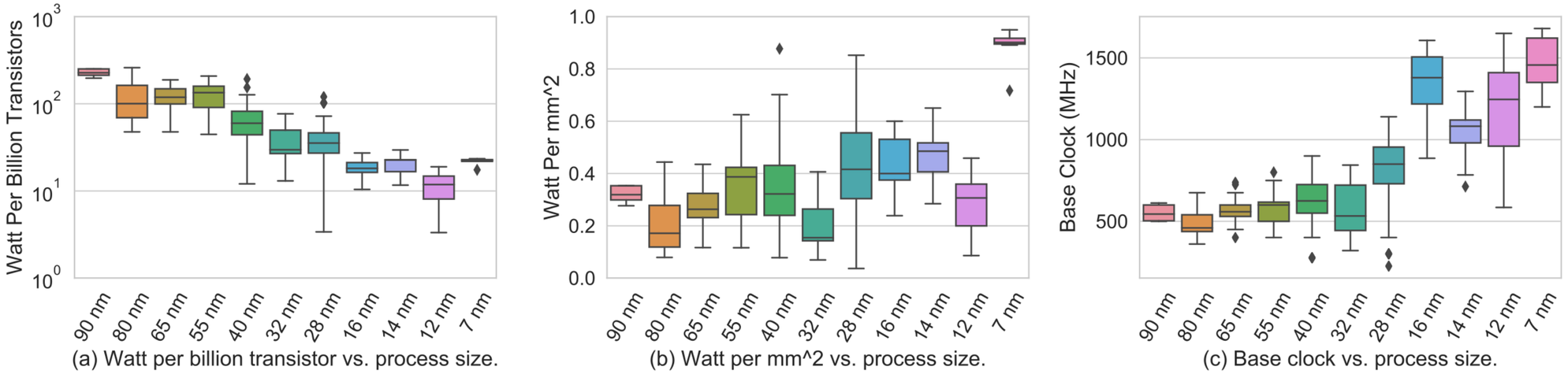


Fig. 4. Process Size vs. (a) Energy consumption per billion transistor. (b) Energy consumption per area. (c) Base clock speed.

* Figure represents characteristics of GPUs, but this is applicable for CPUs too

https://arxiv.org/abs/1911.11313

Dennard Scaling law

- Energy consumption per transistor reduces with decreasing process size **(a)**
- Power usage stays *kinda* constant in various process sizes **(b)**
- Clock frequency increases (hardware is faster) when you have more transistors per area block **(c)**

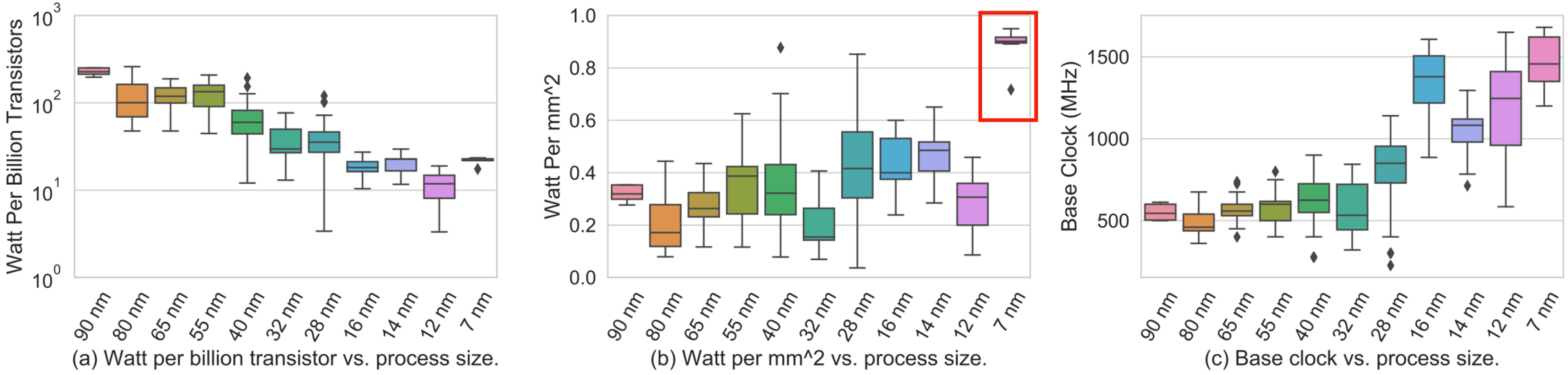


Fig. 4. Process Size vs. (a) Energy consumption per billion transistor. (b) Energy consumption per area. (c) Base clock speed.

* Figure represents characteristics of GPUs, but this is applicable for CPUs too

https://arxiv.org/abs/1911.11313

Fundamental Physical Limits to Transistor Size

- At 4-5nm scale transistors become affected by random electron fluctuations
- Chip’s circuits are manufactured with photolithography — you can’t create circuits smaller half the frequency of the light wave (~10 nm)

https://asteriskmag.com/issues/03/the-transistor-cliff

Fundamental Physical Limits to Transistor Size

- At 4-5nm scale transistors become affected by random electron fluctuations
- Chip's circuits are manufactured with photolithography — you can't create circuits smaller half the frequency of the light wave (~10 nm)
- **Takeaway:** we are limited in scaling hardware performance

https://asteriskmag.com/issues/03/the-transistor-cliff

Fundamental Physical Limits to Transistor Size

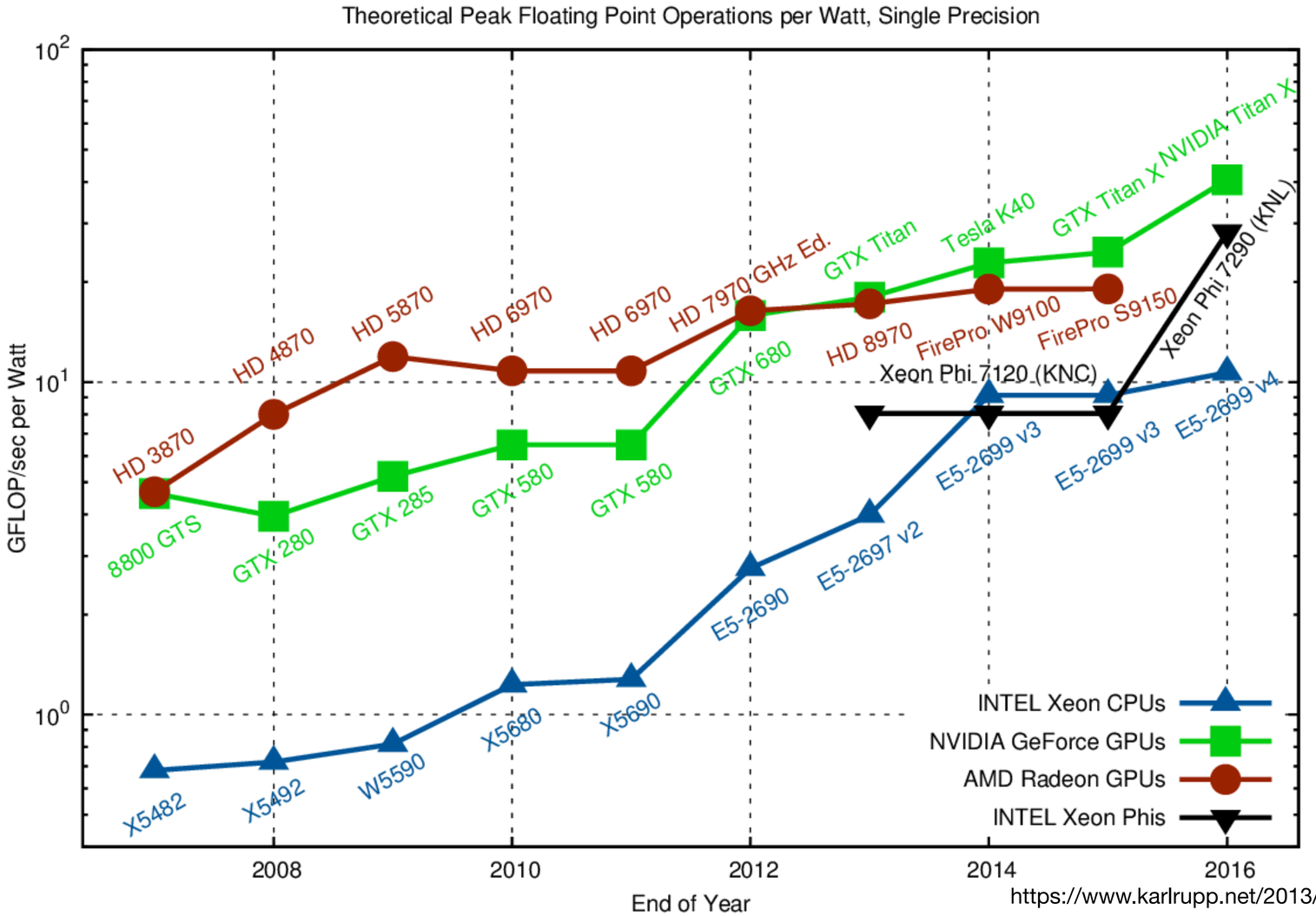
- At 4-5nm scale transistors become affected by random electron fluctuations
- Chip's circuits are manufactured with photolithography — you can't create circuits smaller half the frequency of the light wave (~10 nm)

- **Takeaway:** we are limited in scaling hardware performance, **but we can optimize energy consumption**

https://asteriskmag.com/issues/03/the-transistor-cliff

GPUs are more energy efficient than CPUs

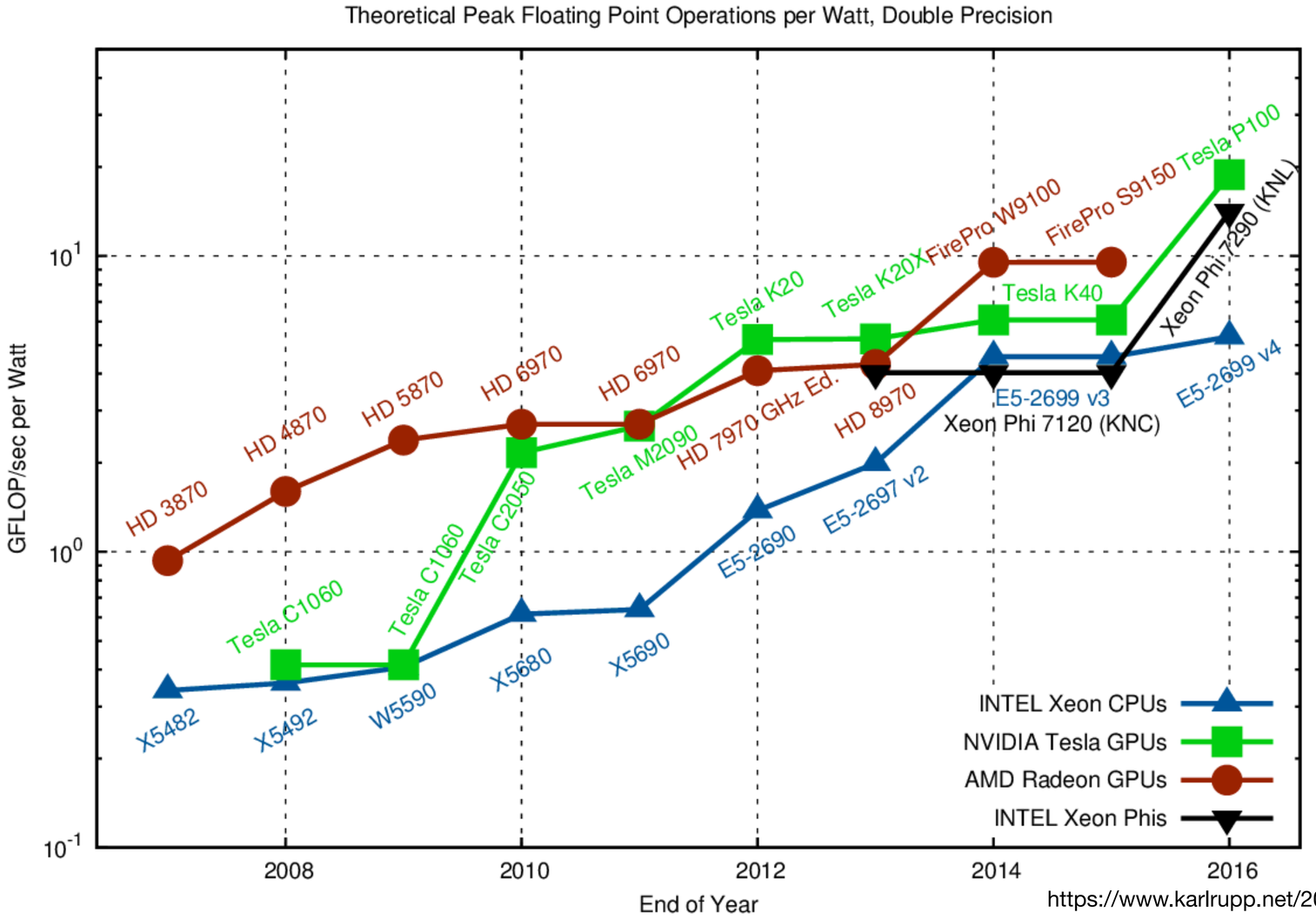
- You can get more performance per Watt on GPUs for numerical operations



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

GPUs are more energy efficient than CPUs

- You can get more performance per Watt on GPUs for numerical operations

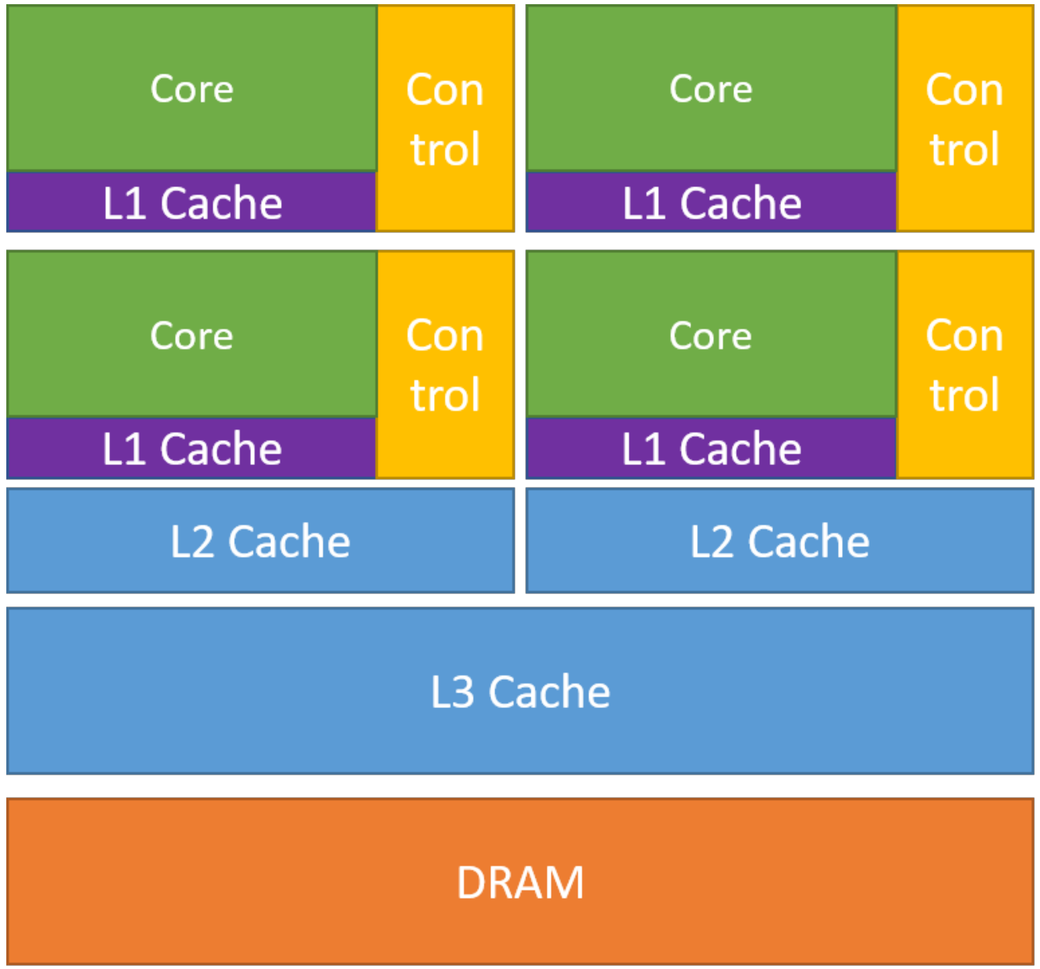


https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

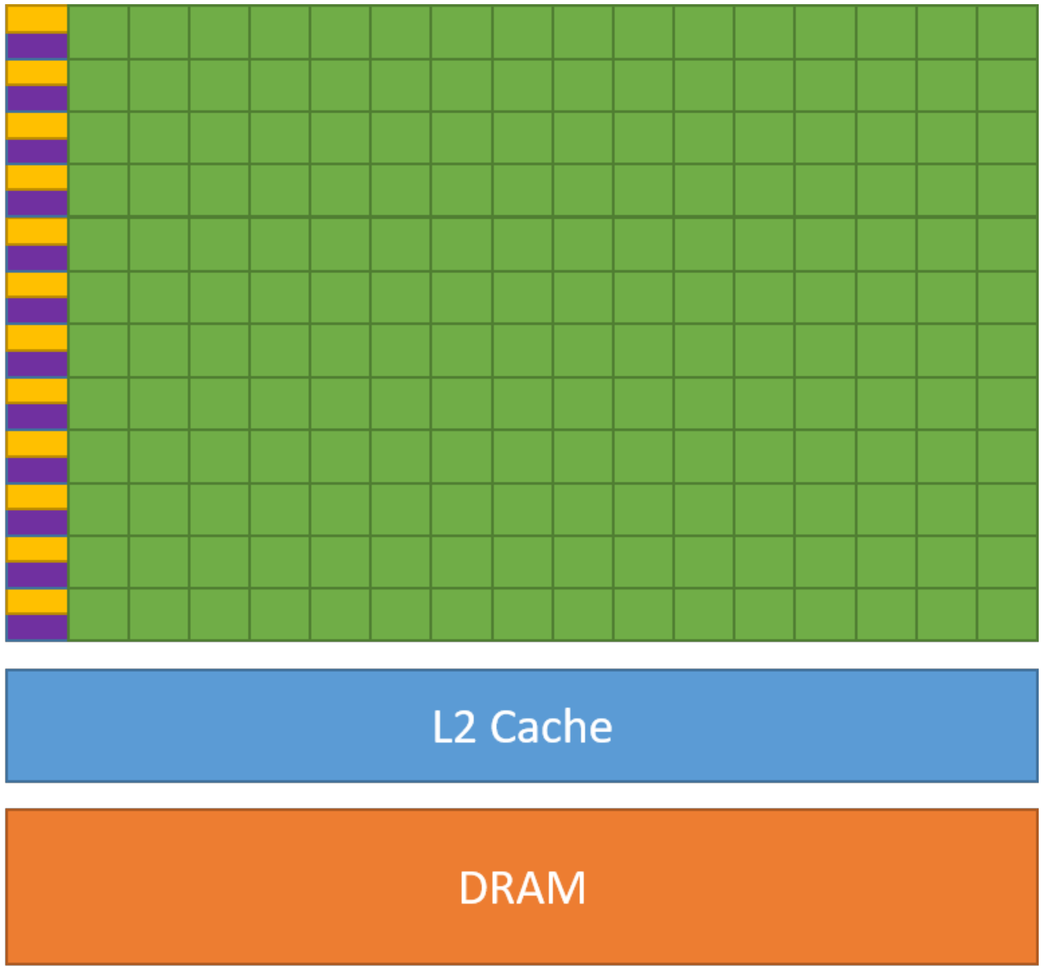
GPU programming paradigm

*Nvidia GPUs

CPU vs. GPU architecture

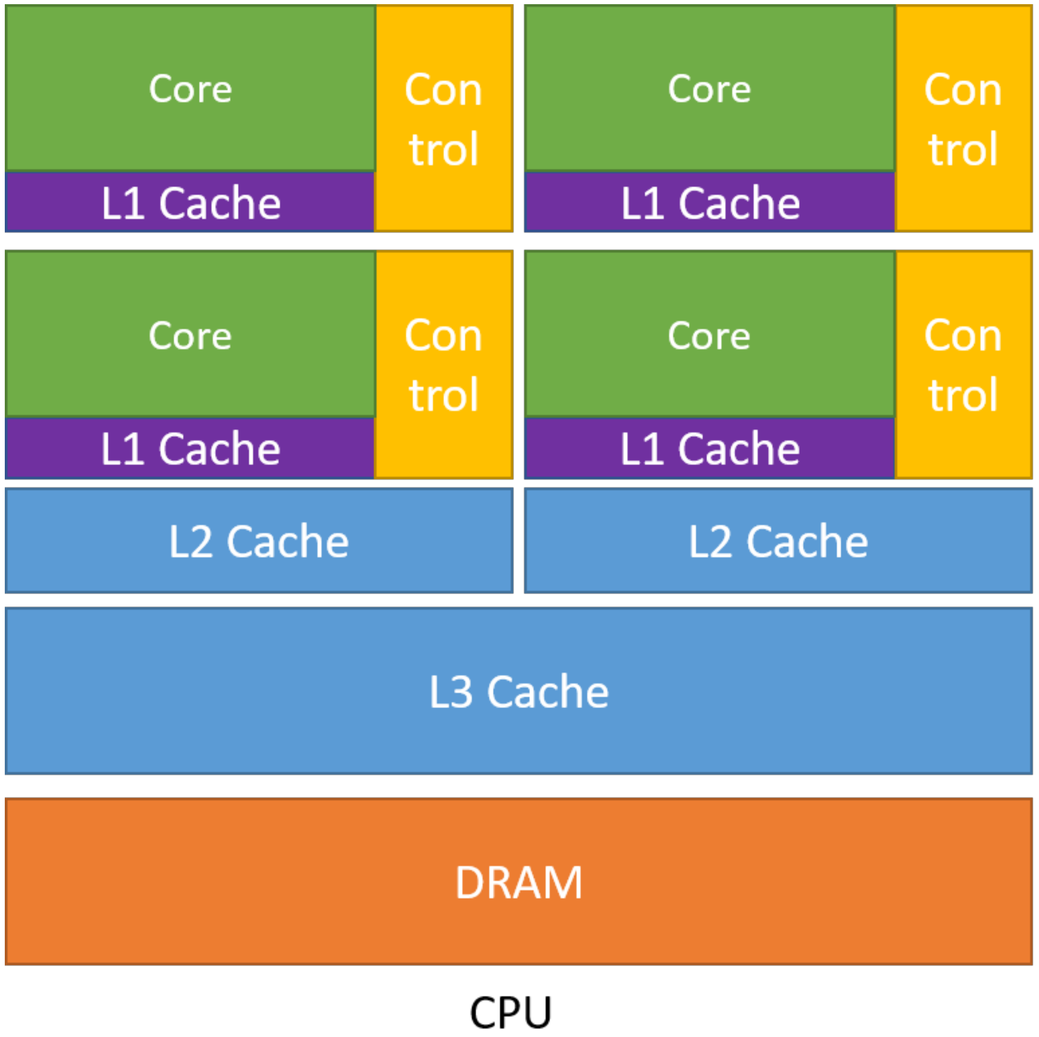


CPU



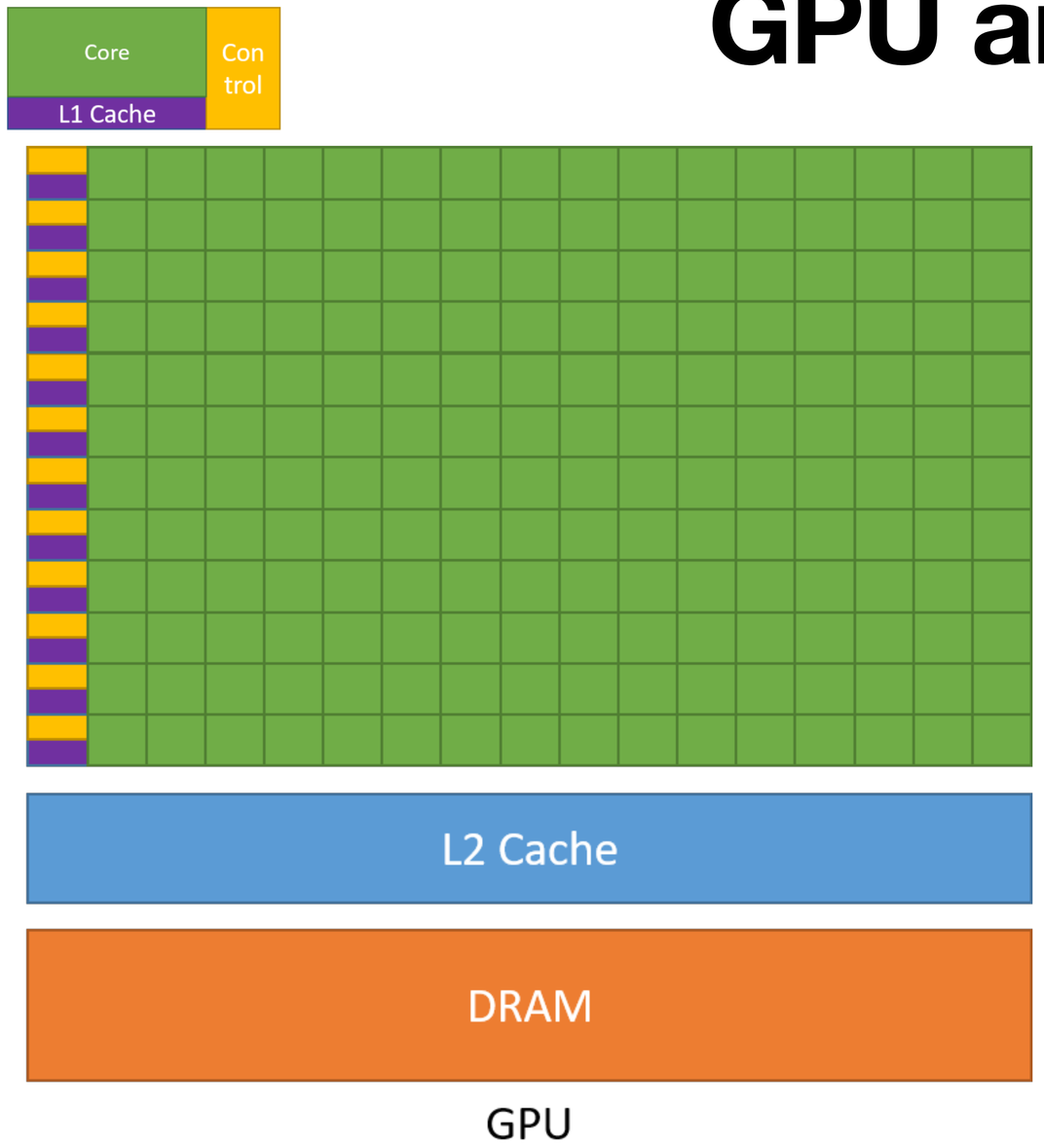
GPU

CPU architecture



- More sophisticated control logic
- More transistors to data caching & flow control
- Better at executing **sequential** operations
- Fast single-thread performance

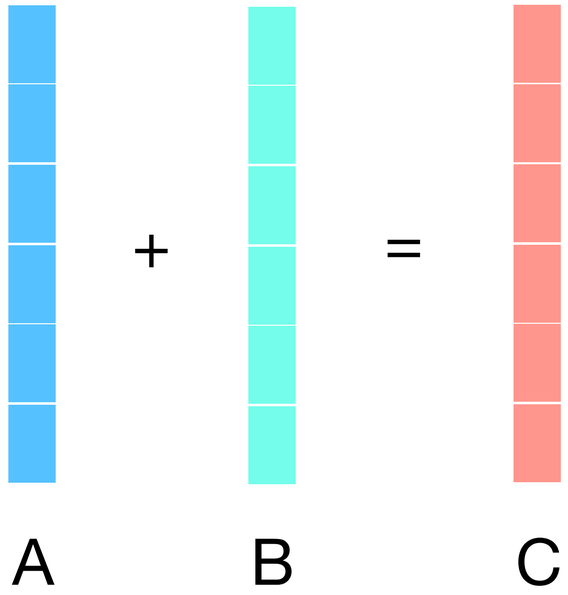
GPU architecture



- Less instructions, simpler logic
- More transistors to data processing
- The best at parallel operations
- Slow single-thread performance (due to latency), **amortized by parallel computation**
- **Higher instruction throughput compared to CPU with similar price and power consumption**

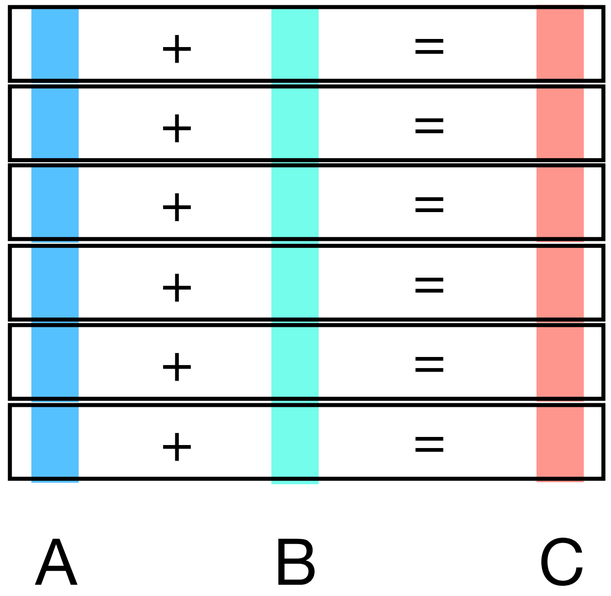
CUDA model for GPU computation

- GPU execute **kernels** - a functions executed on a device in parallel (*addition, matrix multiplication, etc.*)
- Kernel is launched in parallel on different parts of GPU
- Each part computes a portion of a result:



CUDA model for GPU computation

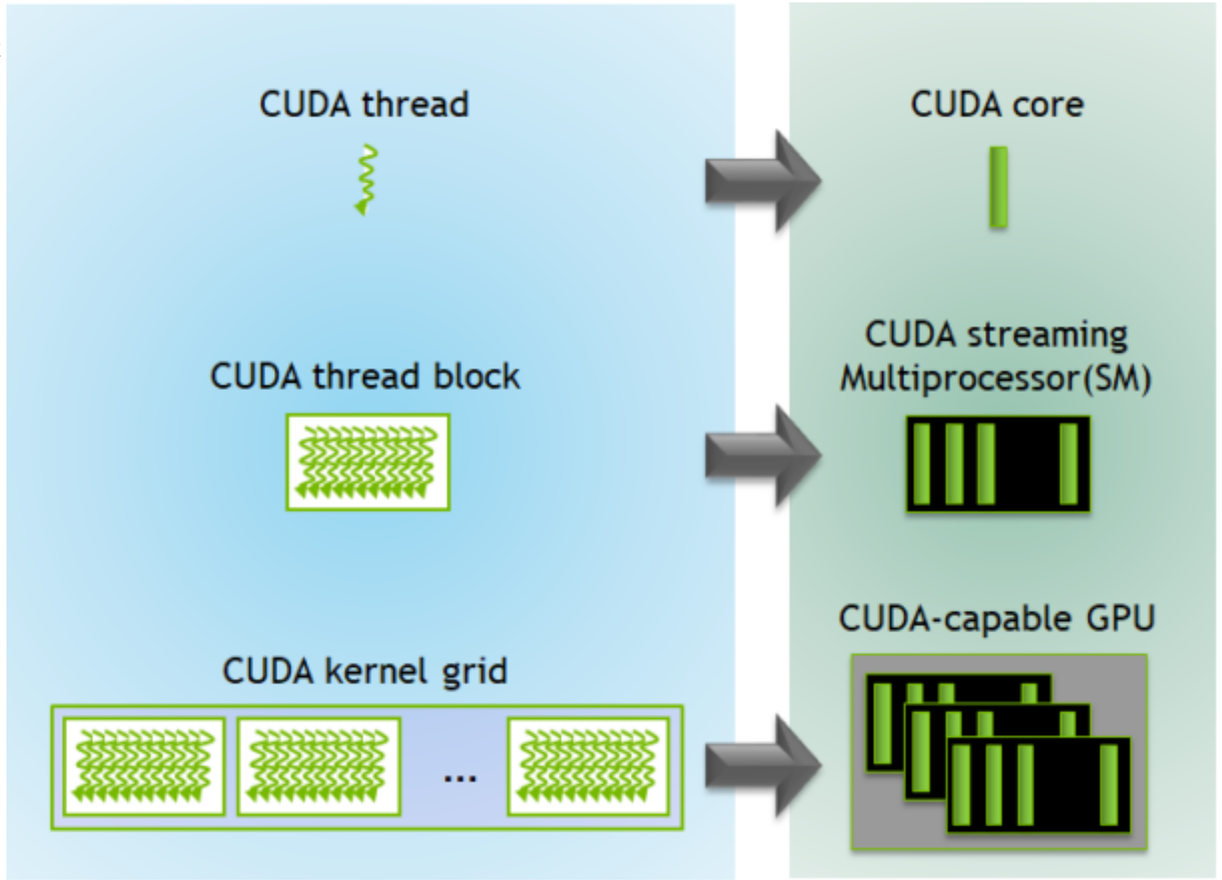
- GPU execute **kernels** - a functions executed on a device in parallel (*addition, matrix multiplication, etc.*)
- Kernel is launched in parallel on different parts of GPU
- Each part computes a portion of a result:



Parallel execution

Physical level for CUDA model

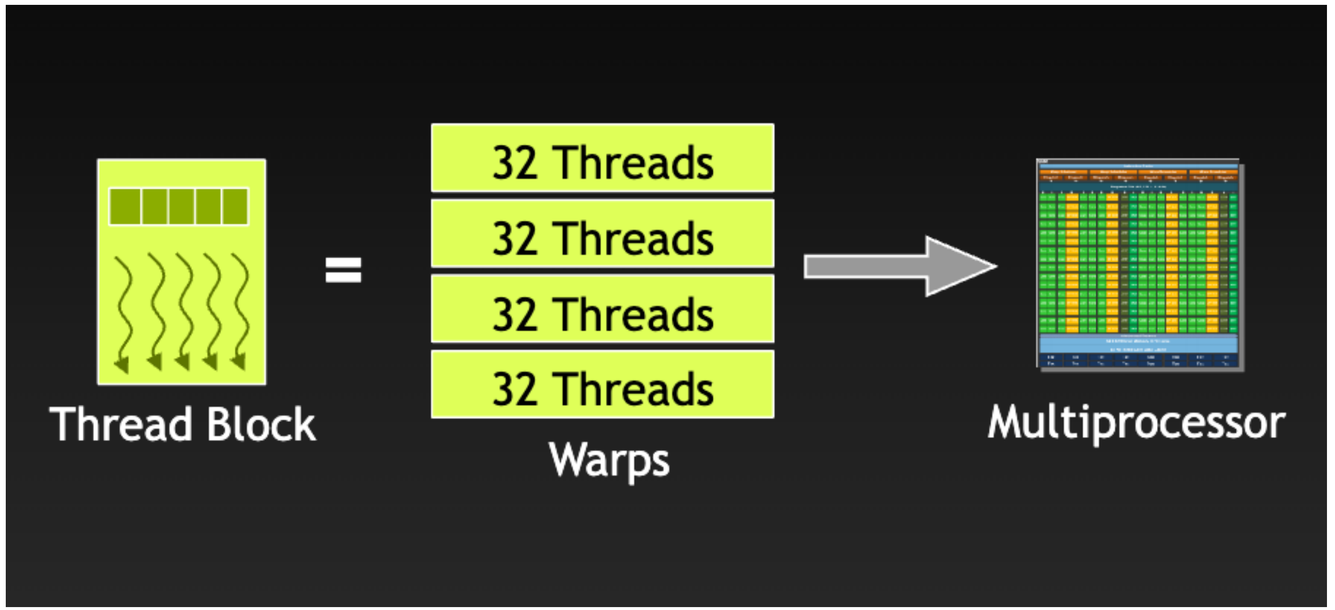
- **Streaming Multiprocessor** (SM) - basic operational unit on a GPU. Contains computation and data loading logic.
- Program launches **threads** - number of threads on SM is limited
- Threads are grouped in **thread blocks**
- Thread blocks are grouped in **grids**
- **Thread blocks** and **grids** can be logically arranged in 1D, 2D, 3D



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

Physical level for CUDA model

- Threads in a thread block are guaranteed to launch on the same SM
- Threads are organized and launched in **warps -** groups of *32* threads
- If there are less threads needed, warp will be **padded** with inactive threads

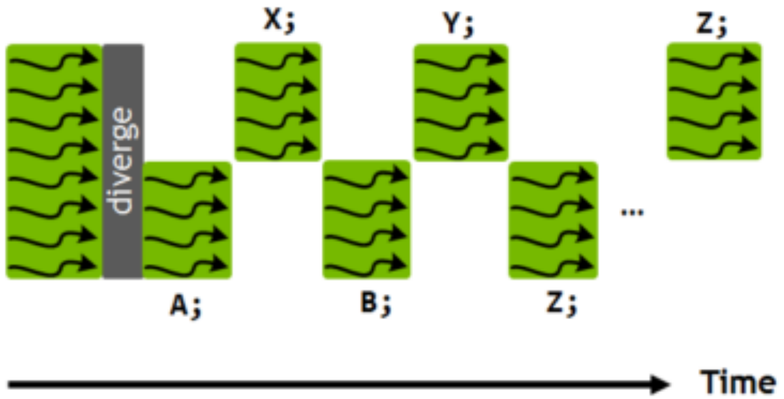


https://www.digitalocean.com/community/tutorials/the-role-of-warps-in-parallel-processing

Physical level for CUDA model

- Threads are organized and launched in **warps -** groups of *32* threads
- Threads within a warp can utilize warp-level primitives to communicate
- **Single Instruction Multiple Threads** (SIMT) - threads in a warp execute the **same instruction at a clock cycle**
  - Branch divergence is processed accordingly (*and can affect performance*):

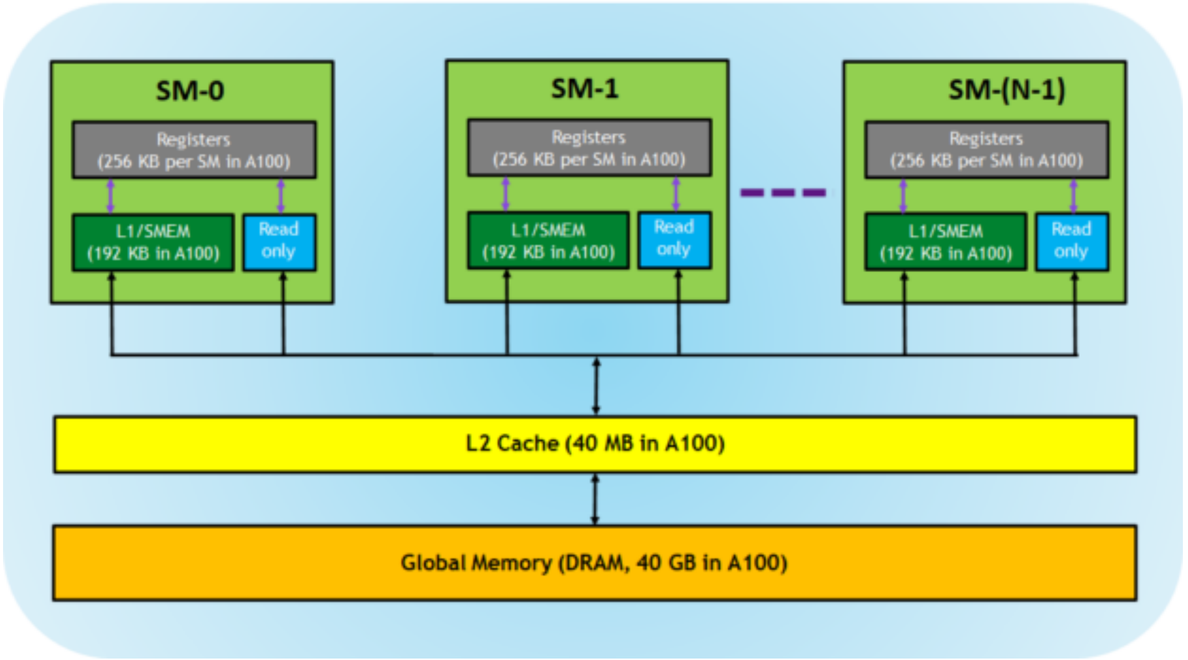
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



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

GPU memory

- **Global memory** (HBM, high-bandwidth memory) - a lot, but **slow**. Frequently accessed elements are stored in L2 cache
- Shared memory - shared across SM - *much* smaller, but **fastest**
- Runtime variables are stored in **registers**
- Memory access can bottleneck execution

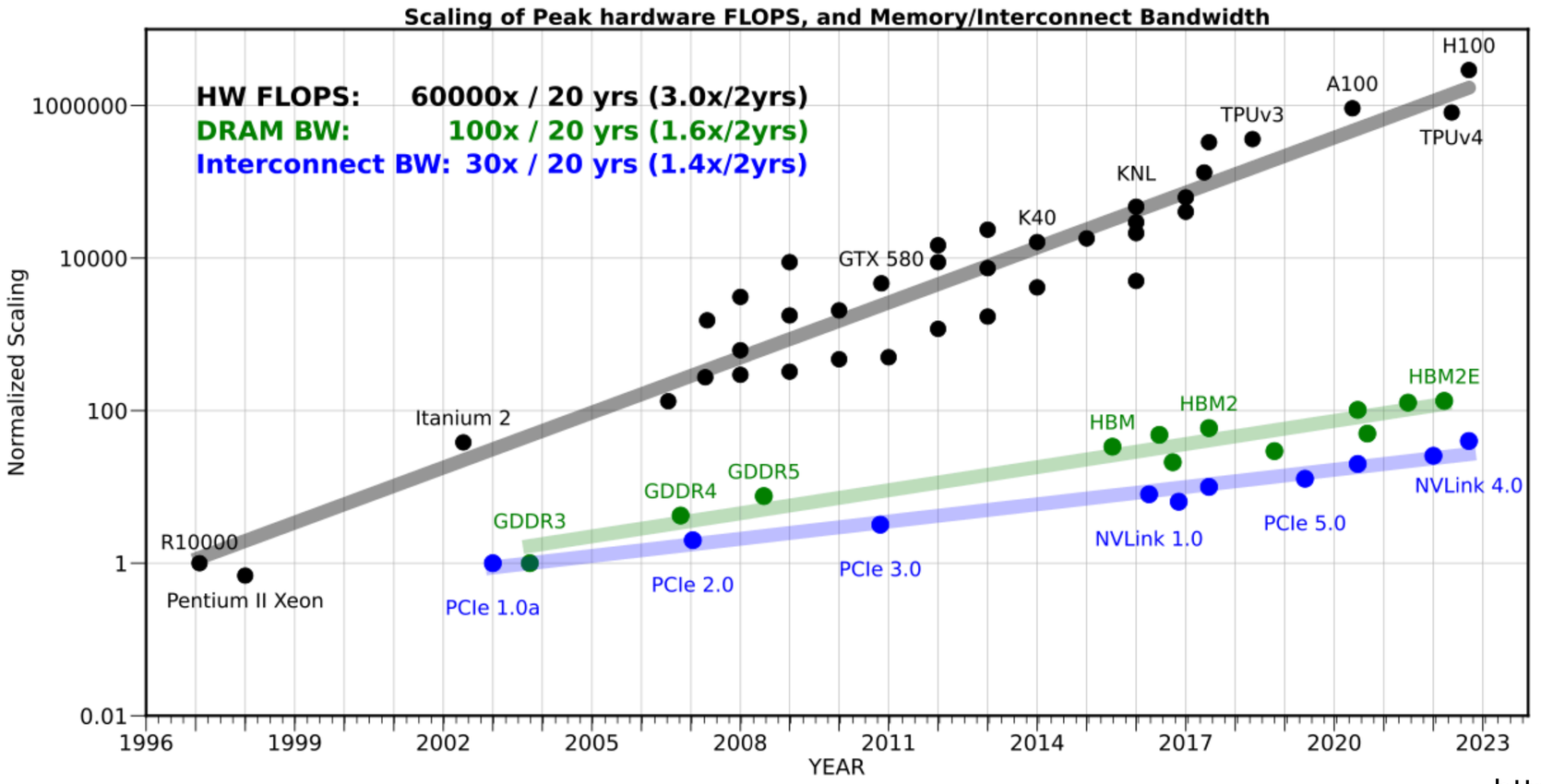


Memory hierarchy on Nvidia A100-40Gb

For more information, please look here: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

GPU memory

- The rate at which bandwidth increases is slower as the increase in computational power
- $\Longrightarrow$ Usually, this is the memory which is a bottleneck



https://arxiv.org/abs/2403.14123

GPU execution bottlenecks

Where the overhead comes from

- GPU underutilization
- Kernel launch latency
- Input shapes
- Host-device synchronization
- Unfused operations
- Device-to-Device or Host-to-Device data transfer

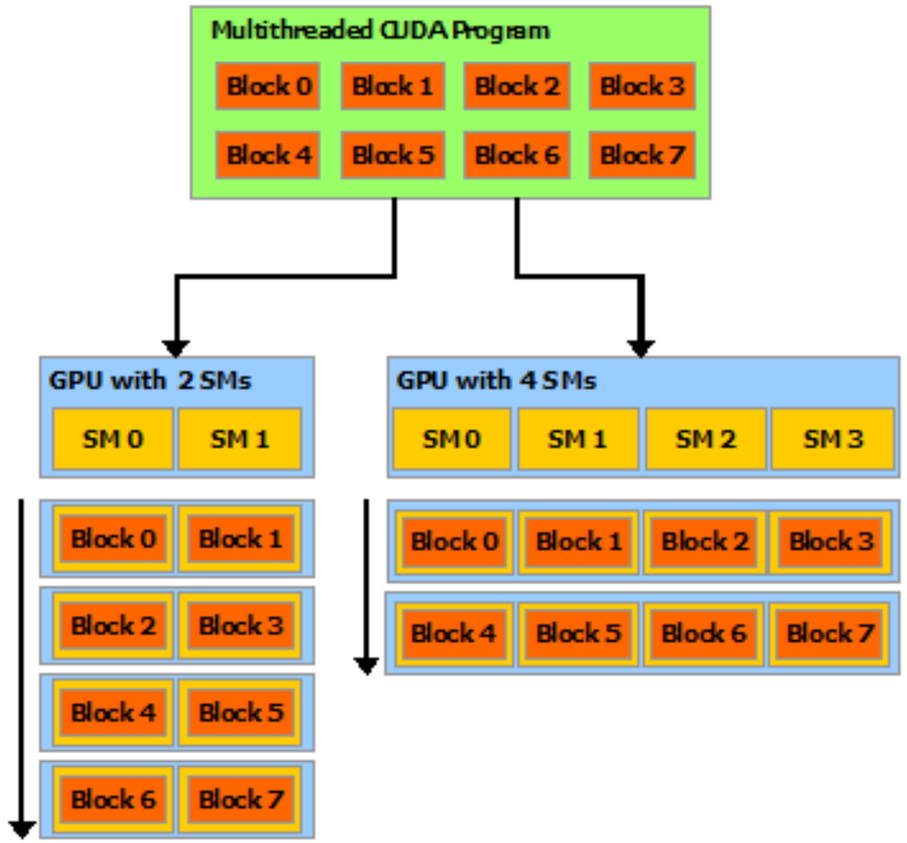
Where the overhead comes from

- Can be fixed from Python (or already handled by PyTorch)
- Can be partially fixed from Python
- Can't be fixed from Python. Can occur in custom code. Won't be covered

- GPU underutilization
- Kernel launch latency
- Input shapes
- Host-device synchronization
- Unfused operations
- Device-to-Device or Host-to-Device data transfer
- Global memory access, branch divergence, register occupancy, shared memory bank conflicts, atomic operations, cache misses, interconnect bandwidth

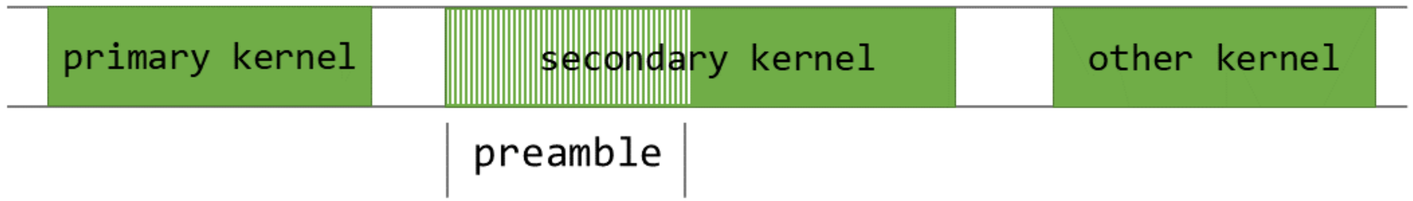
GPU underutilization

- Current batch size can underutilize SMs
- Too small thread blocks size or too large grid size can lead to smaller occupancy (portion of active SMs)



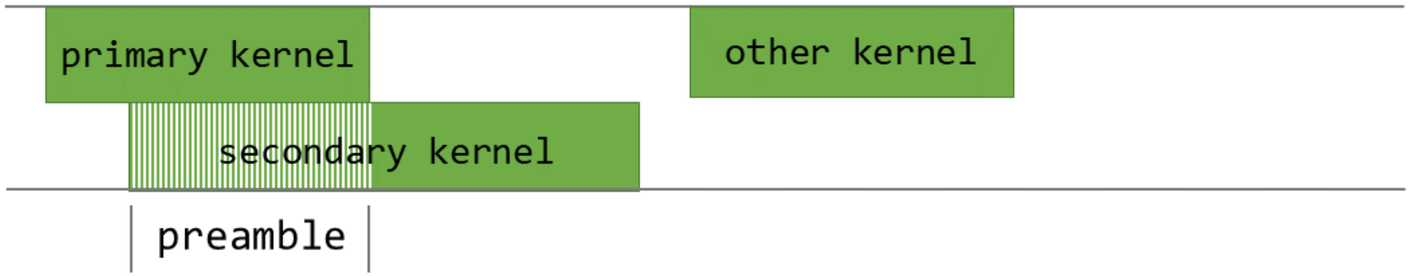
Kernel launch latency

- Kernels have some sort of *preamble* section which prepares the kernel to execute operations (zeroing buffers, loading constant values, etc.) - it accounts for launch latency
- If kernels are independent of each other, they are launched asynchronously in PyTorch to hide the latency
- From python, you can’t do anything about it (PyTorch does it for you), but be careful writing your own kernels!



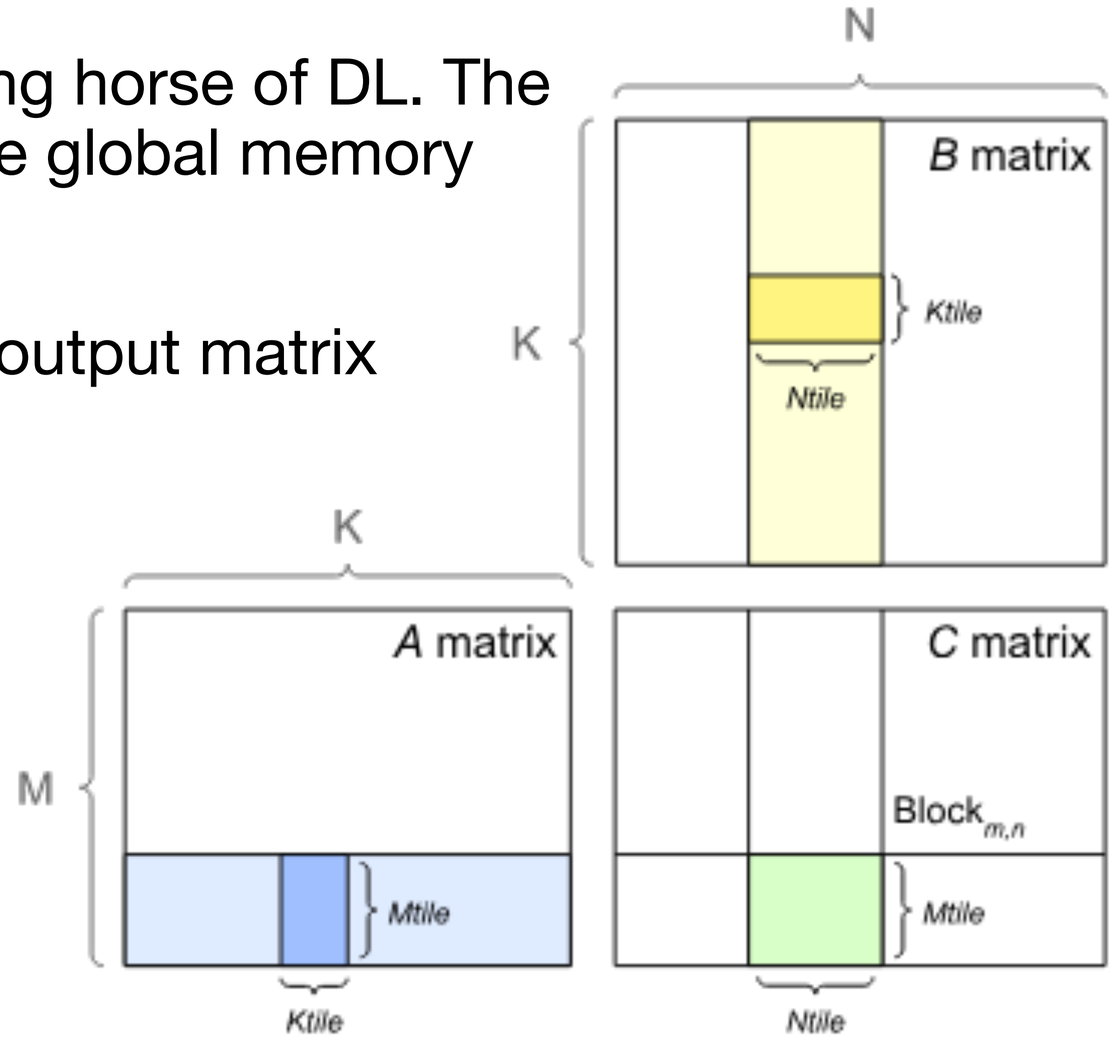
Kernel launch latency

- Kernels have some sort of *preamble* section which prepares the kernel to execute operations (zeroing buffers, loading constant values, etc.) - it accounts for launch latency
- If kernels are independent of each other, they are launched asynchronously in PyTorch to hide the latency
- From python, you can’t do anything about it (PyTorch does it for you), but be careful writing your own kernels!



Inconsistent input shapes

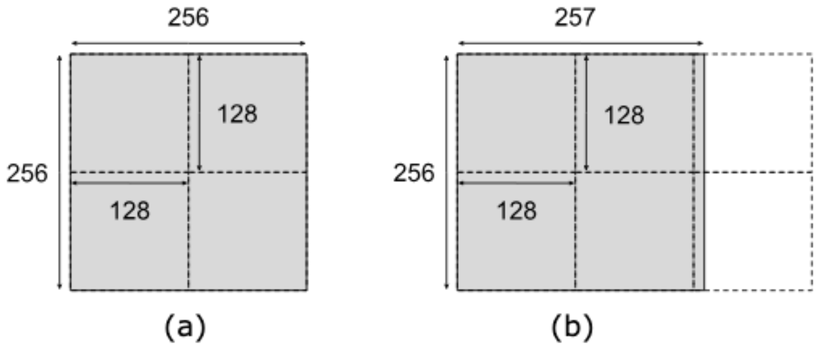
- Matrix multiplication (*matmul*) is the working horse of DL. The algorithms for CUDA utilize **tiling** to reduce global memory accesses
- Each thread block computes its **tile** of an output matrix



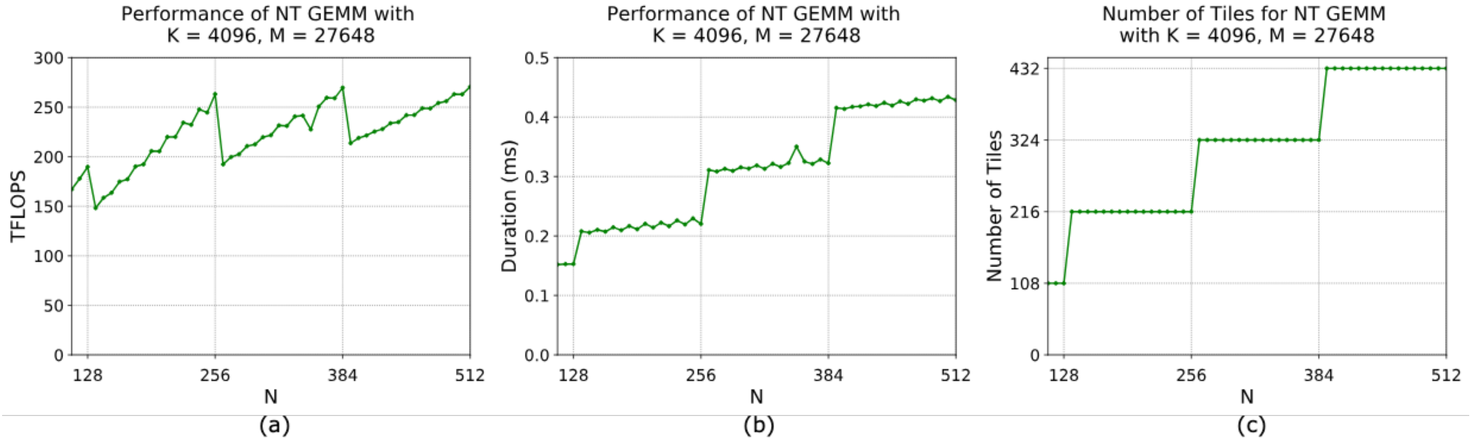
More on Tiling and matmul specs see https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Inconsistent input shapes

- **Tile quantization** — matrix dimensions are not divisible by tile size
- Some thread blocks will do very little work on the corner cases



(a) Best case (b) 2 thread blocks will waste most of the work

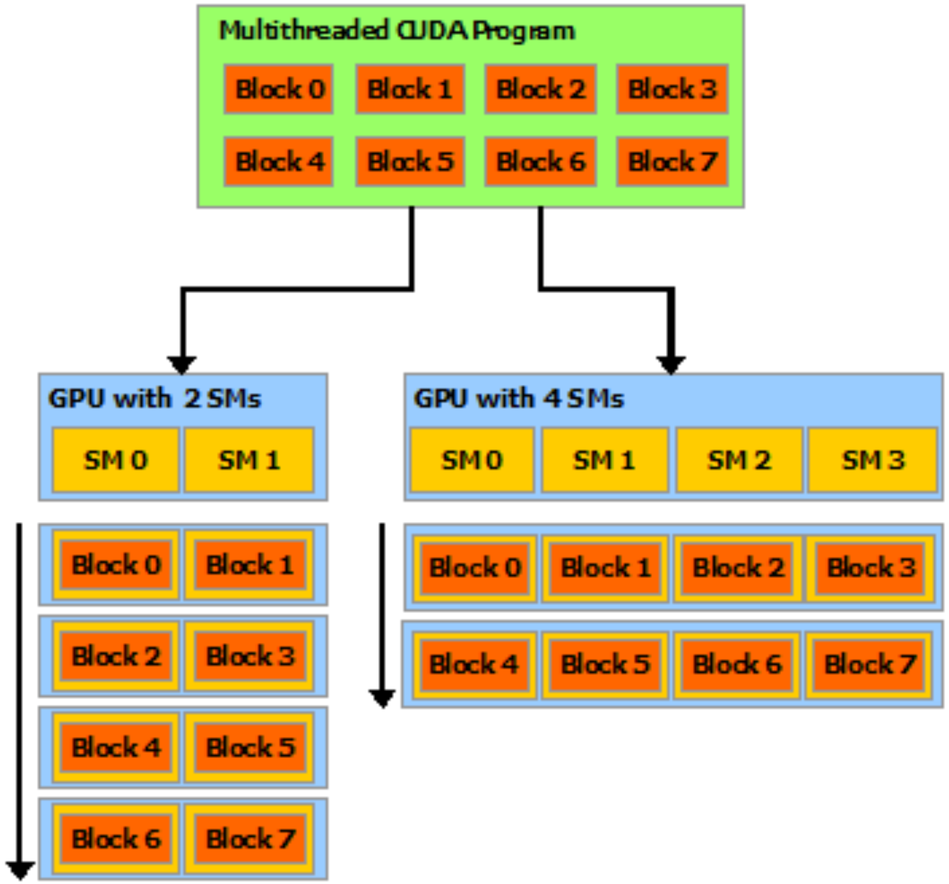


Tile quantization effect in terms of (a) achieved FLOPS throughput and (b) elapsed time, as well as (c) the number of tiles created. Measured with a function that uses 256x128 tiles over the MxN output matrix.

More on Tiling and matmul specs see https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Inconsistent input shapes

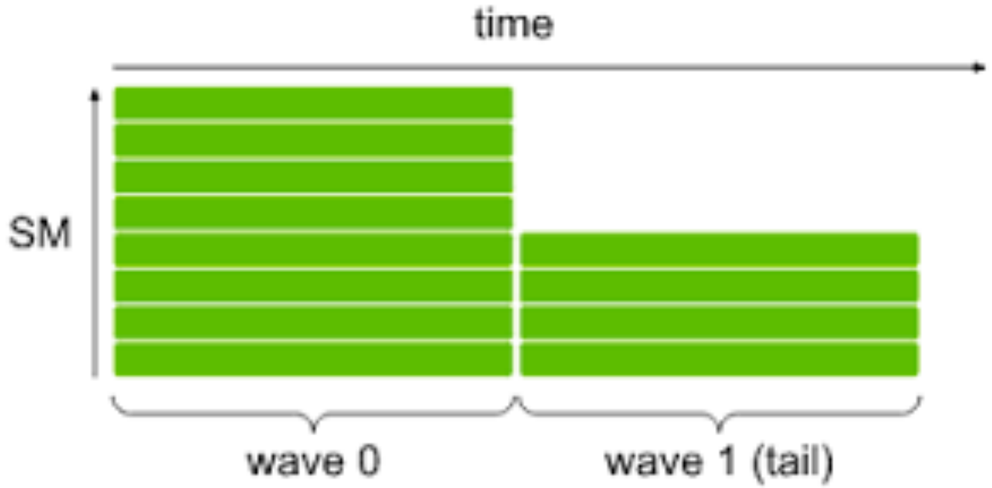
- **Wave quantization** — total number of tiles (thread blocks) is quantized to the number of SMs.
- Kernel will be executed in waves.
- Number of waves is $\left\lceil \frac{\text{Tiles}}{\text{Num. of SMs}} \right\rceil$
- The last wave can underutilize the GPU
- Best utilization occurs with the number of tiles is the multiple os SMs



More on Tiling and matmul specs see https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Inconsistent input shapes

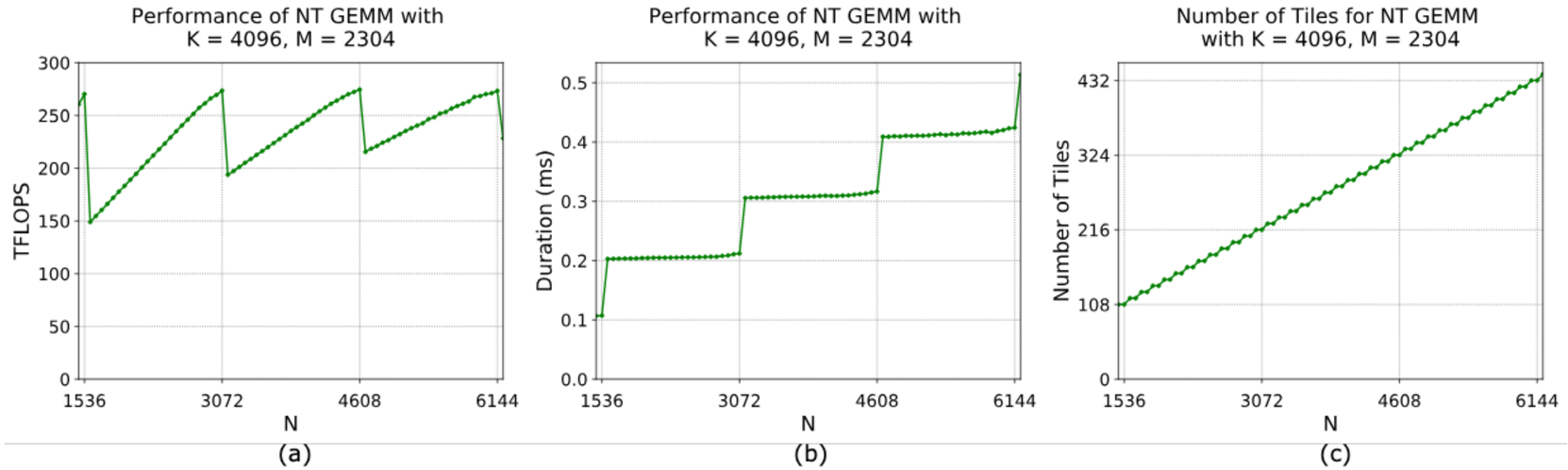
- **Wave quantization** — total number of tiles (thread blocks) is quantized to the number of SMs.
- Kernel will be executed in waves.
- Number of waves is $\left\lceil \frac{\text{Tiles}}{\text{Num. of SMs}} \right\rceil$
- The last wave can underutilize the GPU
- Best utilization occurs with the number of tiles is the multiple os SMs



More on Tiling and matmul specs see https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Inconsistent input shapes

- **Wave quantization** — total number of tiles (thread blocks) is quantized to the number of SMs.
- The last wave can underutilize the GPU

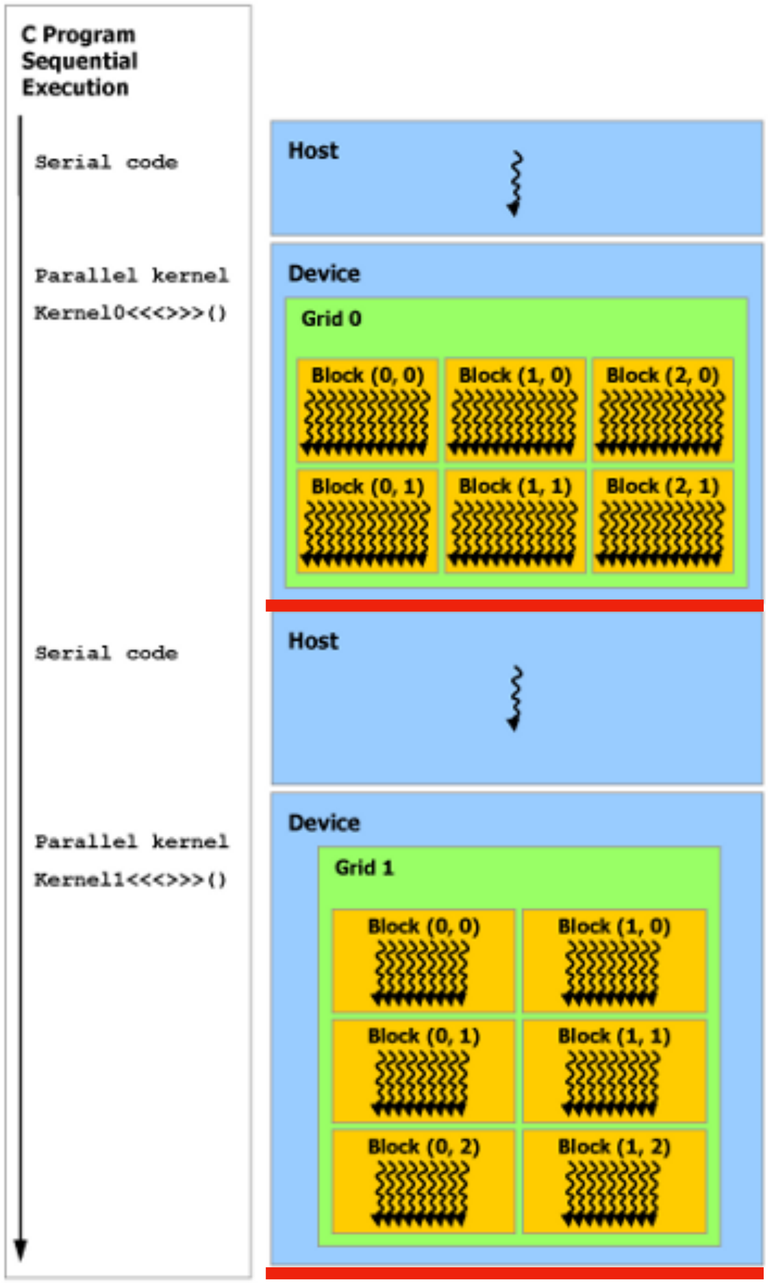


The effects of wave quantization in terms of (a) achieved FLOPS throughput and (b) elapsed time, as well as (c) the number of tiles created. Measured with a function that uses 256x128 tiles over the MxN output matrix.

More on Tiling and matmul specs see https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

Host-device synchronization

- GPU code is executed asynchronously with Python's CPU code
- Any unnecessary CPU-GPU synchronization slows down the GPU execution

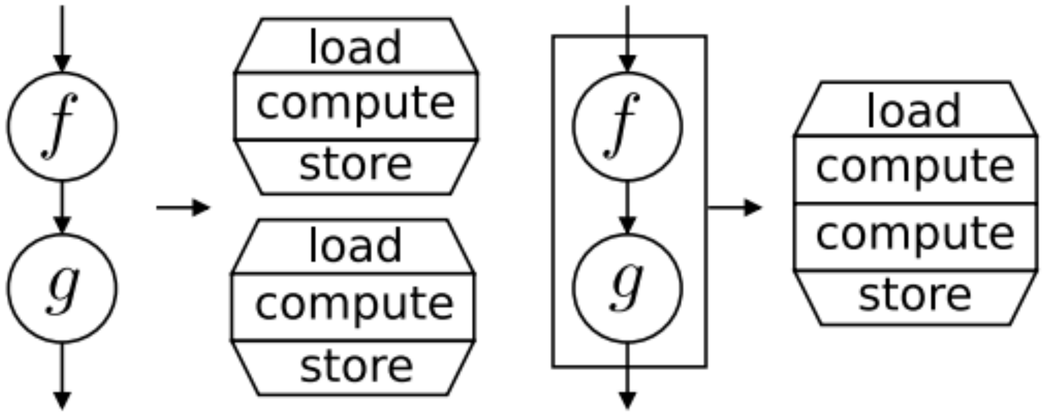


Host-device synchronization

- GPU code is executed asynchronously with Python's CPU code
- Any unnecessary CPU-GPU synchronization slows down the GPU execution
  - E.g. `loss.item()` in PyTorch triggers synchronization— use `loss.detach()`

Unfused operations & Kernel fusion

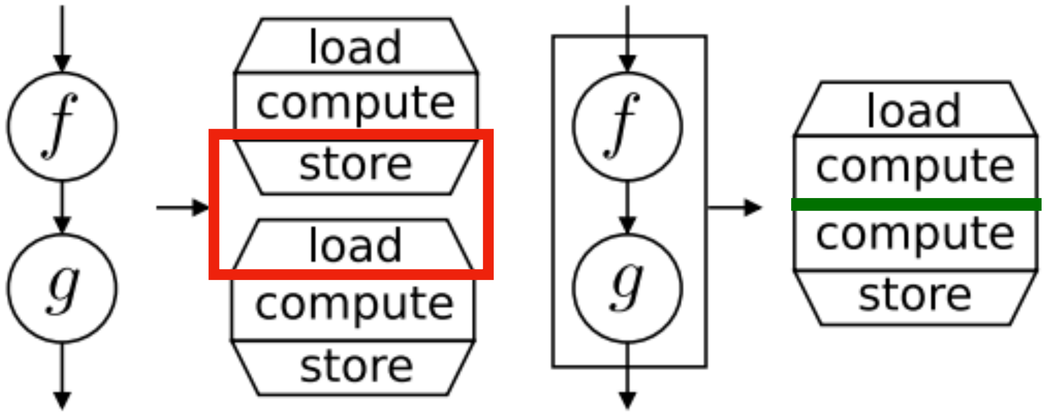
- Typical kernel workflow — load from global memory → execute operation → store in global memory
- For sequential kernel execution, there will be redundant IO-operations — we can place all operations in a single kernel - **fuse** them:



https://arxiv.org/abs/1305.1183

Unfused operations & Kernel fusion

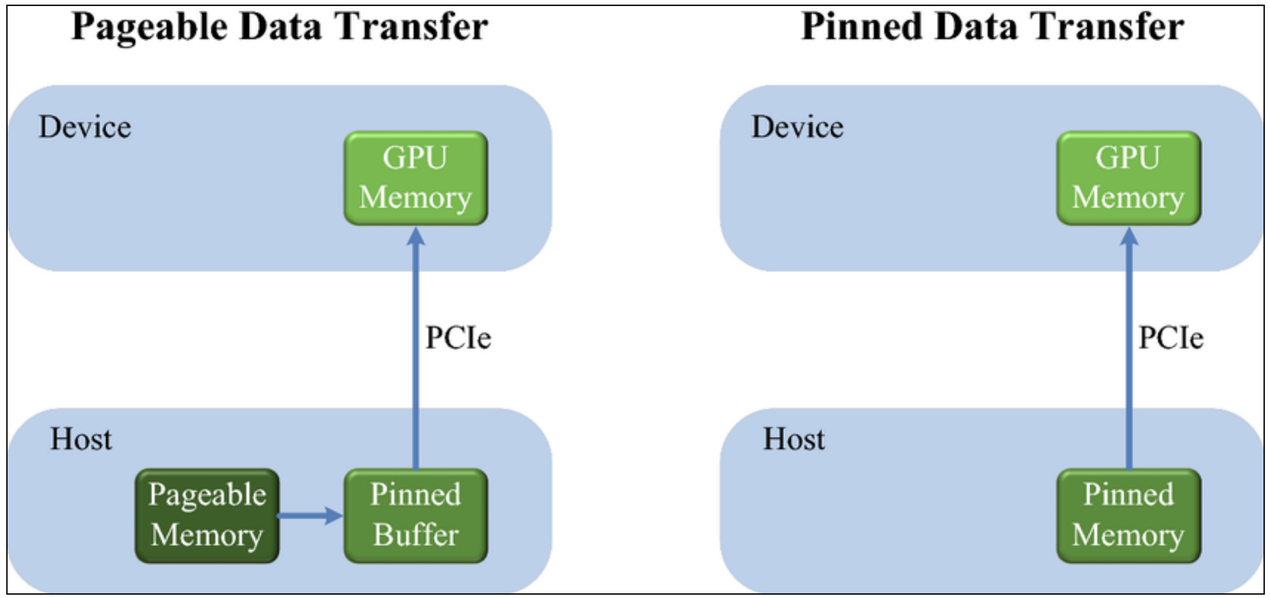
- Typical kernel workflow — load from global memory → execute operation → store in global memory
- For sequential kernel execution, there will be redundant IO-operations — we can place all operations in a single kernel - **fuse** them:



https://arxiv.org/abs/1305.1183

Data transfer

- Host-to-Device (CPU-to-GPU) operations require an object to be located in RAM in a pinned buffer
- Operation system pages RAM and can offload some pages - accessing offloaded pages requires load them to a pinned buffer
- We can pin the memory to speed up the transfer



Identify & deal with bottlenecks

… and speed up the execution

Things you can do:

- Use proper batch size
- Reduce dimensions inconsistencies (e.g. for hidden dims)
- Reduce the amount of CPU-GPU synchronizations
- Use pageable memory (+ cuda Streams - link)
- Compile model (be careful in distributed scenarios)
- Moving additional operations on GPU might help (e.g. normalizations, data transformations & augmentations)
- **Code profiling** — identify slow code samples (e.g. inefficient forward implementation, slow data transfer)

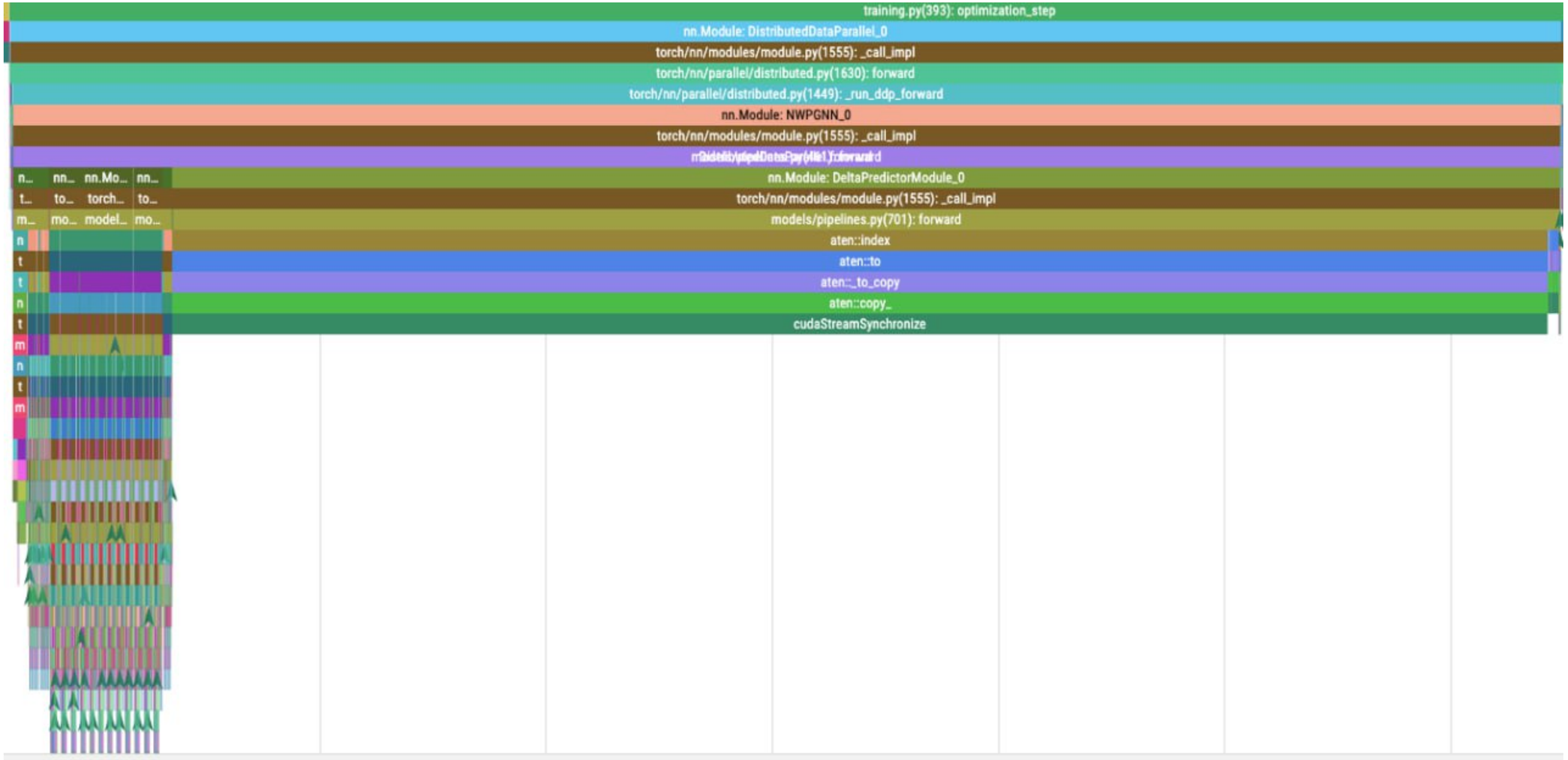
Code profiling

- Identify slow patterns in the code
- For CPU profiler - `py-spy`
- PyTorch has its own profiler — torch.profiler.profiler
  - Very handy with Tensorboard

Code profiling

- Identify slow patterns in the code
- For CPU profiler - `py-spy`
- PyTorch has its own profiler — torch.profiler.profiler
  - Very handy with Tensorboard
- Also you can try `torch.utils.bottleneck`
- For distributed job analysis, you can use Holistic Trace Analysis — https://docs.pytorch.org/tutorials/beginner/hta_intro_tutorial.html

Trace example



`torch.compile`

- Just-in-Time model compilation
- Inspects PyTorch code & modules, makes graph representation
- Optimizes code by making optimized kernels (e.g. fused operators) & removing python overhead
- Different shapes of the input tensors can break fused operations, PyTorch will need to recompile a model
- Use with caution! (see more on a seminar)

Thanks for attention!