Author - Keerthi Ningegowda

This document contains my notes on GPU architecture and CUDA fundamentals—essential prerequisites for writing custom kernels and PyTorch programs that execute on GPUs. The scope is limited to single-GPU, not GPU clusters.

Resources:

1. https://developer.nvidia.com/cuda-toolkit
2. https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html
3. Stanford CS336 Language Modeling from Scratch - Lectures 5 and 6
4. Claude, Gemini, and ChatGPT

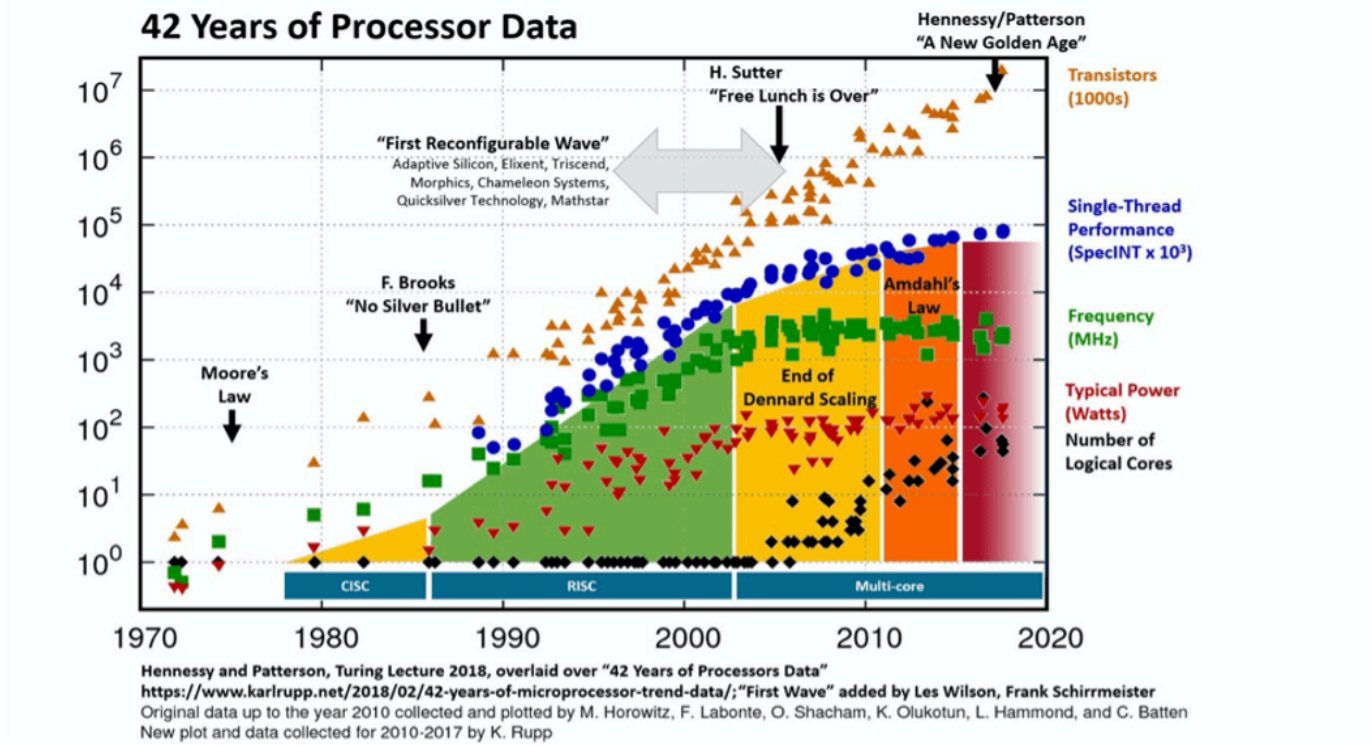
1. Graphics Processing Unit (GPU)

Originally, GPUs were designed primarily for graphics rendering and visual display tasks. Over time, however, their capacity for parallel computation made them attractive for mathematical and scientific workloads, leading to the emergence of General-Purpose GPUs (GPGPUs).

In 2007, NVIDIA introduced the Compute Unified Device Architecture (CUDA), a programming model that gave developers direct programmatic access to GPU hardware, enabling them to run custom computational workloads on the GPU.

2. Parallel computing - Beauty and the Beast - What made GPUs lucrative in computing?

Note:- This article covers chip design evolution and includes interesting perspectives on future trends.
https://semiengineering.com/chip-design-shifts-as-fundamental-laws-run-out-of-steam/



2.1 Moore's Law and the CPU vs. GPU Trade-off

GPUs effectively capitalized on Moore's Law—the observation that the number of transistors on a chip doubles approximately every 2 years.

Does this mean CPU parallelism was a wasted effort? Not at all. CPUs and GPUs optimize for fundamentally different metrics: **latency versus throughput**. While CPUs leveraged Moore's Law to make individual "jack-of-all-trades" cores faster (minimizing latency), GPUs used it to execute many tasks simultaneously (maximizing throughput).

2.2 Pivotal Moment - The End of Dennard Scaling

Dennard scaling—the principle that transistors could be shrunk while maintaining constant power density by proportionally reducing voltage—enabled Moore's Law to continue for decades. However, this scaling eventually hit physical limits.

Once we could no longer add more transistors without increasing chip area and power consumption, thermal management became a major challenge. This is indeed a key reason why modern computers generate so much heat!

2.3 Amdahl's Law and CUDA's Solution

Amdahl's Law provides a reality check for parallel computing: it states that the overall speedup of a program is fundamentally limited by the portions that must execute sequentially, regardless of how much parallel hardware you add.

This insight shifted focus from simply adding more transistors to thoughtfully designing how programs execute. CUDA was transformative here—it enabled developers to offload highly parallelizable portions of their applications to the GPU while keeping the necessarily sequential parts on the CPU.

Anatomy of GPU

Hierarchy of processing units in a GPU

- GPU (entire chip)
- GPC (Graphics Processing Cluster) - contains multiple SMs
- SM (Streaming Multiprocessor) - the image below
- Processing Block/Partition - a block within that image
- CUDA Core/Tensor Core - individual execution units

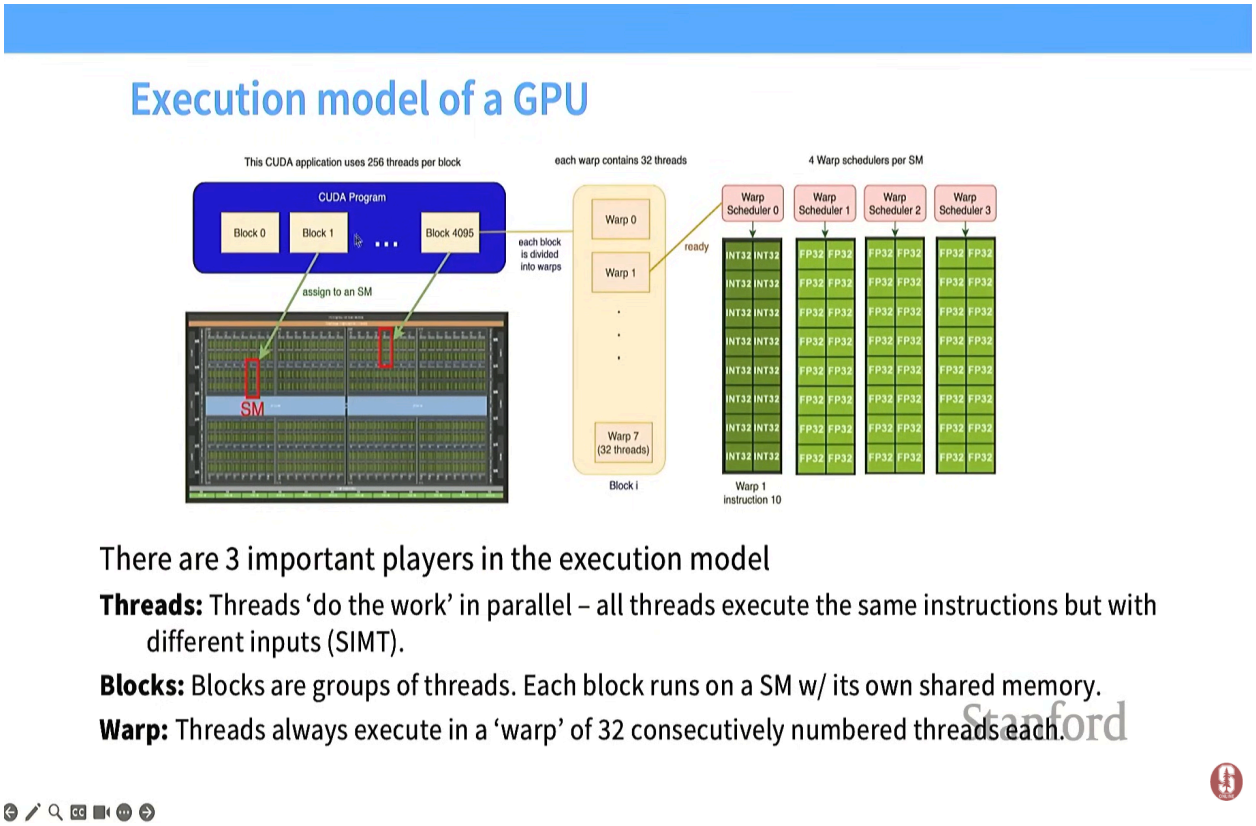
1) **<u>Streaming Multi-processors (SMs)</u> -** It is a hardware unit that can execute an instruction using GPU threads. They consist of multiple cuda core(The green blocks that say FP64, FP32, INT), tensor cores(specialized in Matrix Multiply-Add (MMA) operations), registers, cache memory, warp schedulers etc., It also consist of LD/ST(Load store units for memory access) and SFU (Special Function Units). Each block in that SM (there are 4 of them) is called a processing block or a SM partition. Sometimes it is also called a Quad.
2) **<u>Threads</u>** - These are atomic units within a GPU. They are the ones that execute a same program instruction but with different data
3) **<u>Warps</u>** - These are a combination of 32 threads. Threads dont execute alone, they always go in groups of 32

4) **Blocks** - Collection of threads - Usually a combination of 128 or 256 threads. Each block runs on a SM and will have access to shared memory.
5) **Grid** - A collection of blocks across the entire GPU.

Some caveats to keep in mind.

1) Threads in a warp cannot execute different instructions. They will execute the same operation- **Single Instruction Multiple Threads (SIMT)**
2) Threads in a block can execute different instructions.
3) GPU threads are most said to be efficiently utilized if the arithmetic intensity(W/Q) -> Work done / Memory access is higher.

Check out this image from Stanford, that gives a good presentation of what GPU core execution model looks like.

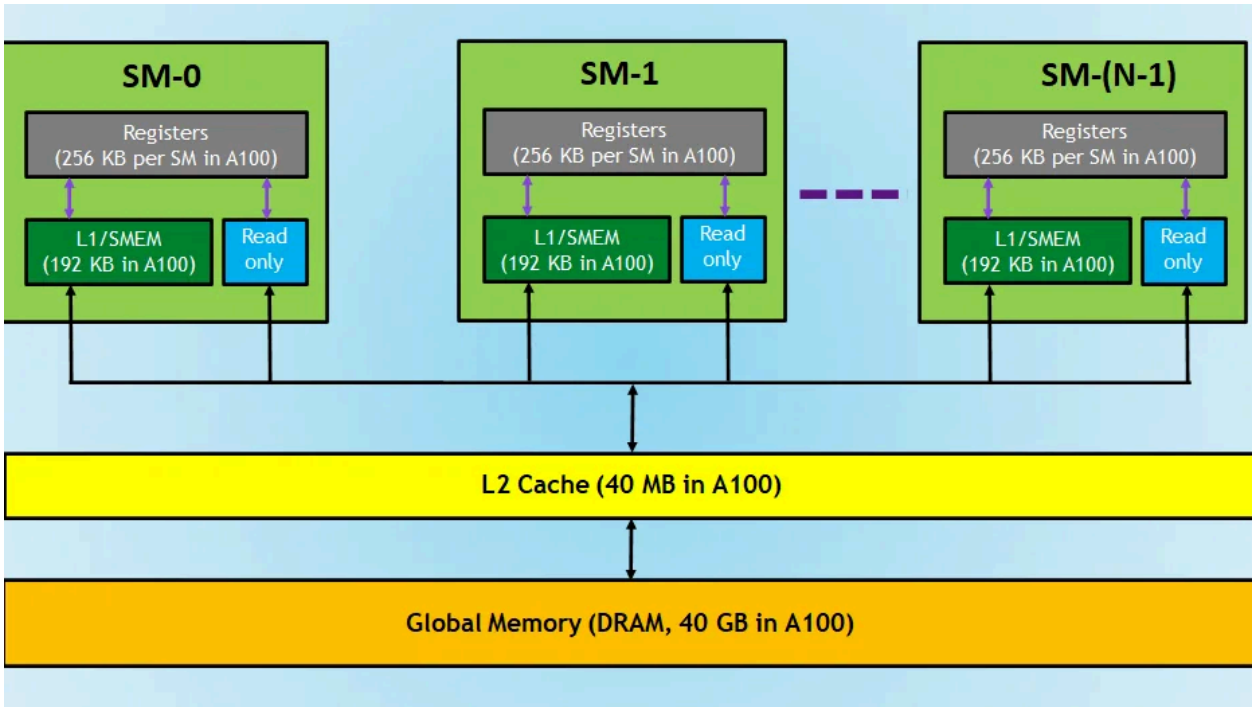


Memory hierarchy in a GPU

From fastest -> slowest

1) Registers
2) Shared memory/L1 cache - sit on GPU
3) L2 cache
4) Global Memory (VRAM)
5) Host (CPU) Memory

See this representation from Nvidia blog here



Long story short the closer the memory unit is to the CPU the slower it is.

Kernel - A kernel is a function/instruction that runs on GPU across many threads. Its like running the function a+b across 100 threads but each time with different data

Key phenomena to keep in mind

1) **Embarrassingly Parallel Execution** Once the CPU queues work to the GPU, all GPU threads execute asynchronously and independently—making GPUs ideal for problems with minimal interdependencies.
2) **Memory Coalescing** Memory accesses should be contiguous and stride-based. Non-coalesced memory access patterns are extremely expensive on GPUs. If your data structure isn't amenable to contiguous access patterns (like matrices), GPUs may not be the optimal choice.
3) **Warp Divergence** GPU threads execute in SIMT (Single Instruction, Multiple Thread) fashion. Branching statements like if-else cause warp divergence: when threads in a warp take different paths, both paths must execute serially (with some threads masked), reducing parallelism and performance.
4) **Operator/Kernel Fusion** Fusing multiple operations into a single kernel avoids expensive intermediate memory writes and reads.

   Consider an example:- Some activation function - x * sigmoid(x)

   Sig = 1 / 1+ exp(-x)
   Final_result = x * sig

The first instruction creates intermediate result which means not only compute time should be considered but also the sig will be written and read back from memory

Instead you can fuse it to

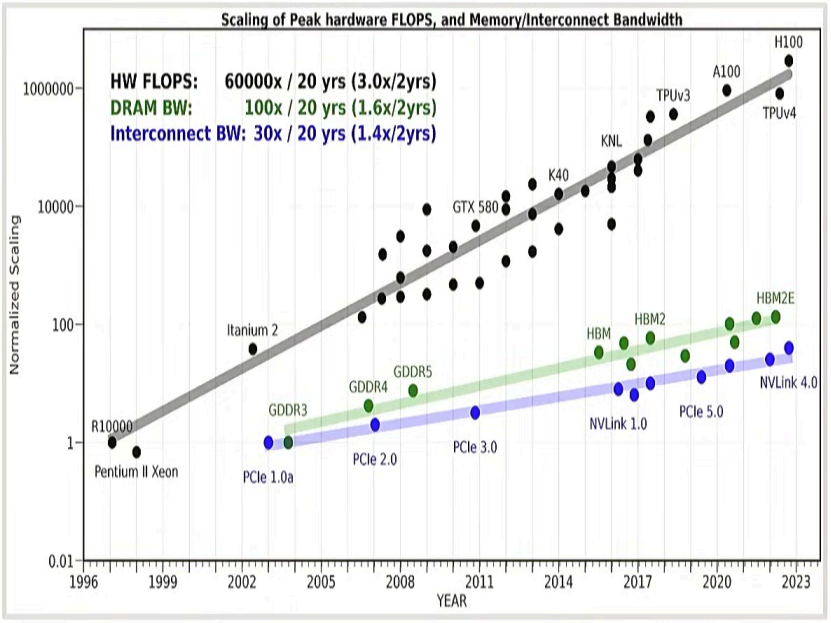
Final_result = x * (1 / 1 + exp (-x))

5) **Tiling** Tiling groups threads to minimize global memory access by loading data into faster shared memory. Key considerations:
   - **Coalesced memory access** - Use row-major access patterns for coalescing

- **Shared memory limits** - Tile size constrained by SM shared memory capacity
- **Divisibility** - Matrix dimensions may not divide evenly by tile size
    - Example: 256×256 matrix with 128×128 tiles ✓ (clean division)
    - Example: 257×257 matrix with 128×128 tiles ✗ (underutilized SMs)
- **Padding strategy** - Pad matrices to powers of 2 for efficient tiling (since warp size = 32)
- **Common tile sizes** - 256×128 is typical/default for matrix multiplication

Why are GPUs mostly memory-bound and not compute bound?

Compute scaling is faster than memory scaling



https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

FLOPs scale faster than memory – it's hard to keep our compute units fed with data!

3 important components

1) Connectivity - PCIe, NVLink for GPU-GPU or GPU-CPU connectivity eg:- data transfers
2) DRAM - Memory where the data is stored
3) HW FLOPS - compute

See the gap between 1,2 and 3

The gap between “memory components” and the hardware components is large which tells that the advancements in memory as it relates to GPU has not as grown as the computer power

Where do GPUs struggle/ will be inefficient?

1) Memory access is not contiguous
2) The algorithms that do not access the data in a contiguous way i.e. Graphs. Though there has been some advancements with eg:- CuGraph
3) Heavy branching - control flow divergence
4) Using high precision numbers like there is no tomorrow - Using FP64 instead of FP32
5) Frequency syncing between CPU-GPU - Don’t use print statements too much when training your models
6) Sequential problems or the ones that depend on results of previous steps - Like reduction problems

So can I use Spark/Dask instead of GPU parallelism?

DO NOT over-engineer your solution to use spark/dask when you can get it over with a single GPU.

People sometimes conflate distributed frameworks (Dask/Spark) with GPU libraries (CuPy/PyTorch), which is a mistake—they're designed for fundamentally different scenarios. Using Dask/Spark for single-node workloads is overkill and introduces unnecessary overhead.

Scale:

- **Spark/Dask:** Handles terabytes to petabytes across distributed clusters
- **GPU (CuPy/CUDA):** Handles datasets that fit in GPU memory (typically 8-80 GB per GPU)

Use Case:

- **Spark/Dask:** Data engineering, ETL pipelines, distributed data processing, large-scale batch jobs
- **GPU:** Numerical computing, linear algebra, deep learning, scientific computing, high-performance matrix operations

Performance Characteristics:

- **Spark/Dask:** Higher latency due to network communication and serialization overhead, but scales horizontally across nodes
- **GPU:** Extremely fast for parallel mathematical operations with minimal overhead, limited by single-device memory

When to Use What:

Use Spark/Dask when:

- Data exceeds single-machine memory capacity
- Performing data transformations, joins, aggregations, or ETL on massive datasets
- Fault tolerance and distributed coordination are required
- The workload is I/O-bound or data movement-heavy

Use GPU when:

- Data fits comfortably in GPU memory
- Performing compute-intensive numerical operations (matrix multiplication, convolutions, FFTs)
- Maximum throughput for mathematical operations is critical
- Training deep learning models or running scientific simulations

Key Insight: Don't use distributed frameworks for problems a single GPU can handle—the coordination overhead will hurt performance rather than help.

It is also worth noting that there are Dask-CUDA and Spark RAPIDS versions. So choose deliberately and carefully.