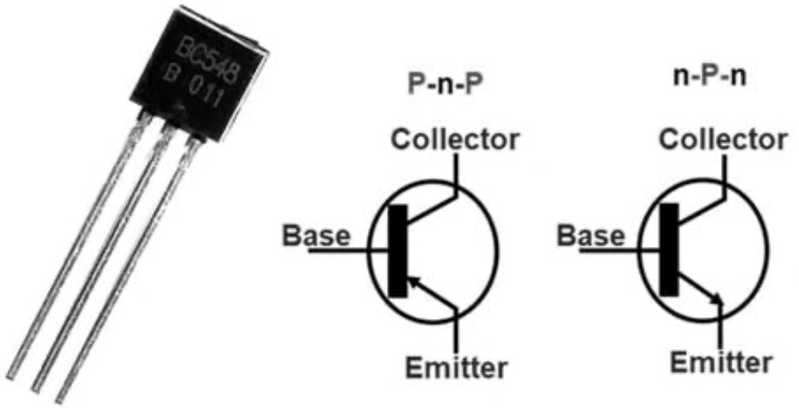
Essentials of GPU

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

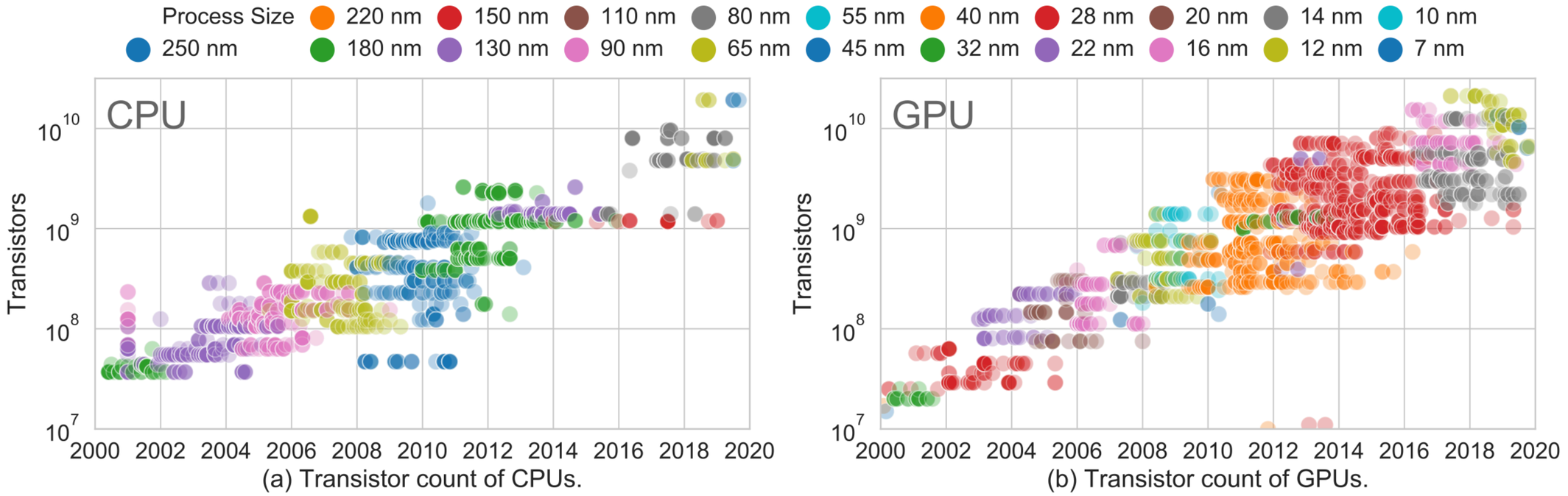


Fig. 1. Moore’s Law is still valid for both CPUs and GPUs.

https://arxiv.org/abs/1911.11313

Dennard Scaling law

- Energy consumption per transistor reduces with decreasing process size **(a)**

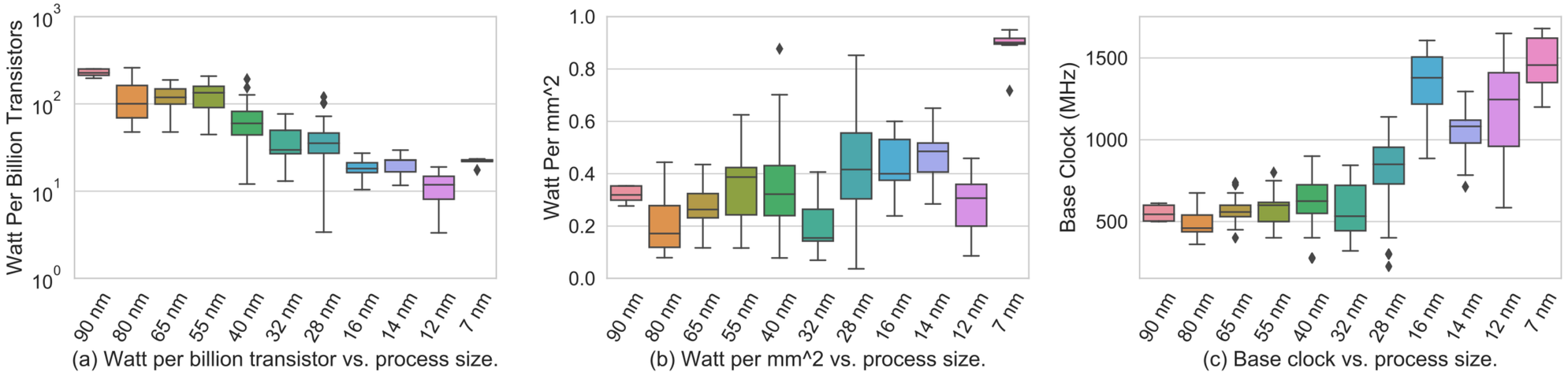


Fig. 4. Process Size vs. (a) Energy consumption per billion transistor. (b) Energy consumption per area. (c) Base clock speed.

* 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 constant in various process sizes **(b)**

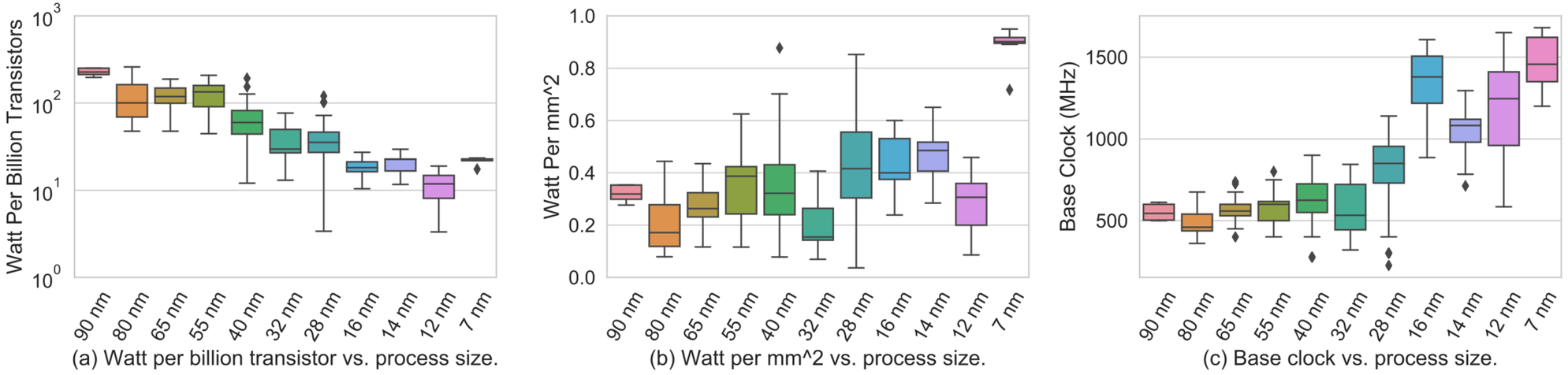


Fig. 4. Process Size vs. (a) Energy consumption per billion transistor. (b) Energy consumption per area. (c) Base clock speed.

* 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)**

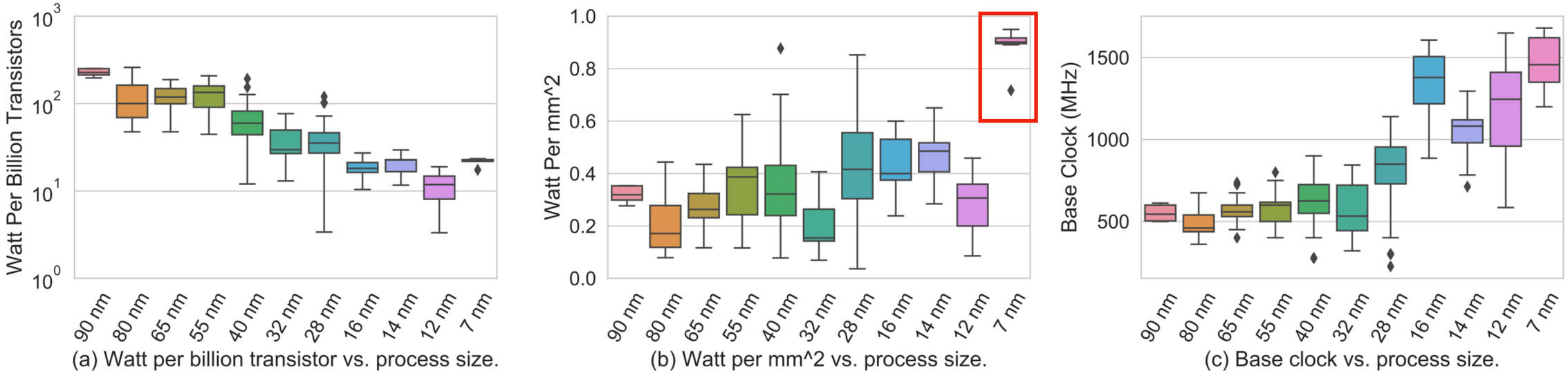


Fig. 4. Process Size vs. (a) Energy consumption per billion transistor. (b) Energy consumption per area. (c) Base clock speed.

* 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)**

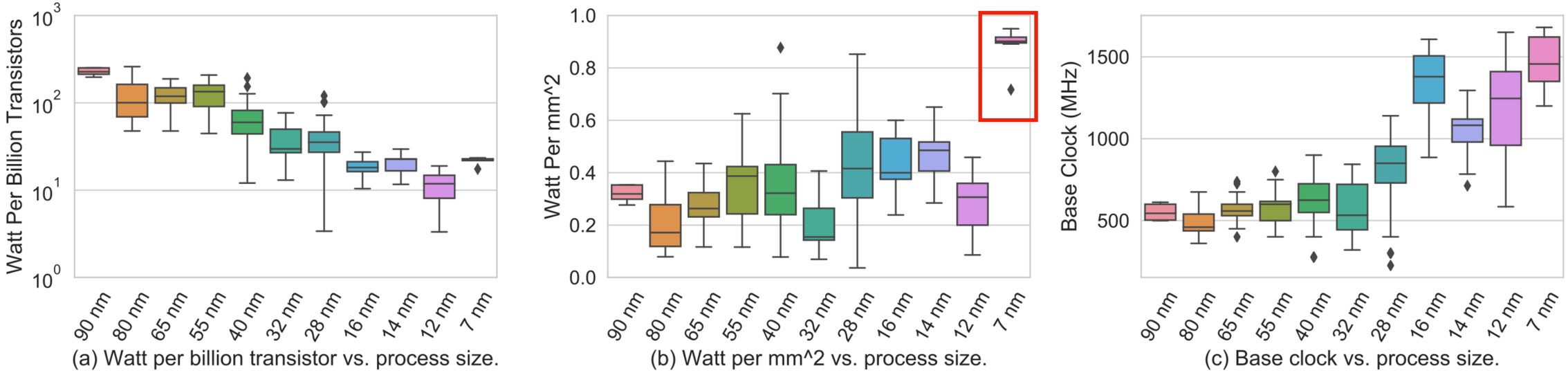


Fig. 4. Process Size vs. (a) Energy consumption per billion transistor. (b) Energy consumption per area. (c) Base clock speed.

* 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

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

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)

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

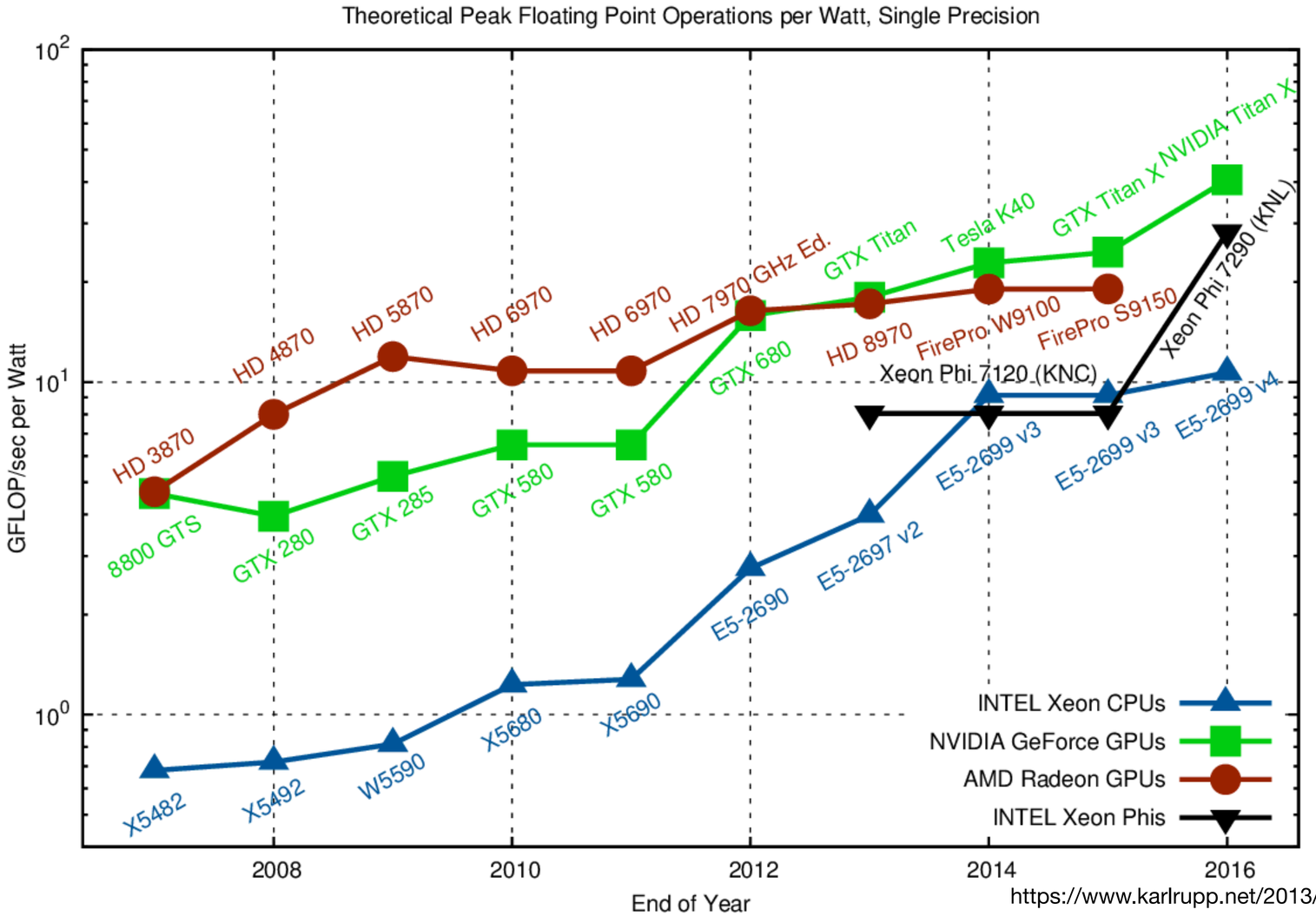
- At 4-5nm scale transistors become affected by random electron fluctuations
- Chip's circuits are manufactured with photolithography — you can't create circuits smaller half the frequency of the light wave (~10 nm)

- **Takeaway:** we are limited in scaling hardware performance, **but we can optimize the parts we need for parallel computation**

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

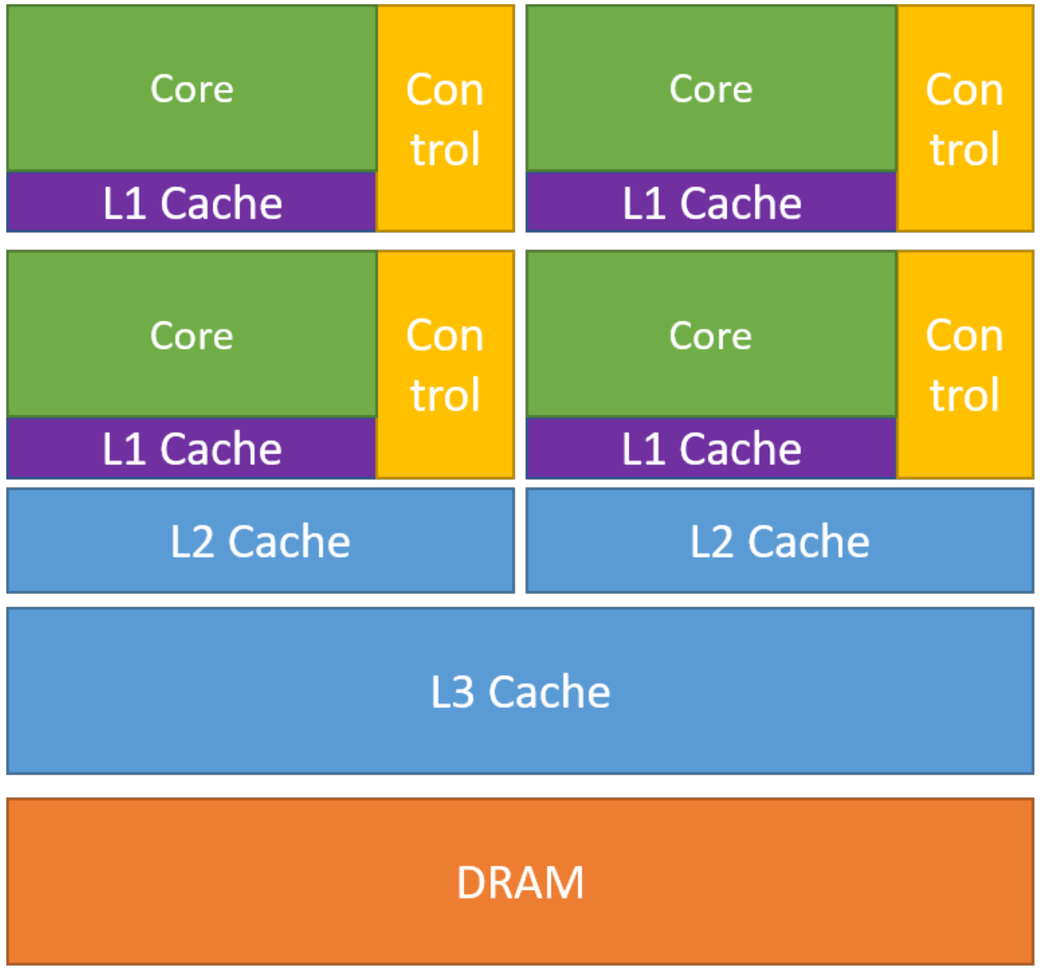


https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

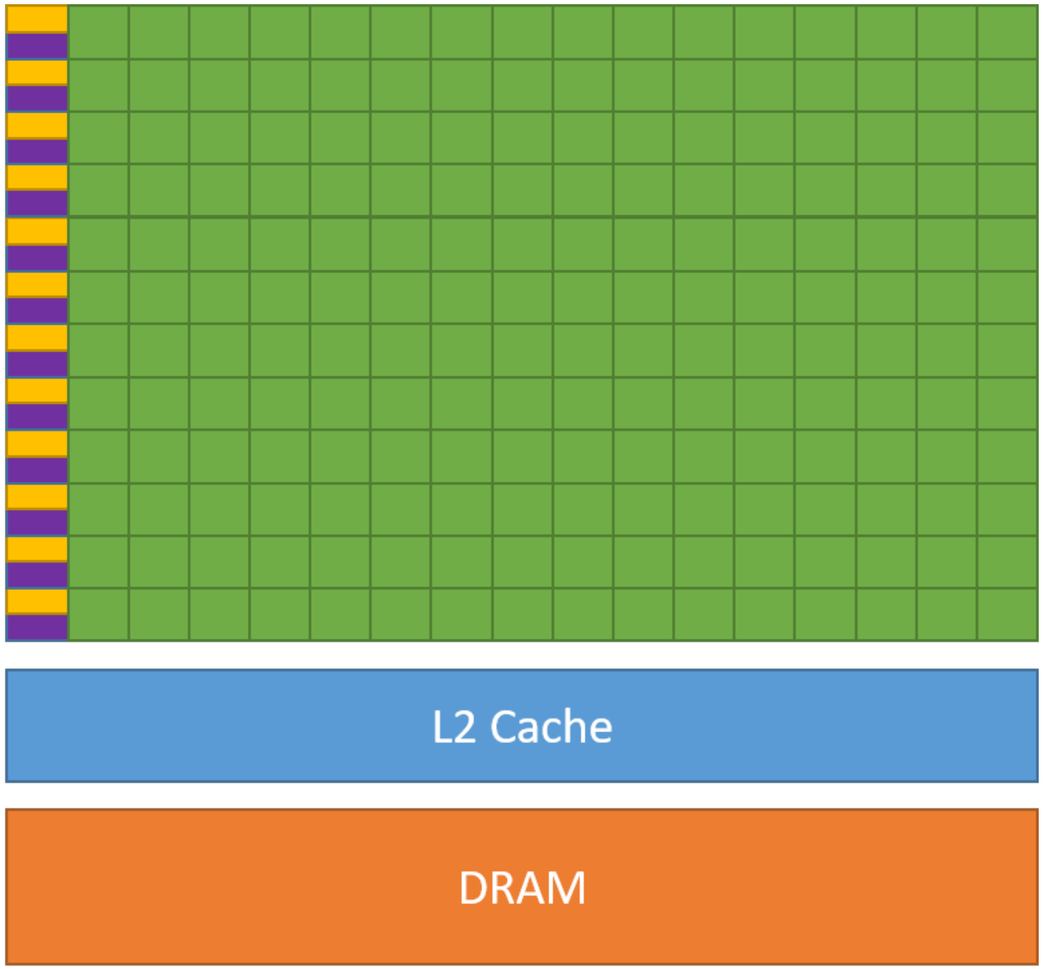
GPU programming paradigm

*Nvidia GPUs

CPU vs. GPU architecture

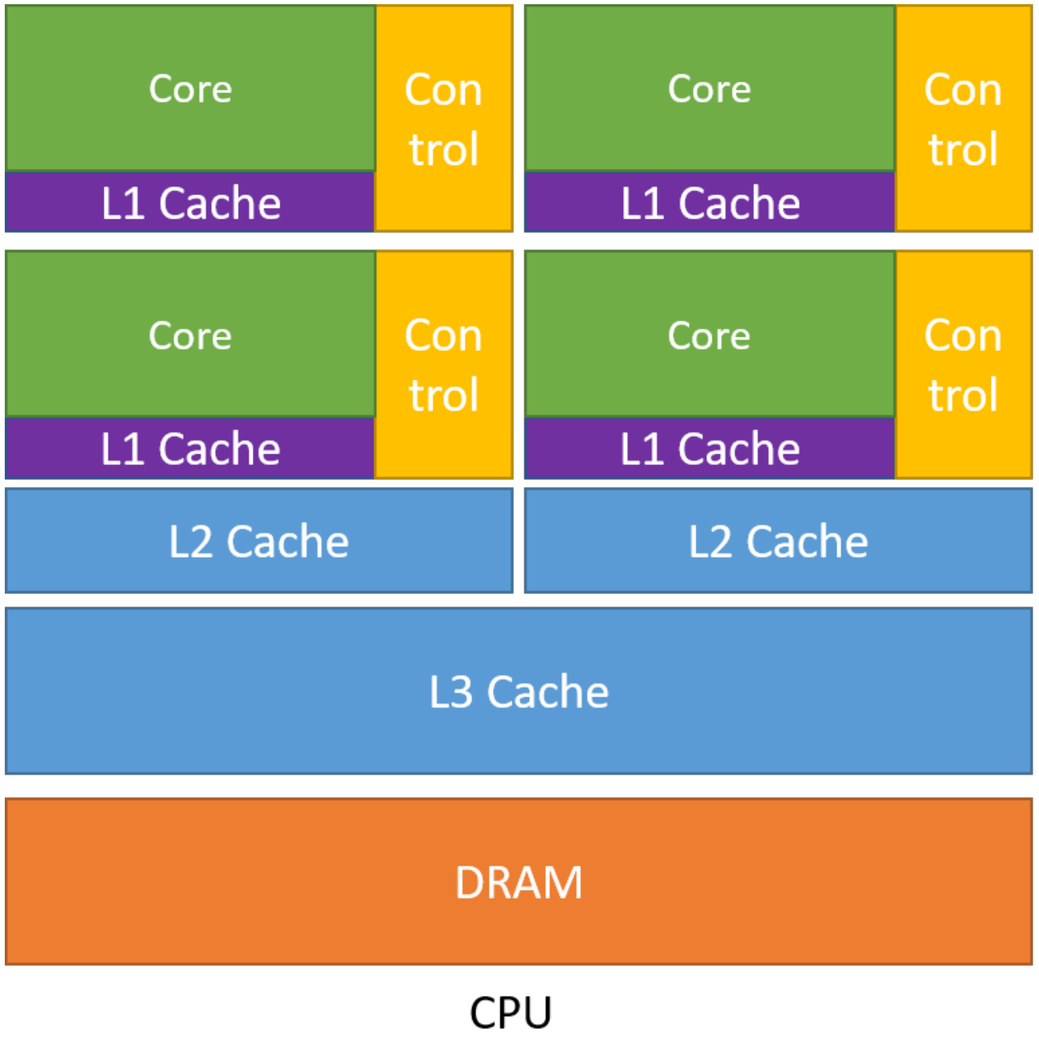


CPU



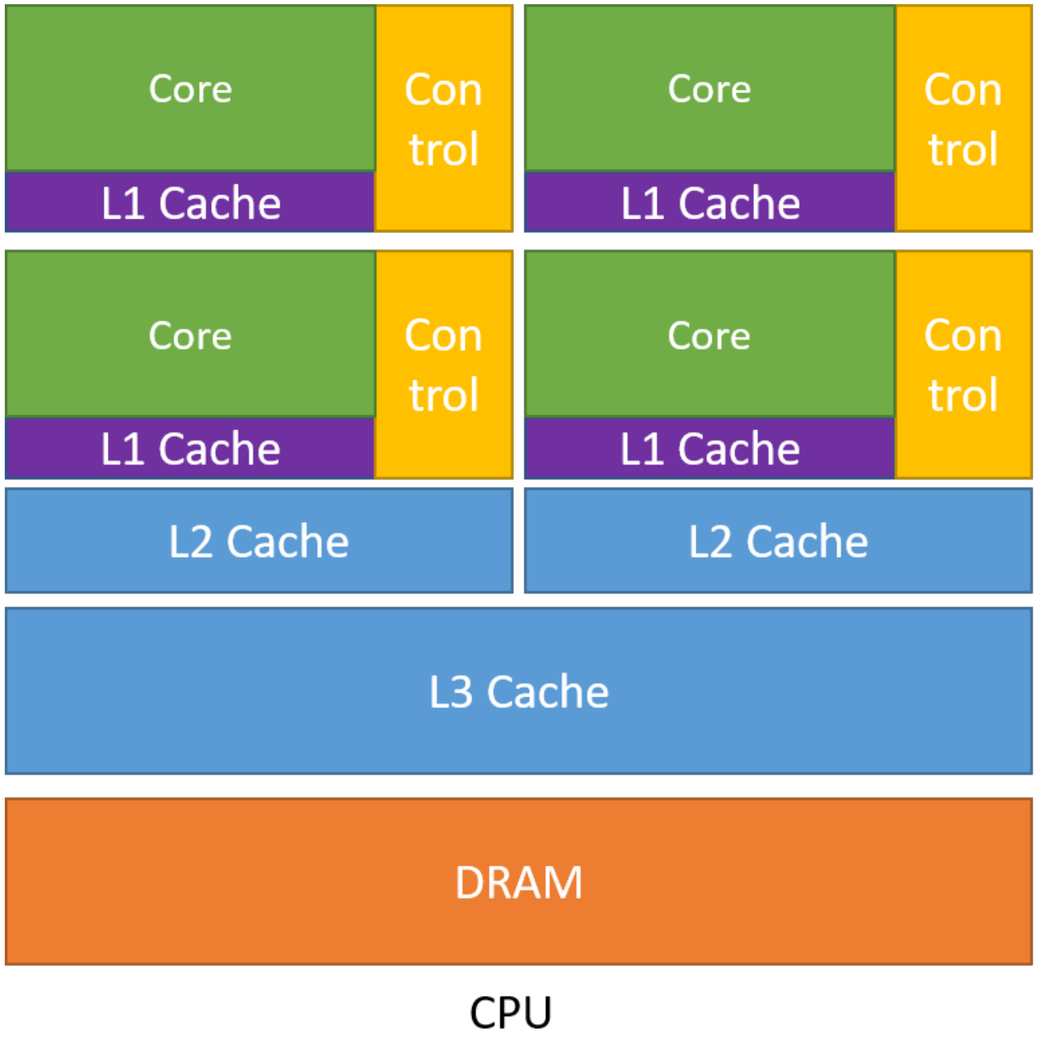
GPU

CPU architecture



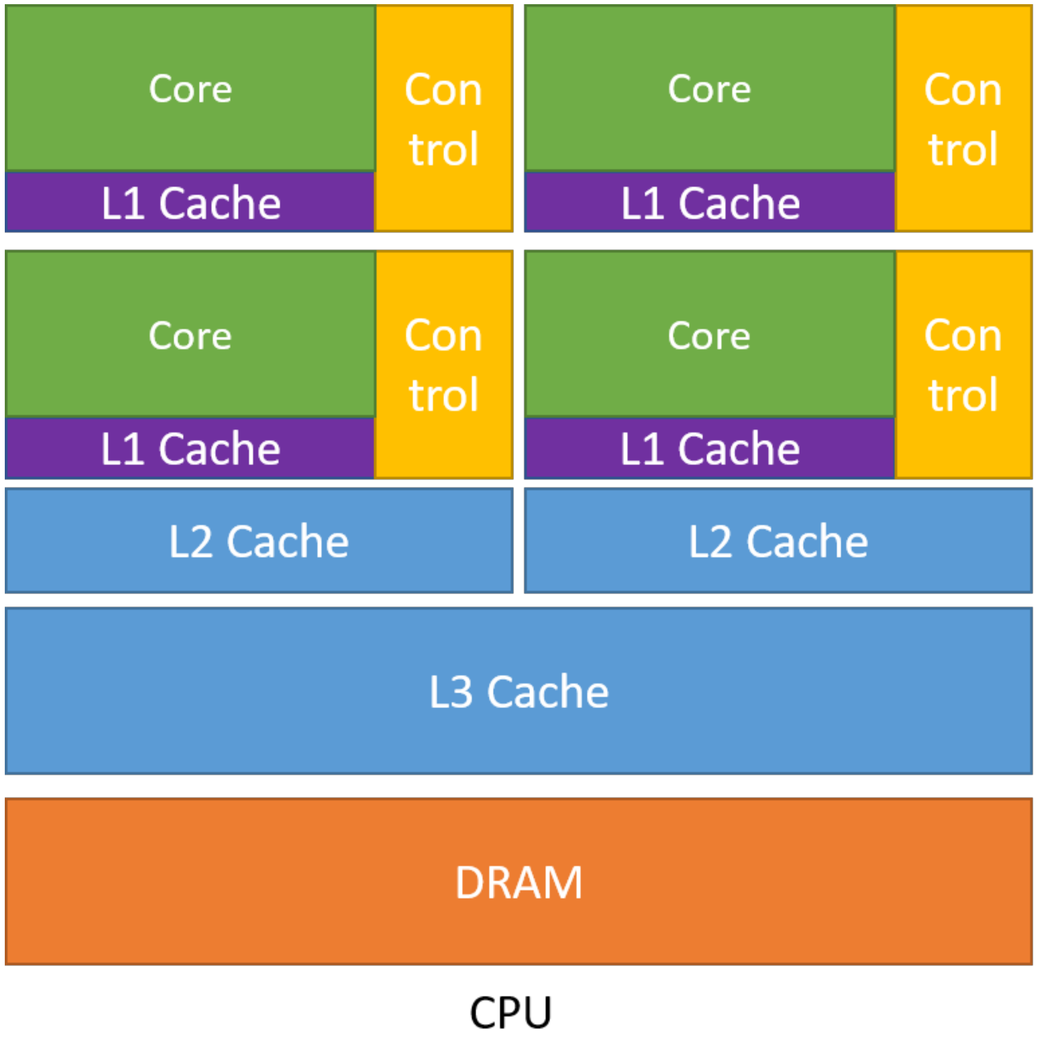
- Complex control logic

CPU architecture



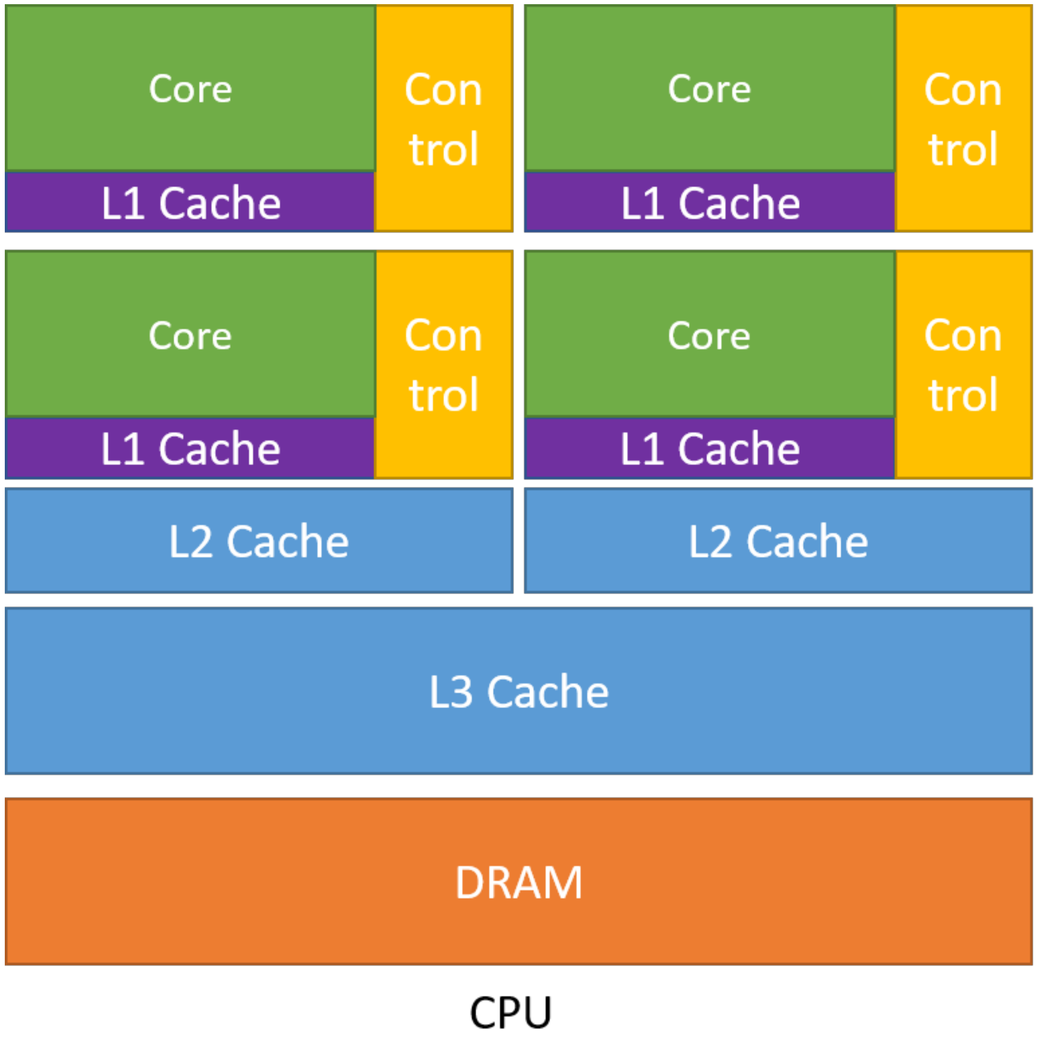
- Complex control logic
- More transistors to data caching & flow control (Larger caches)

CPU architecture



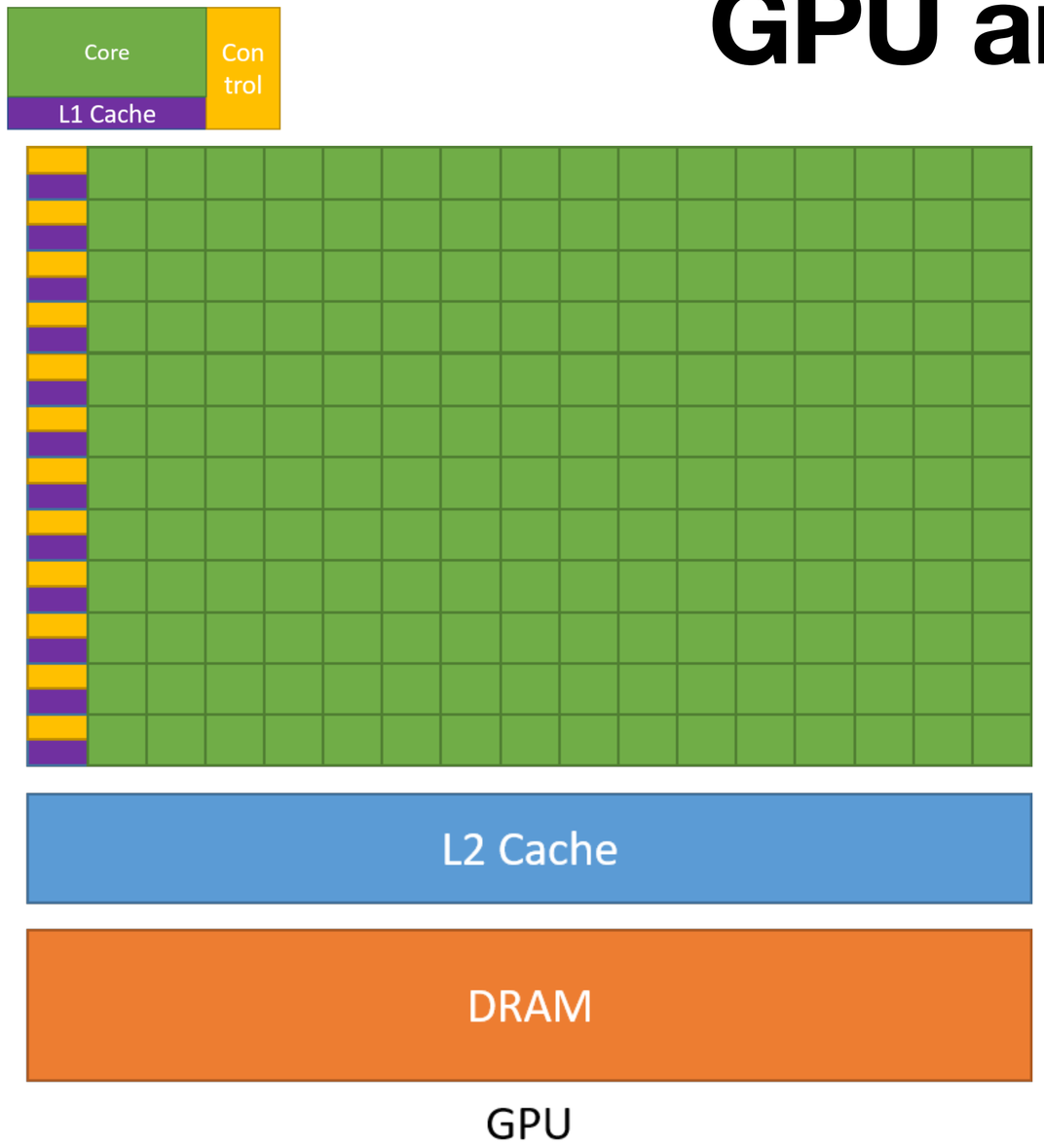
- Complex control logic
- More transistors to data caching & flow control (Larger caches)
- Better at executing **sequential** operations
  - Fewer execution units
  - Higher clock speeds

CPU architecture



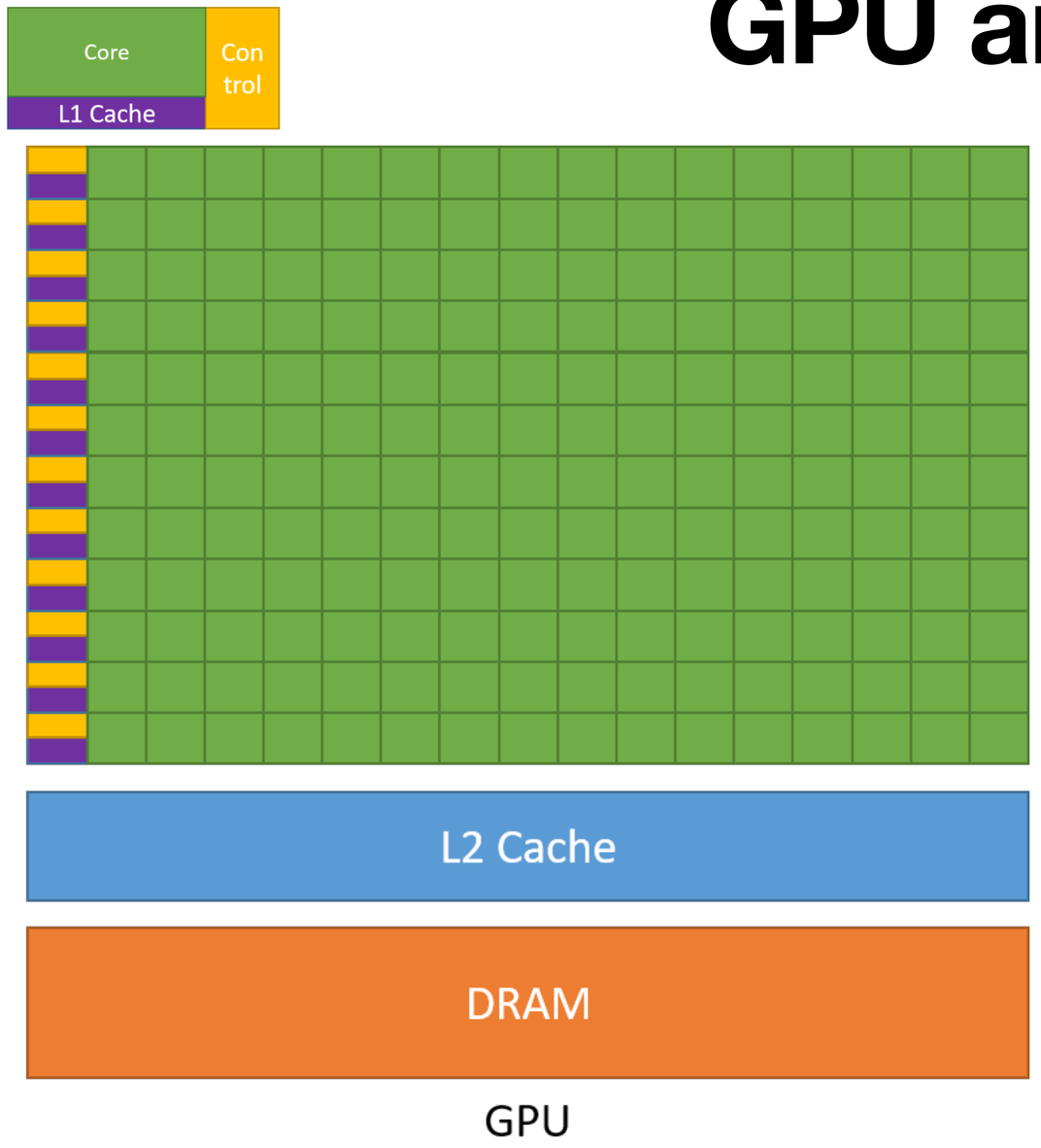
- Complex control logic
- More transistors to data caching & flow control (Larger caches)
- Better at executing **sequential** operations
  - Fewer execution units
  - Higher clock speeds
- Fast single-thread performance
  - Newer CPUs have more parallelism

GPU architecture



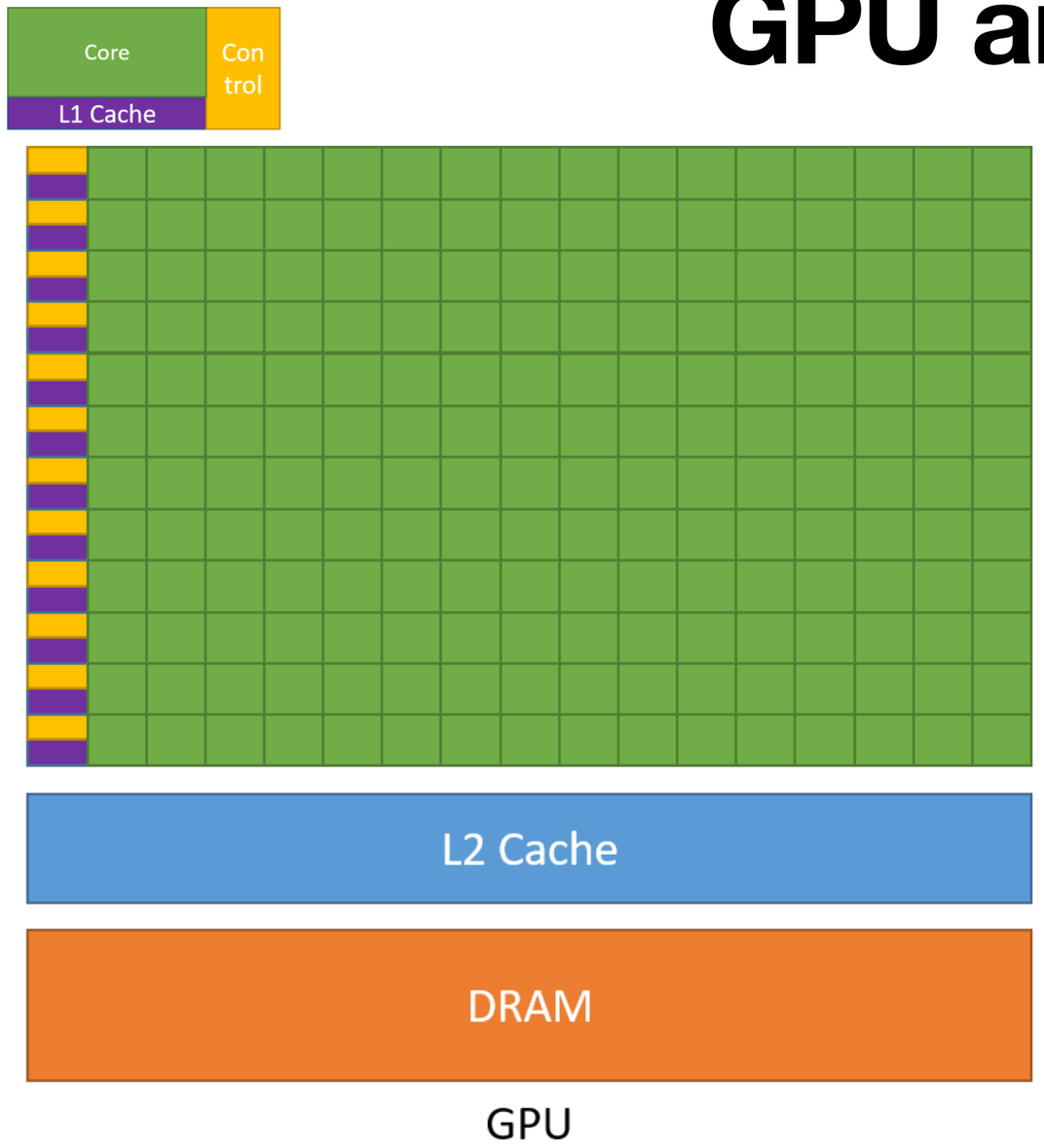
- Shared control blocks across multiple cores

GPU architecture



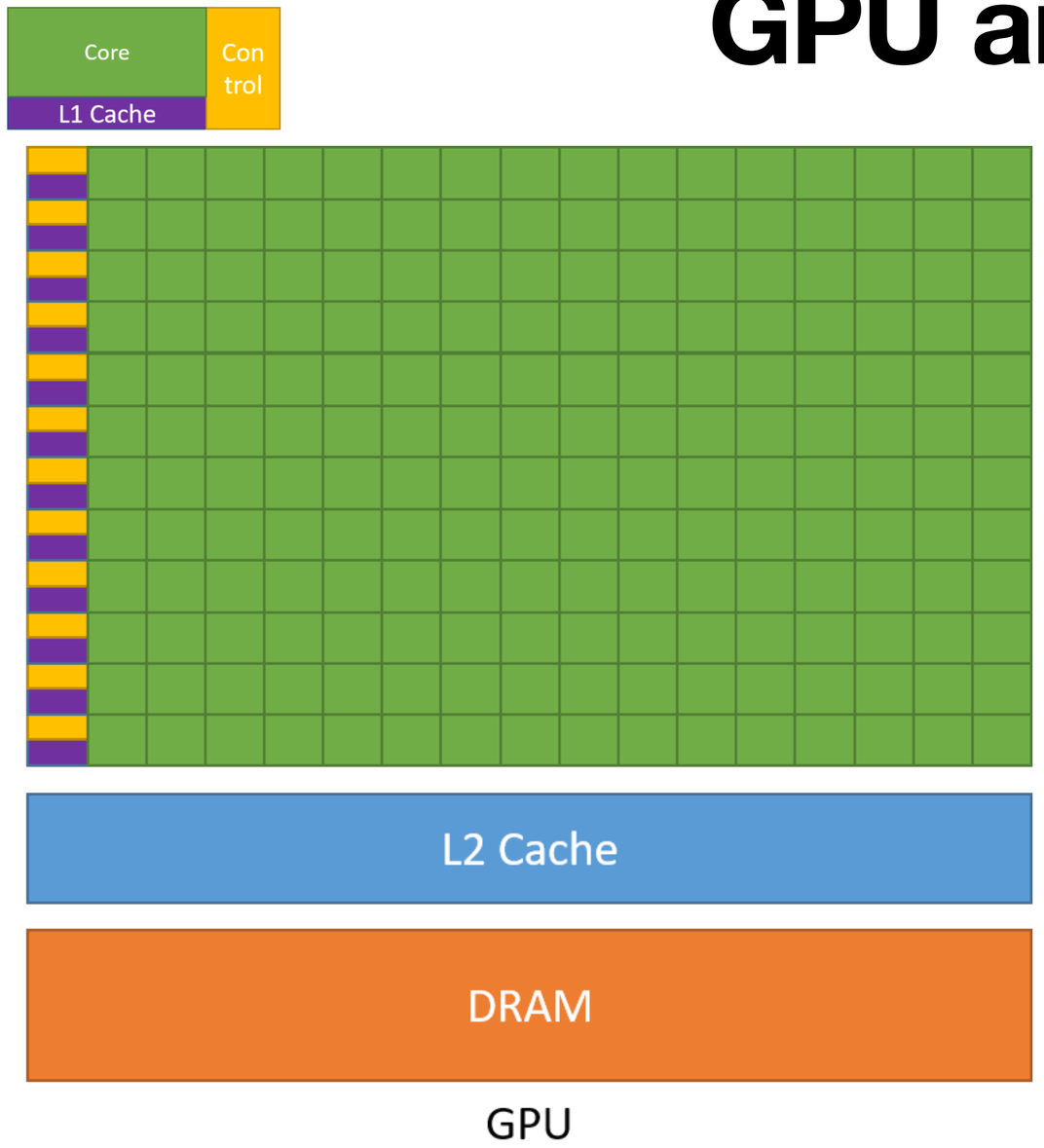
- Shared control blocks across multiple cores
- More transistors to data processing

GPU architecture



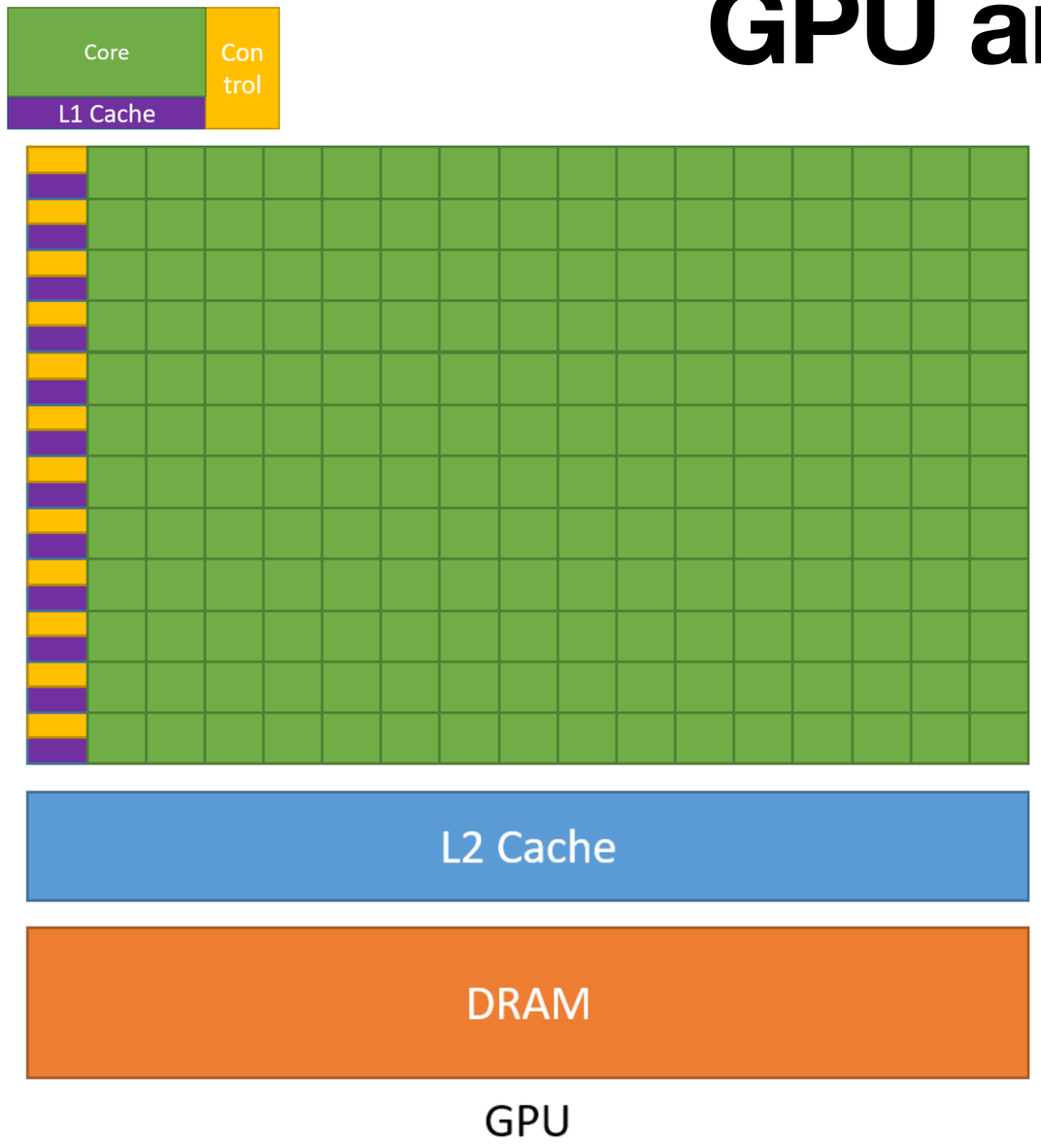
- Shared control blocks across multiple cores
- More transistors to data processing
- Built for parallel operations

GPU architecture



- Shared control blocks across multiple cores
- More transistors to data processing
- Built for parallel operations
- Slow single-thread performance (due to latency), **amortized by parallel computation**

GPU architecture



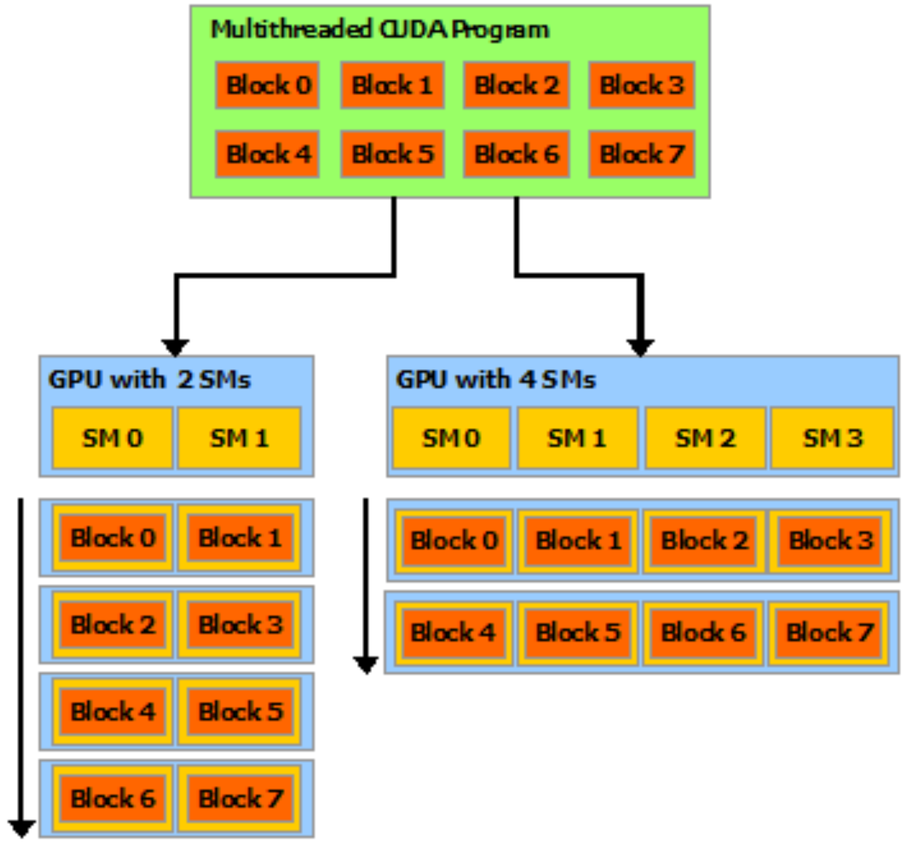
- Shared control blocks across multiple cores
- More transistors to data processing
- Built for parallel operations
- Slow single-thread performance (due to latency), **amortized by parallel computation**
- **Higher instruction throughput compared to CPU with similar price and power consumption**

CUDA programming model

*Nvidia GPUs

CUDA (Compute Unified Device Architecture)

- Parallel computing platform on NVIDIA-built hardware
- API for writing software on NVIDIA GPUs
- The same program can run on any device regardless of their compute capability

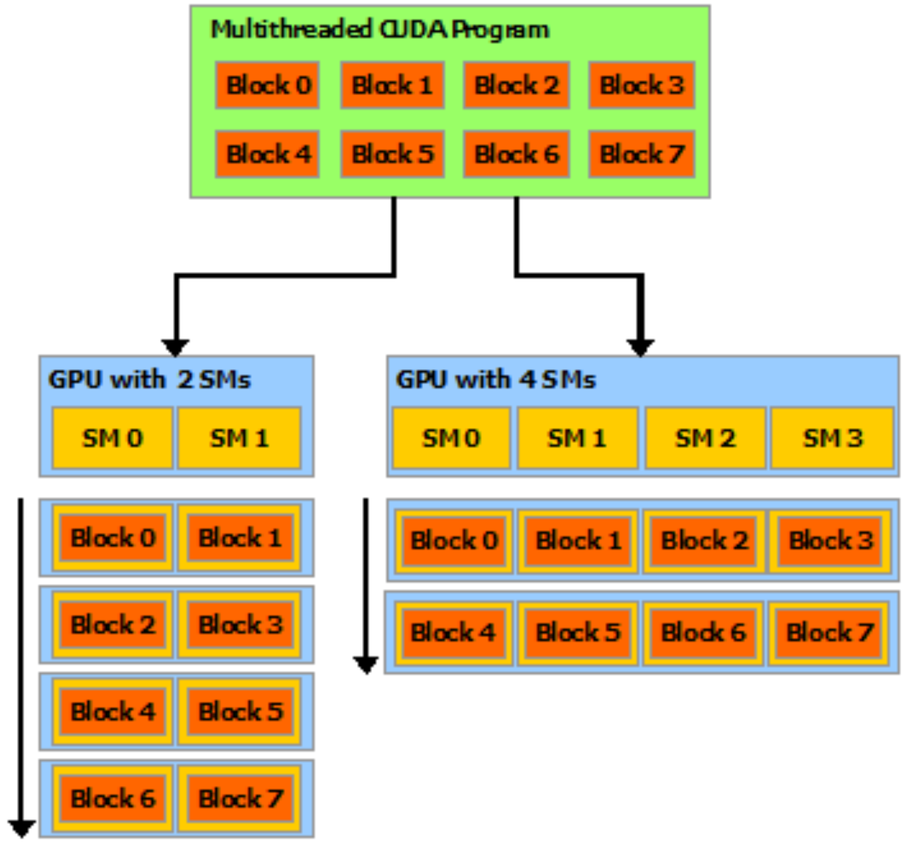


Different GPUs execute the same program

https://developer.nvidia.com/cuda-gpus

CUDA (Compute Unified Device Architecture)

- Parallel computing platform on NVIDIA-built hardware
- API for writing software on NVIDIA GPUs
- The same program can run on any device regardless of their compute capability *(kinda)*
  - We have backward compatibility
  - We don’t have forward compatibility (can’t use new instructions on the older GPUs)



Different GPUs execute the same program

https://developer.nvidia.com/cuda-gpus

CUDA (Compute Unified Device Architecture)

- Parallel computing platform on NVIDIA-built hardware
- API for writing software on NVIDIA GPUs
- The same program can run on any device regardless of their compute capability *(kinda)*
  - We have backward compatibility
  - We don’t have forward compatibility (can’t use new instructions on the older GPUs)

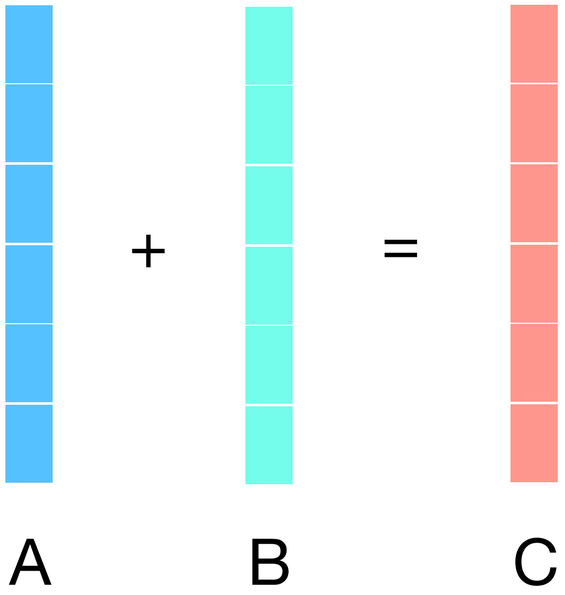
Major Revision Number	NVIDIA GPU Architecture
9	NVIDIA Hopper GPU Architecture
8	NVIDIA Ampere GPU Architecture
7	NVIDIA Volta GPU Architecture
6	NVIDIA Pascal GPU Architecture
5	NVIDIA Maxwell GPU Architecture
3	NVIDIA Kepler GPU Architecture

Compute capability versions

https://developer.nvidia.com/cuda-gpus

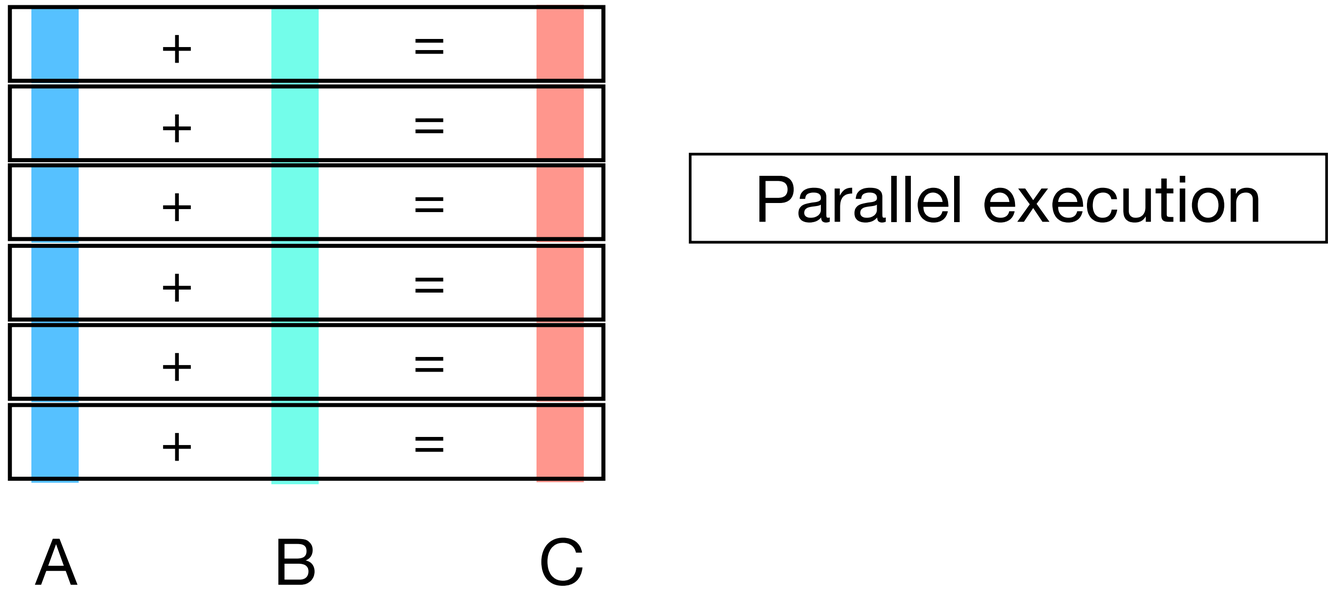
An example of GPU computation. Kernels

- GPU execute **kernels** - a functions executed with set of instructions 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:



An example of GPU computation. Kernels

- GPU execute **kernels** - a functions executed with set of instructions 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:



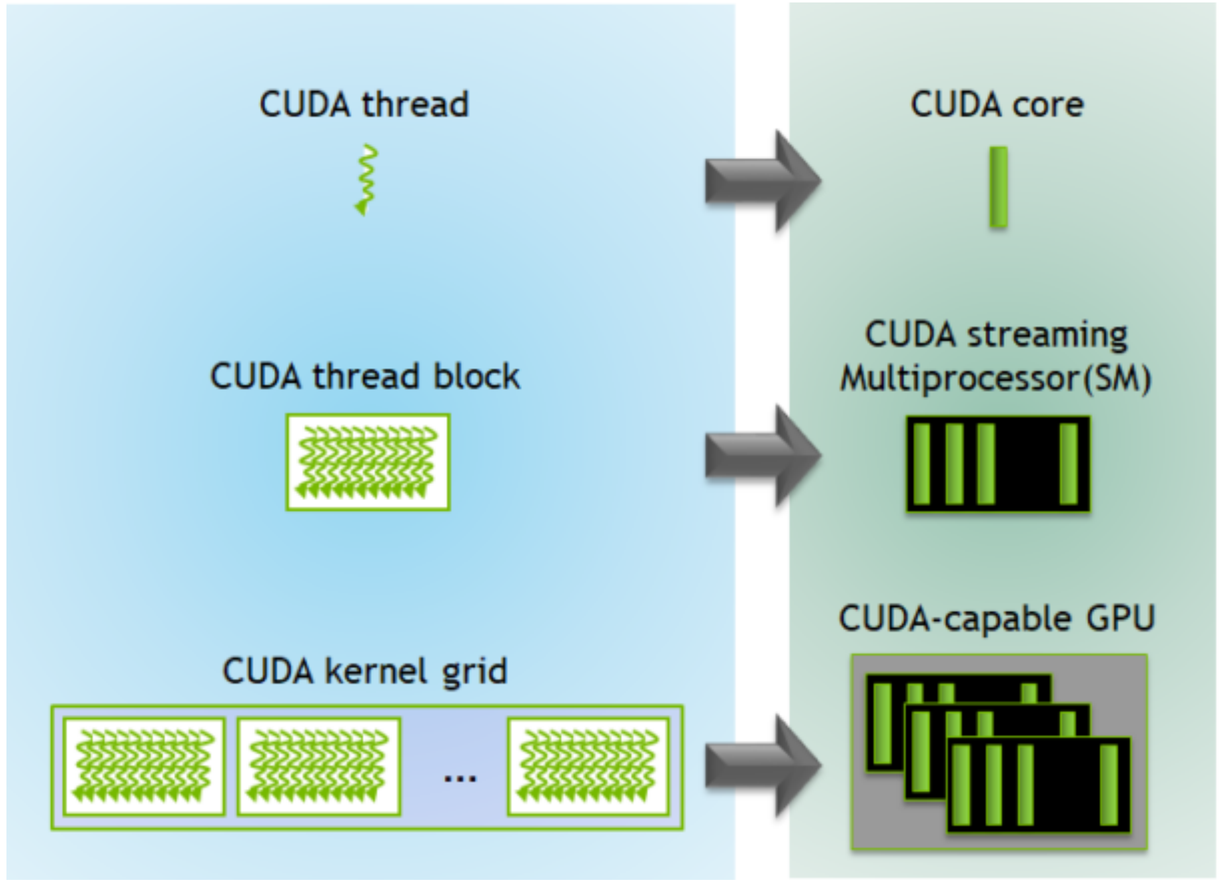
Streaming Multiprocessor (SM)

- Basic operational unit on a GPU
- Contains computation and data loading logic:
  - Arithmetical units
  - Registers (thousands)
  - Warp schedulers (for issuing instructions)
  - Cache units (L1, constant cache, texture cache)
  - Shared memory
  - On H100 and newer - blocks for fast memory transfer (TMA)
- Some resources are shared during the execution (memory, cache)

http://thebeardsage.com/cuda-streaming-multiprocessors/

Physical level for CUDA model

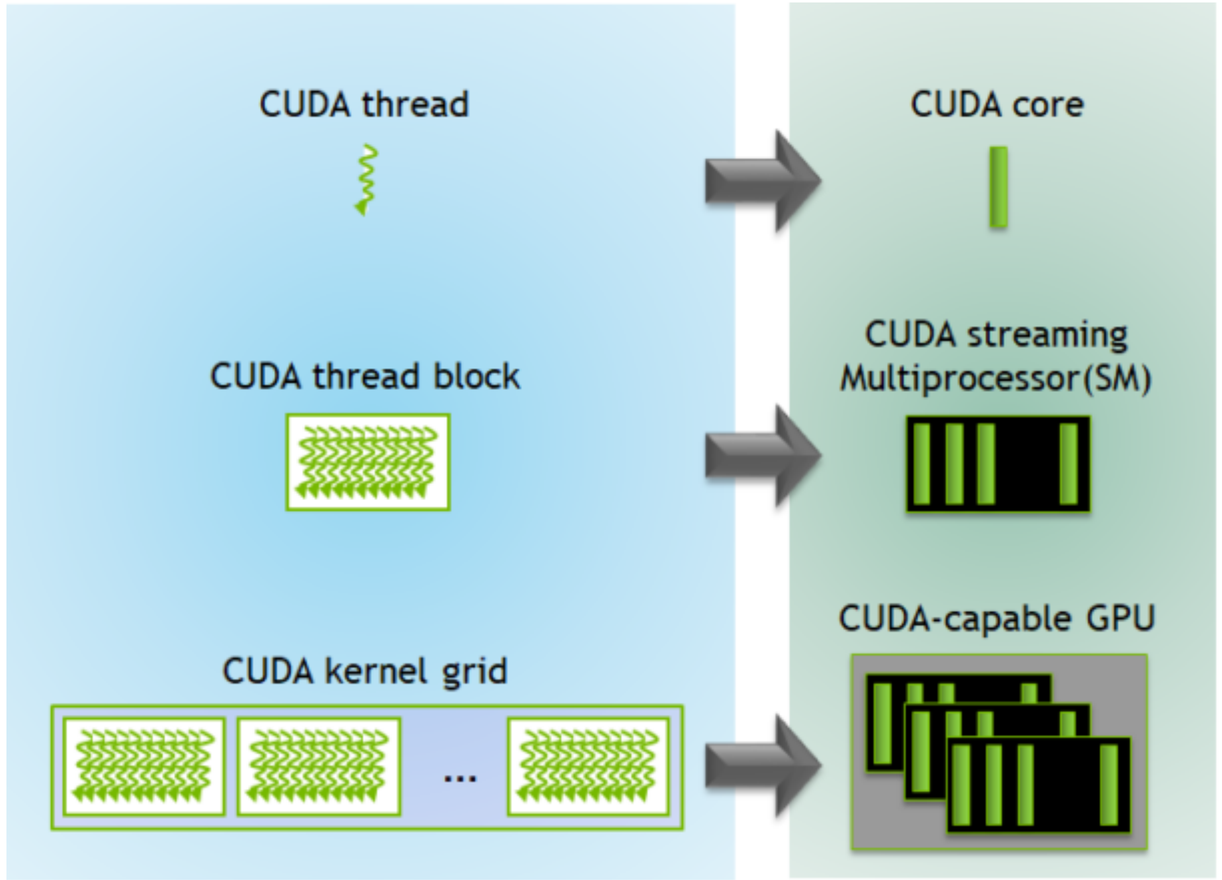
- Program launches **threads** - number of threads on SM is limited
  - Current GPUs support up to 1024 threads



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

Physical level for CUDA model

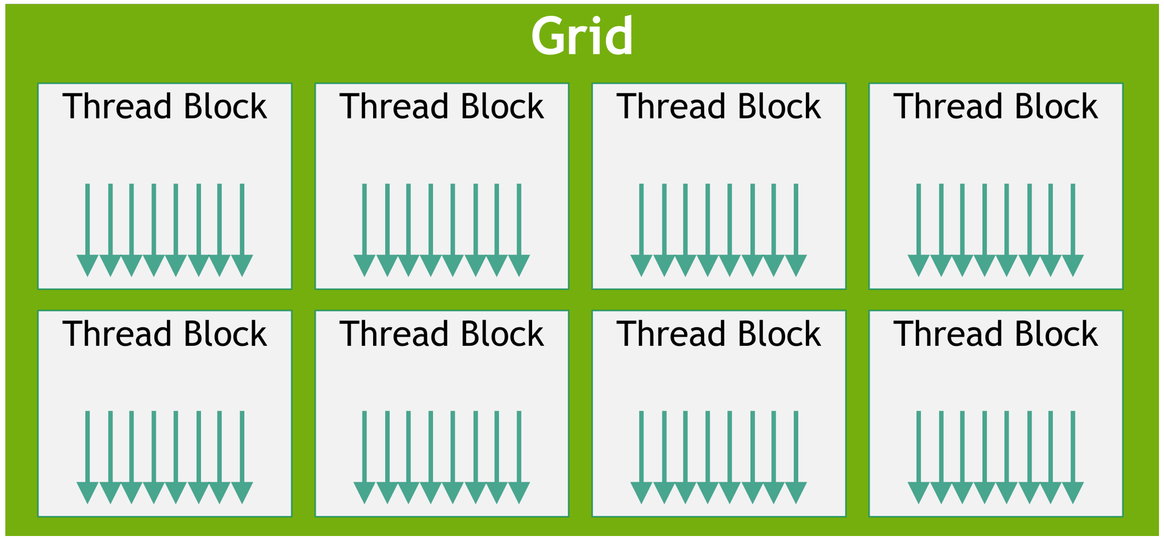
- Threads are grouped in **thread blocks**
  - Required to execute independently
- Threads within a thread block can cooperate
  - Shared memory (see later)
  - Synchronization primitives

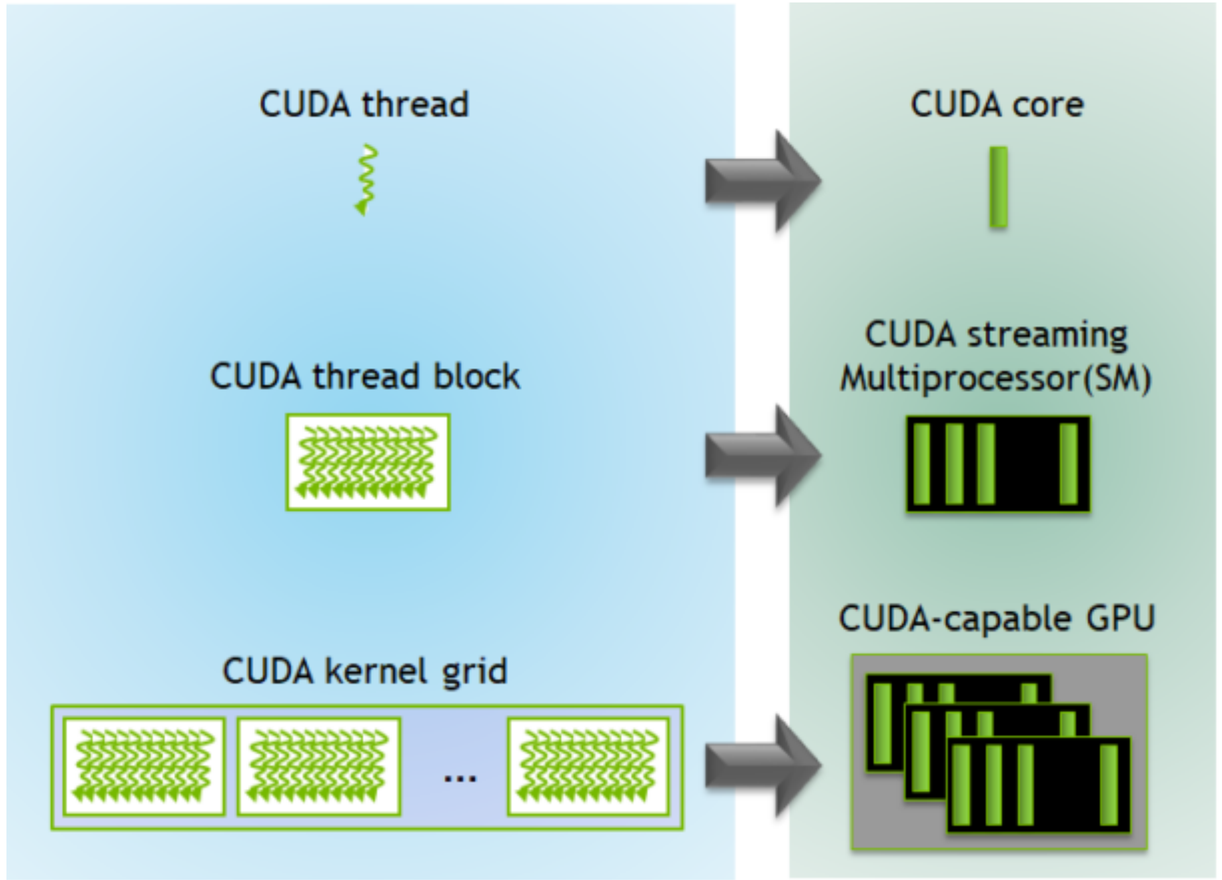


https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Physical level for CUDA model

- Thread blocks are grouped in **grids**
- **Thread blocks** and **grids** can be logically arranged in 1D, 2D, 3D

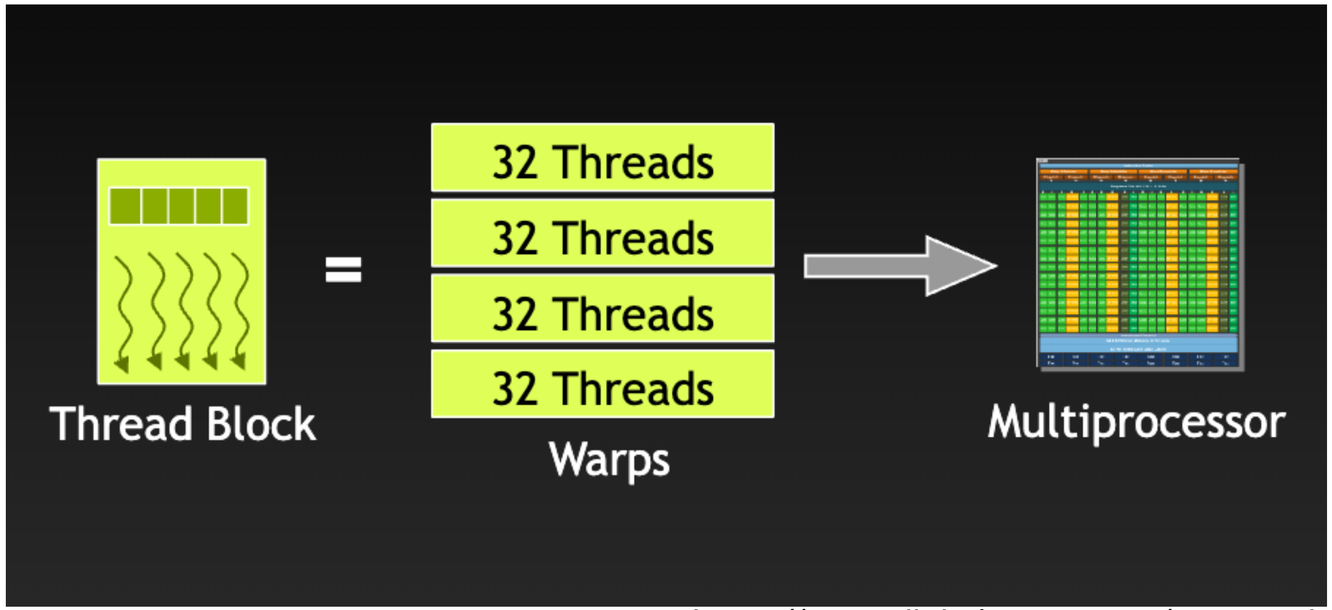




https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

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
- Like in a thread block, threads in a warp can cooperate via warp-level primitives

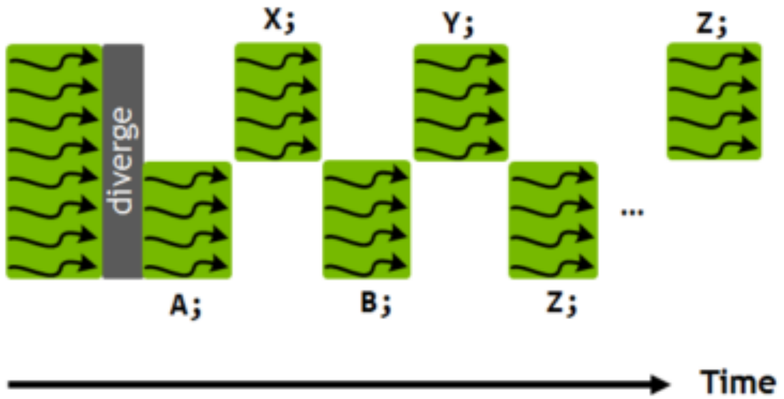


https://www.digitalocean.com/community/tutorials/the-role-of-warps-in-parallel-processing

Physical level for CUDA model

- **Single Instruction Multiple Threads** (SIMT) - threads in a warp execute the **same instruction during every clock cycle**
  - Branch divergence is processed accordingly (*and can affect performance*):

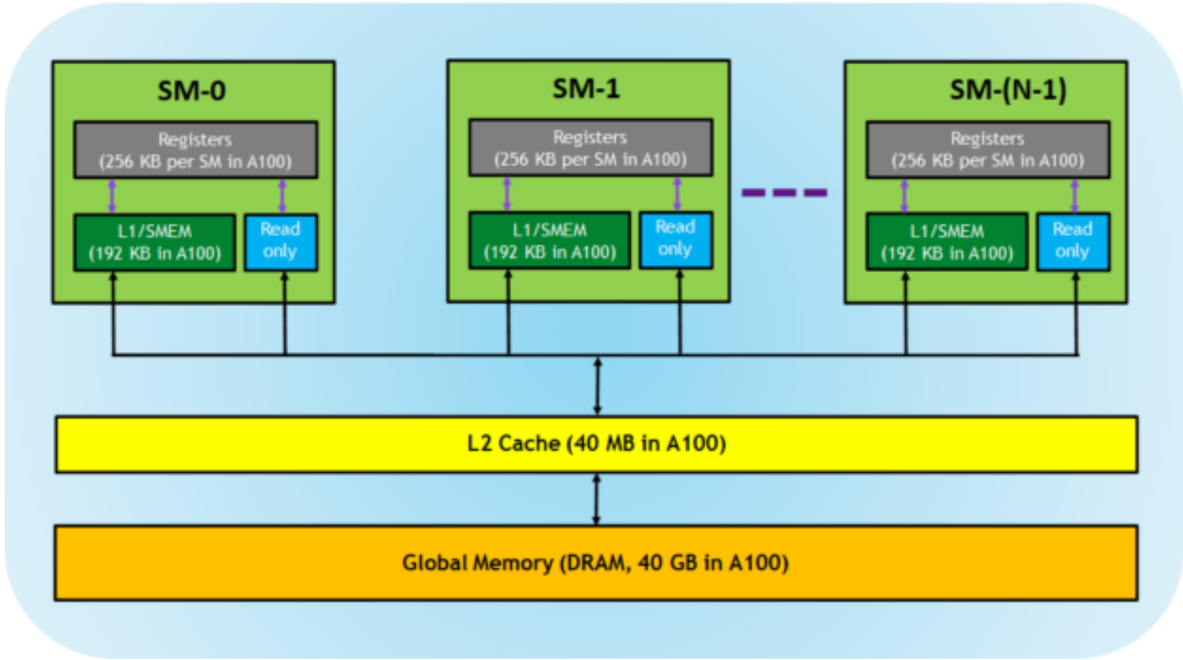
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



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

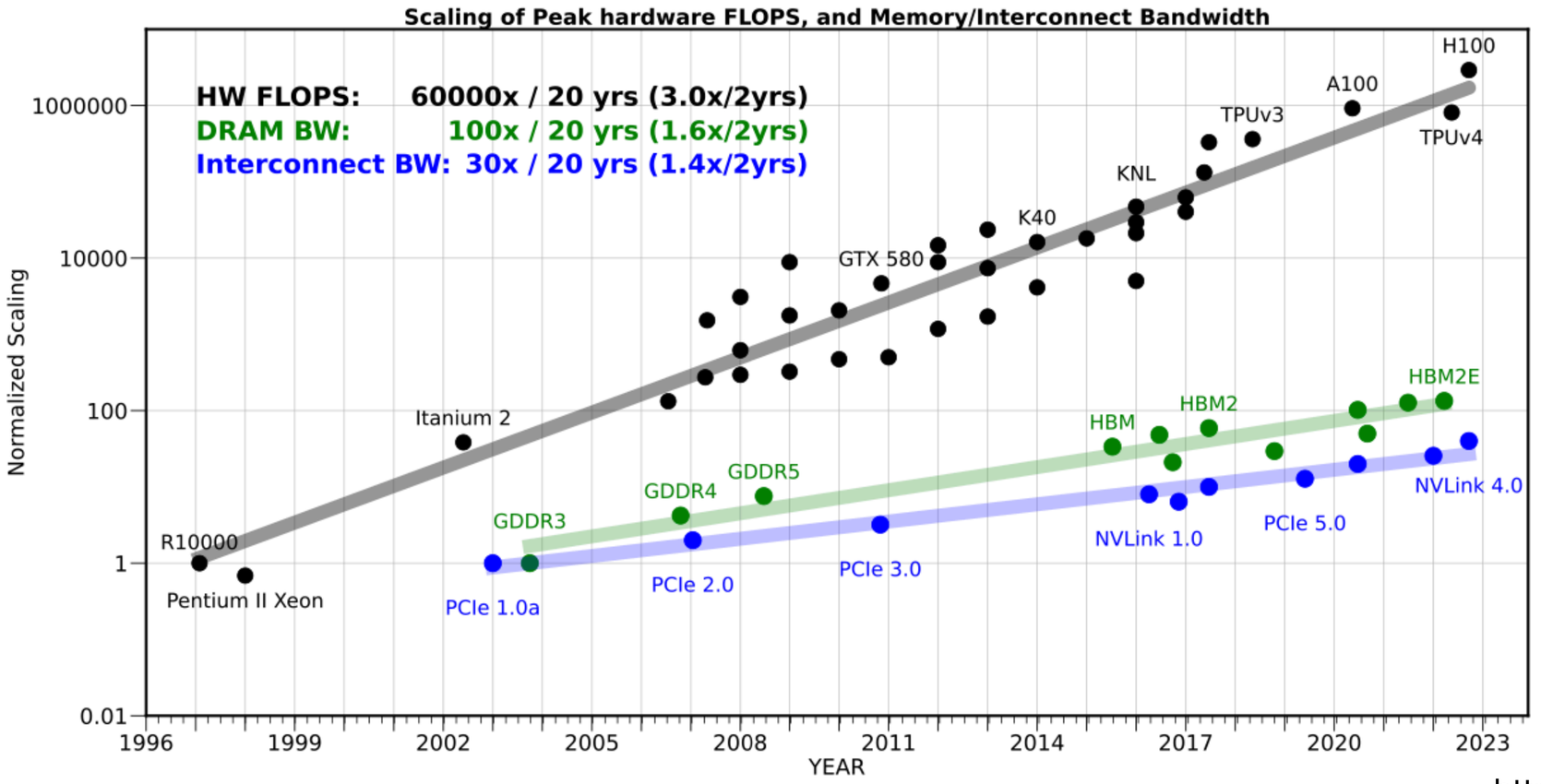


Memory hierarchy on Nvidia A100-40Gb

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

GPU memory

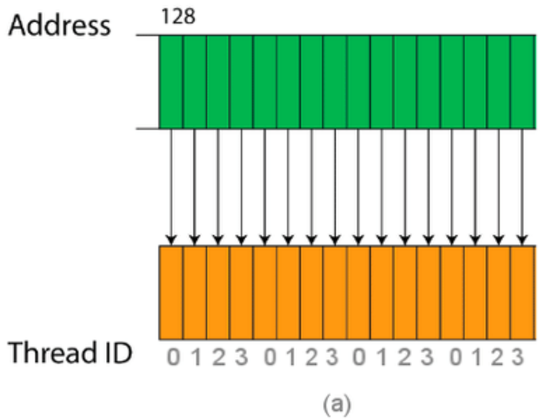
- The rate at which bandwidth increases is slower as the increase in computational power
- $\Longrightarrow$ Usually, this is the memory which is a bottleneck



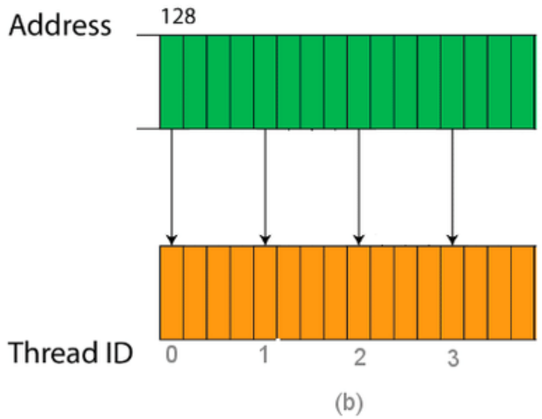
https://arxiv.org/abs/2403.14123

GPU memory. Global memory

- Accessible for all threads
- Slowest, but largest in size
- Accessed by vectorized memory transactions via 32-, 64-, 128-byte
  - Multiple threads accesses can be packed in a single transaction (*coalesced memory access*)



(a) coalesced memory access

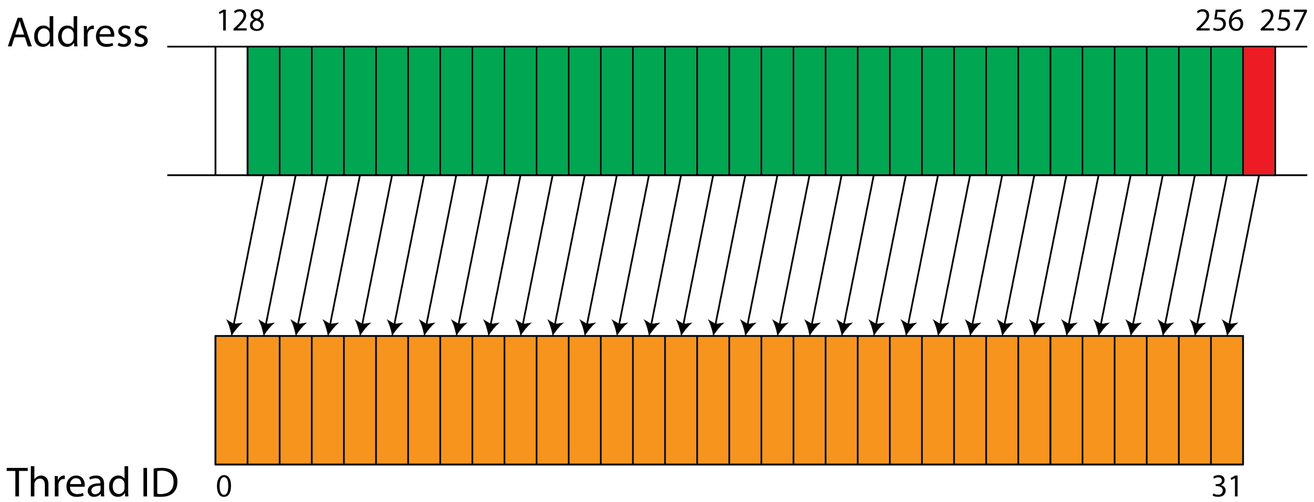


(a) uncoalesced memory access. Multiple transactions

https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses

GPU memory. Global memory

- Accessible for all threads
- Slowest, but largest in size
- Accessed by vectorized memory transactions via 32-, 64-, 128-byte
  - Multiple threads accesses can be packed in a single transaction (*coalesced memory access*)
  - **Addresses should be aligned** — starting address must be divisible by 128



https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses

GPU memory. Shared memory

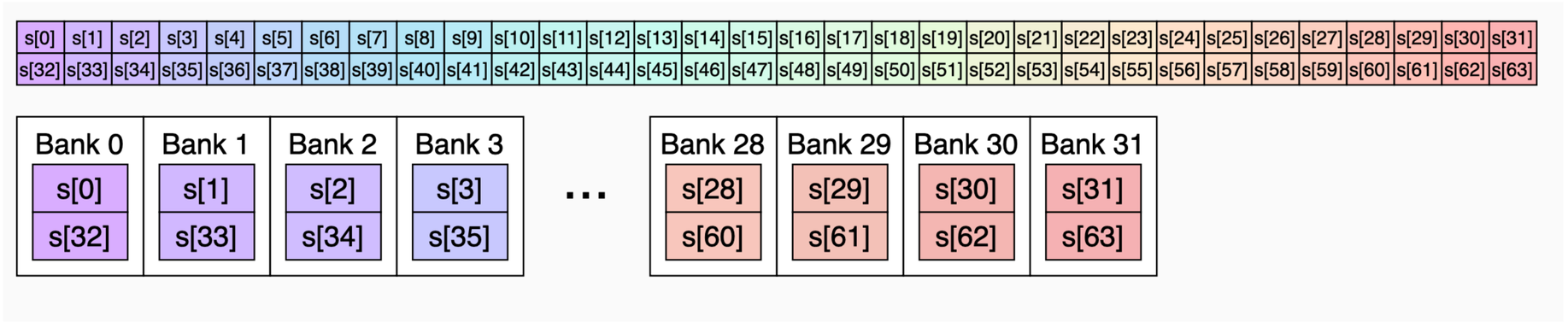
- Shared memory is directly on the chip $\Longrightarrow$ lower latency than global memory
- We can’t share memory between ALL cores & can’t get this fastest access for HBM
- Accessible for all threads **in a thread blocks**

https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses

GPU memory. Shared memory

- Shared memory is directly on the chip $\Longrightarrow$ lower latency than global memory
- We can't share memory between ALL cores & can't get this fastest access for HBM
- Accessible for all threads **in a thread blocks**
- Organized into 32 banks (not a coincidence!)
- $\text{bank}(\text{address}) := \left(\frac{\text{address}}{4}\right) \% 32$

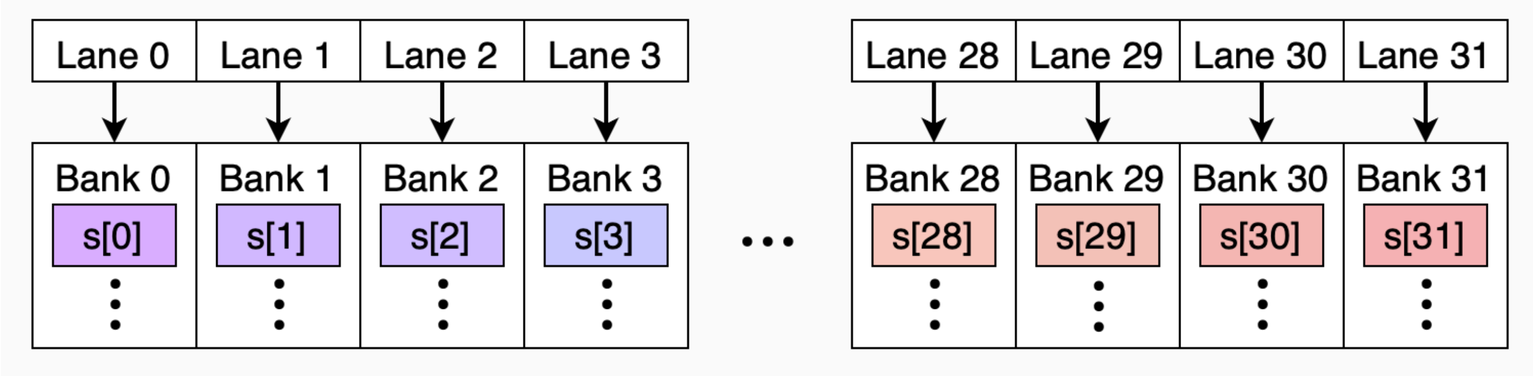
```
__shared__ float s[64];
```



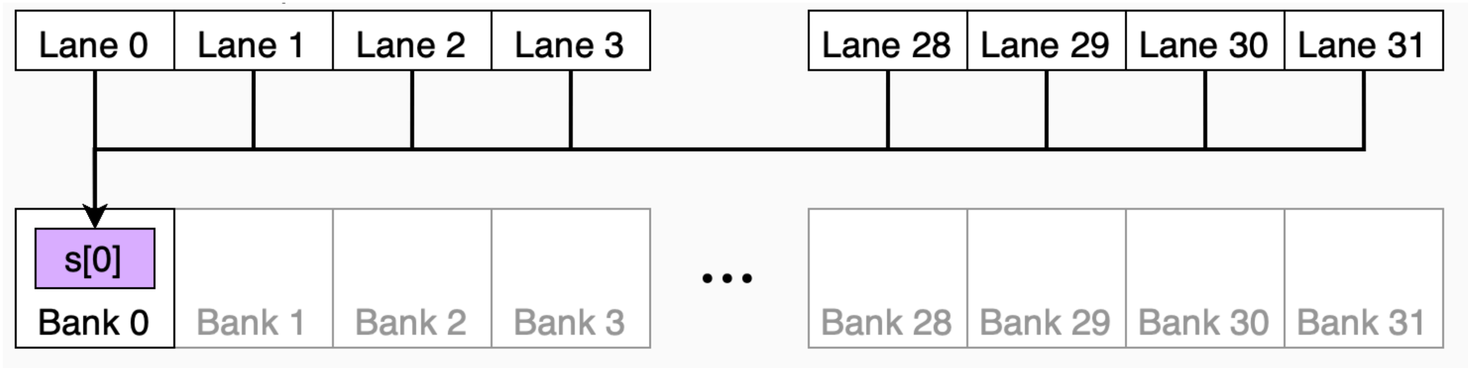
https://feldmann.nyc/blog/smem-microbenchmarks

GPU memory. Shared memory. Conflict-Free Access

- If threads access the different banks — no bank conflicts and load everything as fast as possible (consider *lane* as an individual *thread* in a warp):



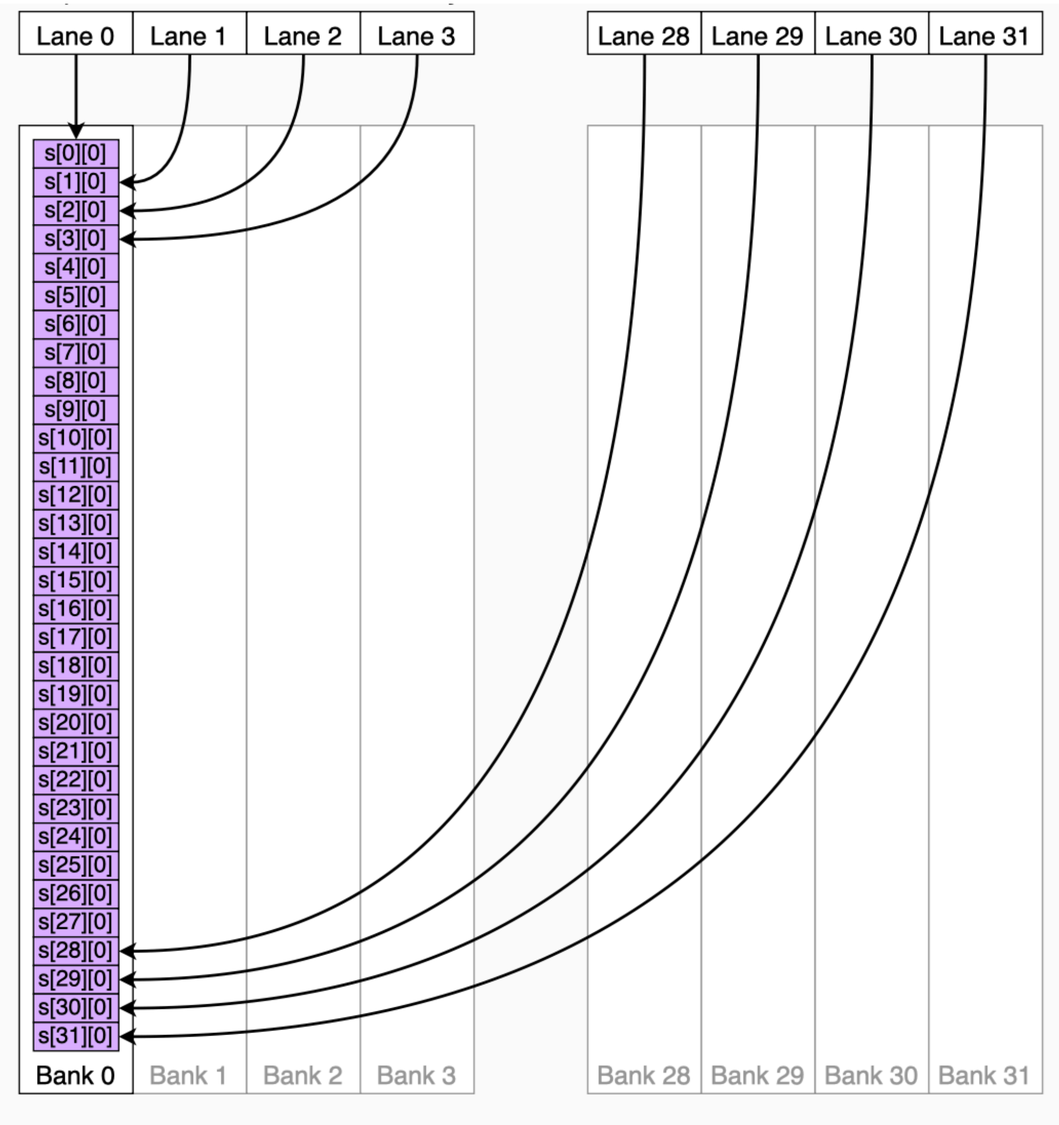
- If threads access the same bank — no conflict due to broadcasting:



https://feldmann.nyc/blog/smem-microbenchmarks

GPU memory. Shared memory. Bank conflicts

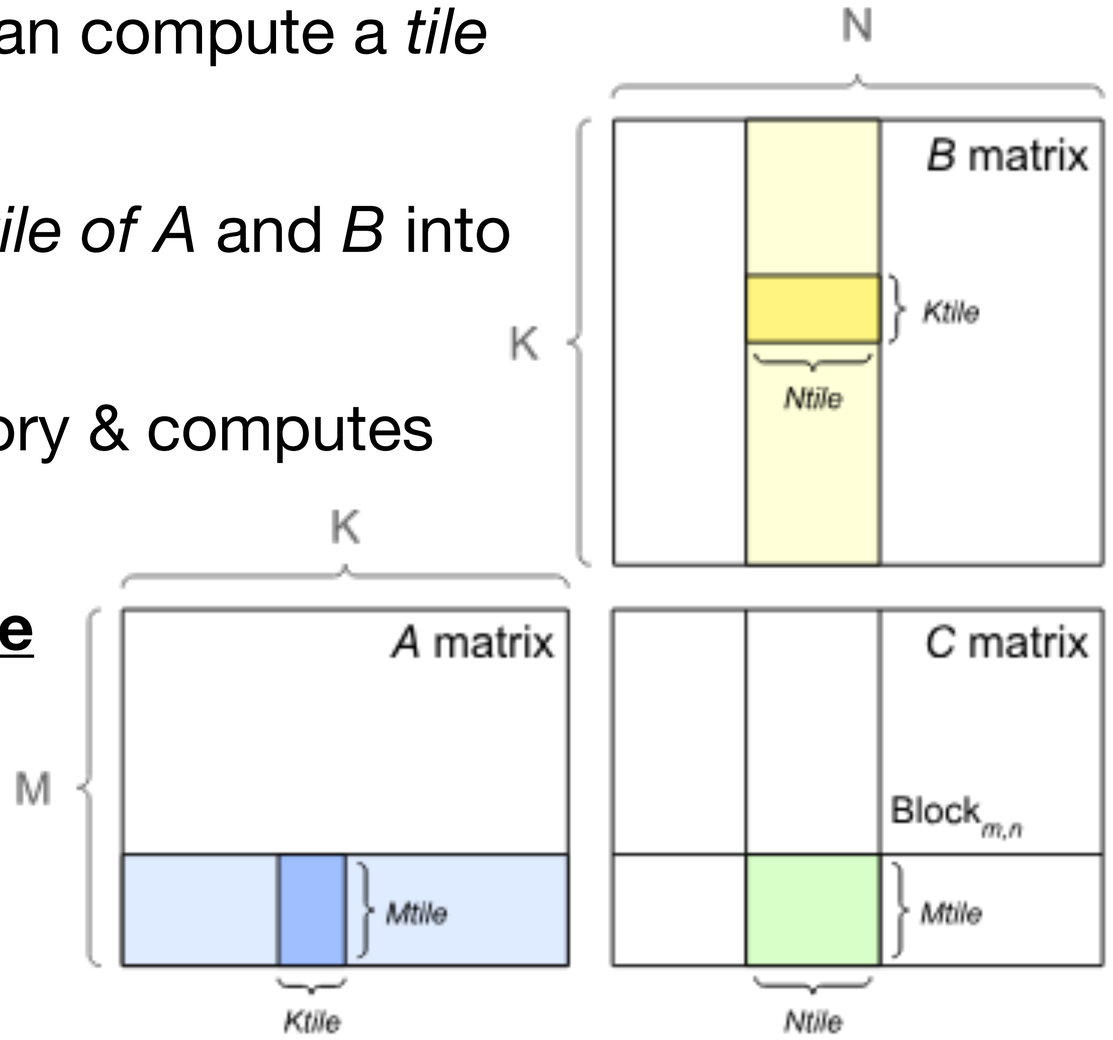
- If threads access different addresses in a bank — bank conflict occurs
- Banks can only serve one load per cycle
- The access will be serialized



https://feldmann.nyc/blog/smem-microbenchmarks

GPU memory. Shared memory example usage

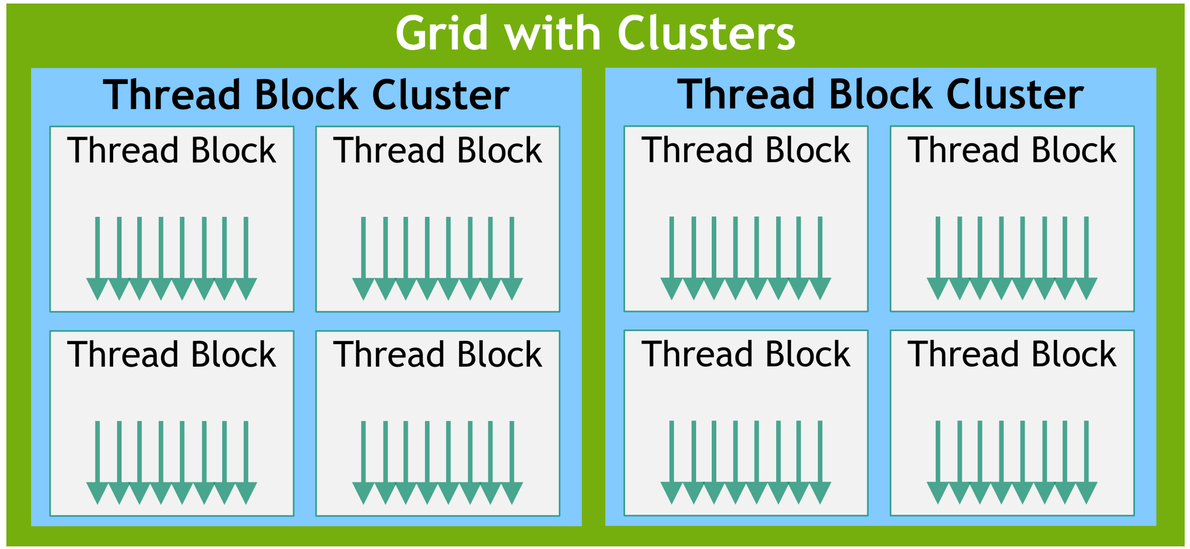
- Accessible for all threads **in a thread block**
- In matrix multiplication, one thread block can compute a *tile* of the output matrix:
  - Each thread loads one element of *input tile of A* and *B* into the shared memory
  - Each thread accesses only shared memory & computes one element of *output tile of C*
- Only 2 reads and 1 write **per thread <u>per tile</u>**
- IO is reduced proportionally to tile size

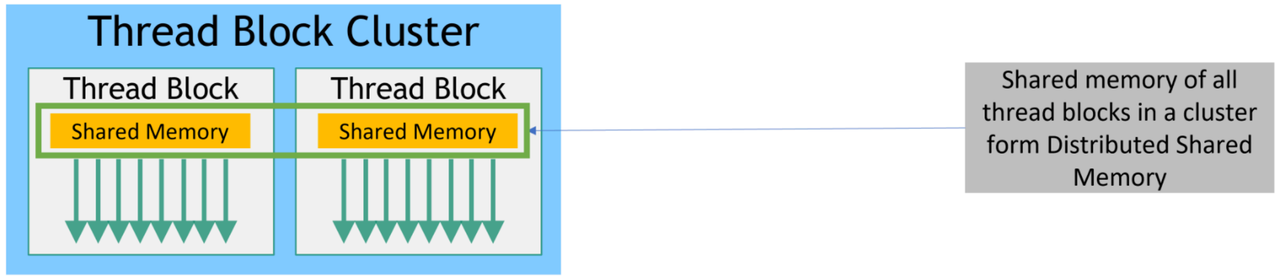


https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses

How to communicate between thread blocks?

- Cooperative groups API — can synchronize between block clusters
- On Hopper Architecture (H100, H200)— thread clusters can communicate within distributed shared memory
  - Thread block clusters are guaranteed to be co-scheduled on a GPU Processing Cluster (GPC) in the GPU (*cluster os SMs*)





https://docs.nvidia.com/cuda/cuda-c-programming-guide/#thread-block-clusters

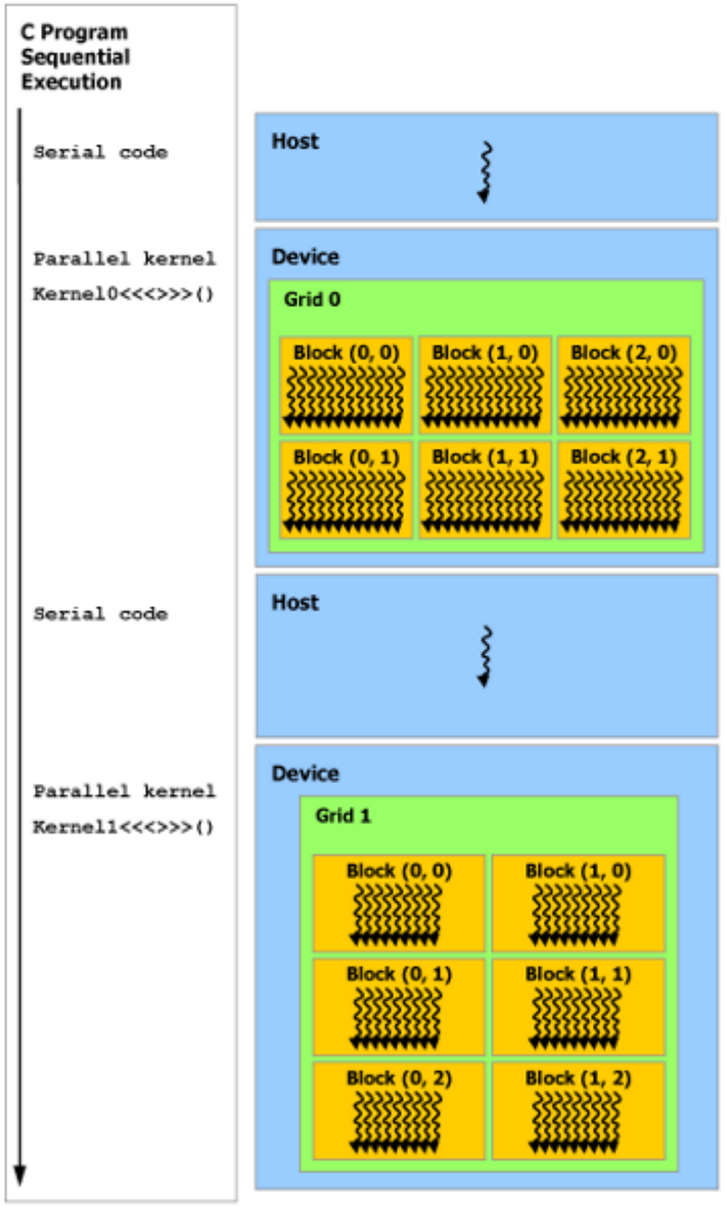
How to communicate between thread blocks?

- Cooperative groups API — can synchronize between block clusters
- On Hopper Architecture (H100, H200)— thread clusters can communicate within distributed shared memory
- Atomic operations on Global Memory — safe updates to shared variables without race conditions
  - No other thread can interrupt the execution of currently started operation

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

Heterogeneous Programming Approach

- Host — CPU — launches a program
- Device — GPU — executes the program
- Device & Host synchronize
- Something similar, right?



Thanks for attention!