Computer Architecture: Introduction to GPU Architecture and Programming

Hossein Asadi (asadi@sharif.edu)
Department of Computer Engineering
Sharif University of Technology
Spring 2025

Copyright Notice

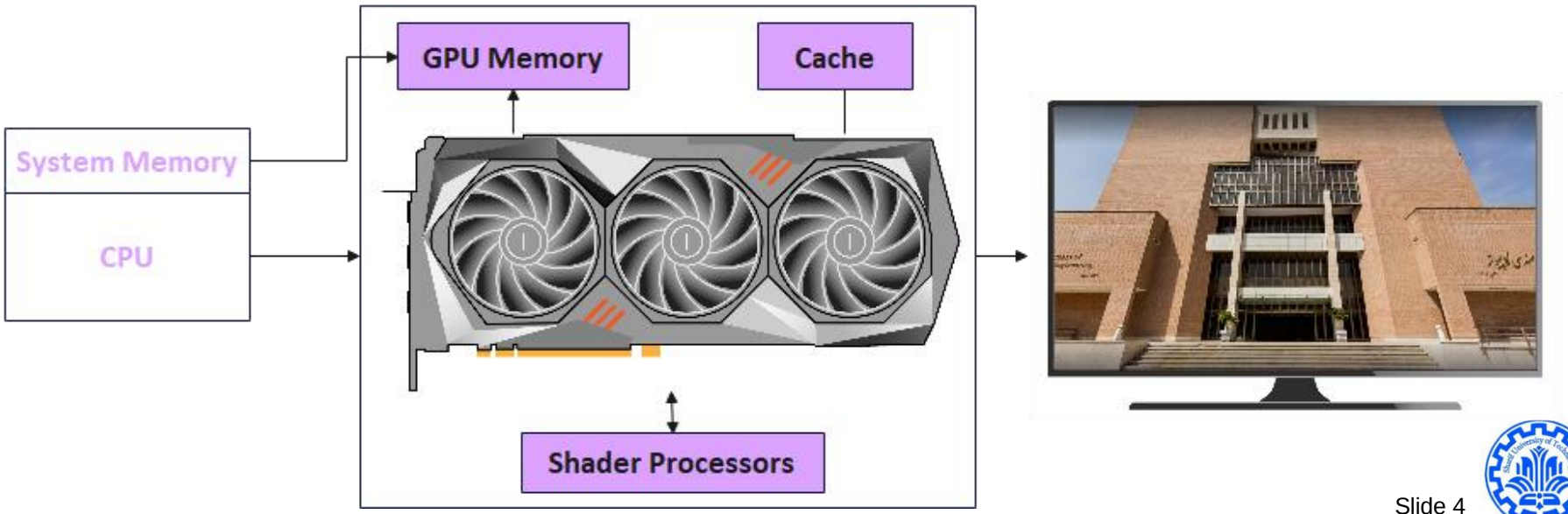
- Some Parts (text & figures) of this Lecture adopted from following:

Topics Covered in This Lecture

- Intro to GPUs & Parallelism
- GPU Architecture Overview
- SM Architecture: Inside the Streaming Multiprocessor
- CUDA Cores & Warp Execution
- Warp Scheduling & Latency Hiding
- Memory Hierarchy: Registers, Shared, Global
- Load/Store Units (LSUs)
- Special Function Units (SFUs)
- Tensor Cores
- Tensor Cores vs CUDA Cores

What is a GPU?

- **GPU = Graphics Processing Unit**
- Originally built for **image rendering** in 1990s
- Became widely known with **NVIDIA GeForce 256 (1999)**
  - marketed as the first “GPU”
- Designed to handle **parallel operations** – many tasks at the same time
- Especially good at **processing thousands of threads simultaneously**
- Powers **modern tasks** like: Gaming, AI & Machine Learning, Video editing & rendering, Crypto mining, and Scientific simulations



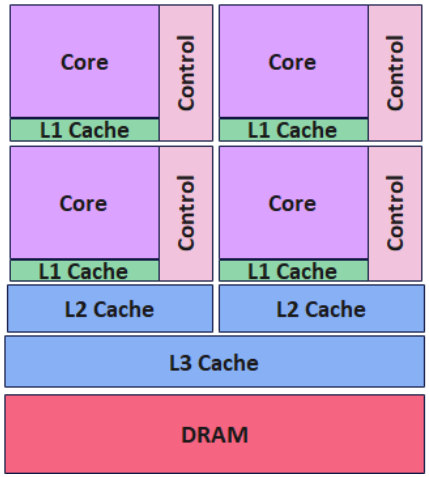
GPU vs CPU

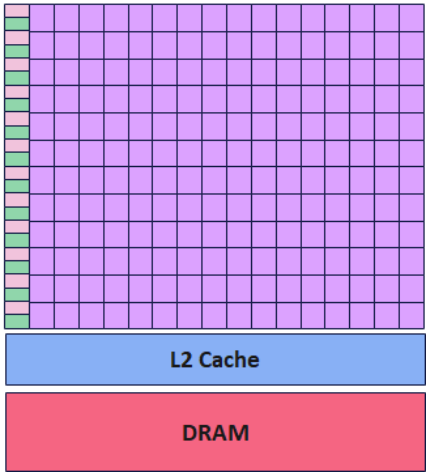
CPU:

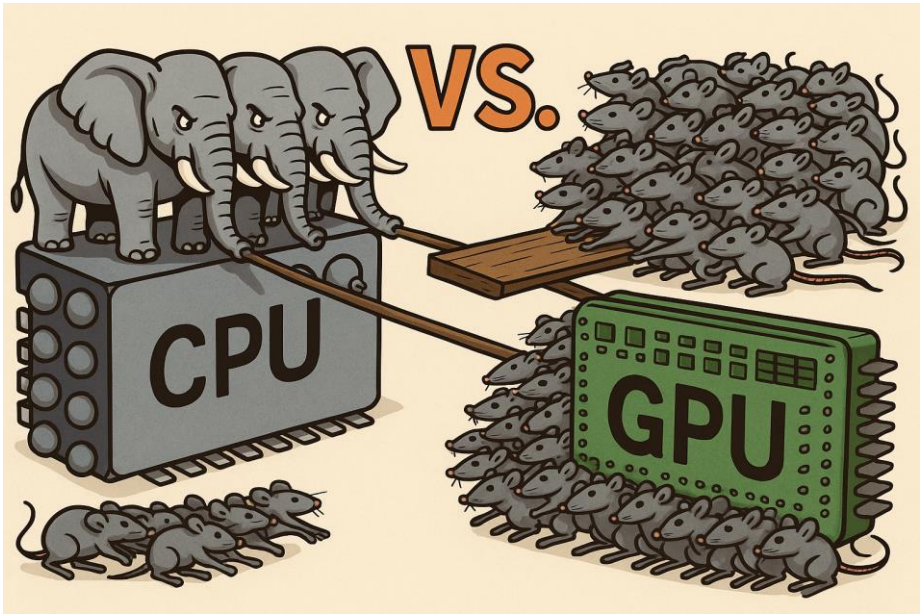
- Few **powerful, complex cores**
- **Optimized for sequential** tasks (1 or few threads at a time)
- Has **large caches** (L1, L2, L3)
- Great at tasks with **branching, logic, or low parallelism**
- Example: Running OS, web browsing, compiling code

GPU:

- **Many simple cores** (hundreds to thousands)
- Built for **parallel execution**
- Best for **same operation on lots of data** (SIMD-style)
- Great for graphics, ML training, simulations
- Designed for **high throughput**, not high latency sensitivity







GPU vs CPU (Cont.)

CPU:

- Few powerful, general-purpose cores
- Prioritizes low latency & sequential logic
- Designed for task switching & responsiveness

GPU:

- Many simple, specialized cores
- Optimized for massive data parallelism
- High throughput with smart scheduling

Feature	CPU	GPU
Core Count	4~128	Hundreds to thousands
Task Style	Sequential	Parallel
Cache Size	Large	Small
Latency	Low	High (but hidden with warps)
Best Use Case	Logic-heavy tasks	Data-parallel computations

Instruction Dependencies

Case	Instruction Style	Can Run in Parallel?	Why?
1	**Dependent**	No	Later instructions need earlier results
2	**Independent**	Yes	Each one is free to execute alone

Case 1:

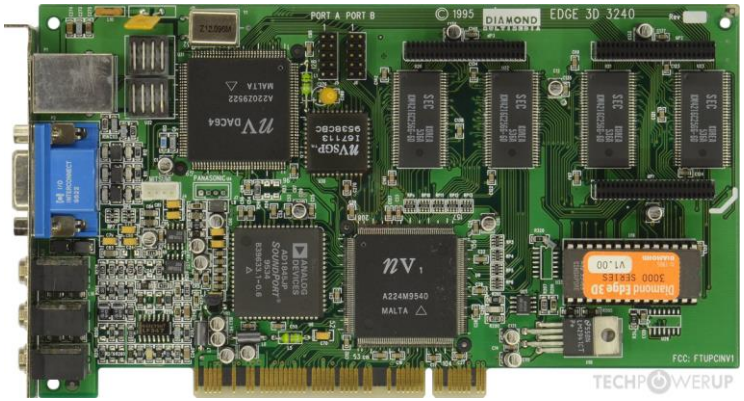
- Add R1, R2, R3
- Mul R4, R1, R5 ← uses result of Add
- Div R6, R4, R7 ← uses result of Mul

Case 2:

- Add R1, R2, R3
- Mul R4, R5, R6
- Div R7, R8, R9

The NVIDIA Journey

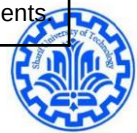
- **1993** – Jensen Huang founded **NVIDIA**.
- **1995** – NVIDIA launches its first product: **NV1**.
- **1997** – **RIVA 128** launches:
  → **1 million units sold in 4 months**
  → First real success, **3D acceleration**, for **desktop PCs**
- **1999** – **NVIDIA invents the GeForce GPU** ($12):
  → Considered a revolution in computing and graphics.



GPU Name	Processor	Pixel Shaders	Vertex Shaders	TMUs	ROPs	Memory Size	Memory Type	Bus Width
NVIDIA NV1	NV1	1	N/A	1	1	2 MB	EDO	64-bit
RIVA 128	NV3	1	N/A	1	1	4 MB (up to 8 MB)	SGRAM	128-bit
GeForce 256	NV10	4	N/A	4	4	32 MB	SDR	64-bit

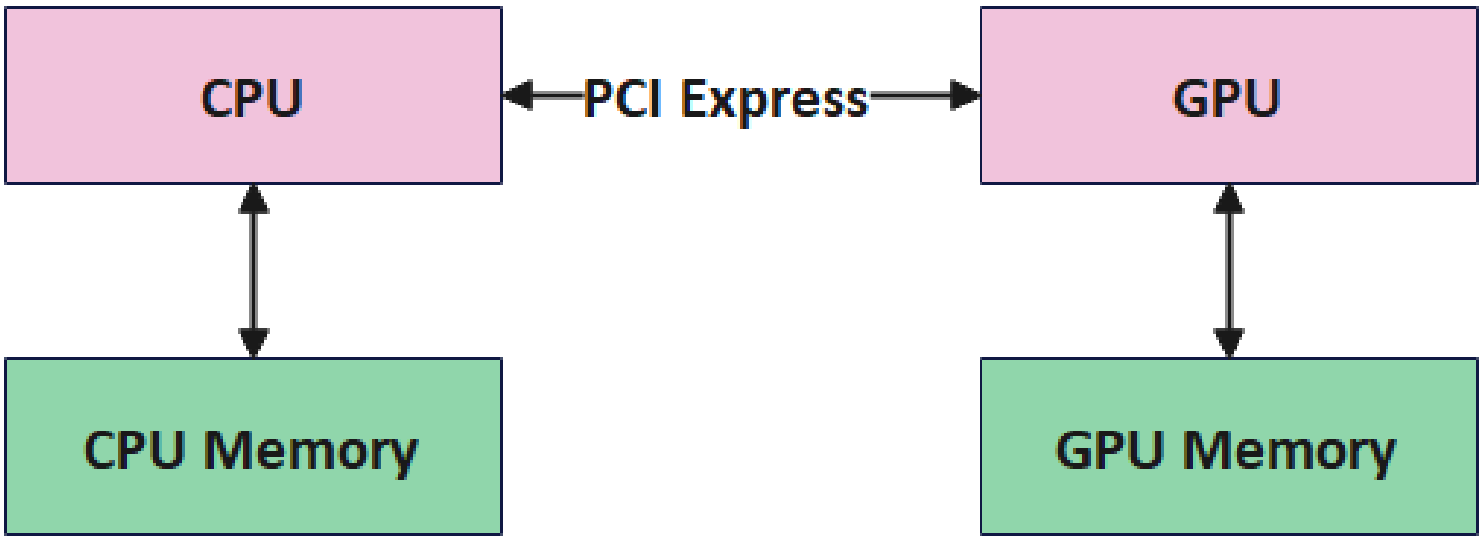
NVIDIA GPU Architecture Timeline

Years	Architecture	Named After	Key Highlights
1998–2000	**Fahrenheit**	Daniel Fahrenheit	Pre-GPU T&L era; fixed-function pipeline.
1999–2001	**Celsius**	Anders Celsius	GeForce 256: first "GPU" with T&L engine.
2001–2003	**Kelvin**	Lord Kelvin	Programmable shaders introduced.
2003–2005	**Rankine**	William Rankine	Shader Model 2.0, DX9 improvements.
2003–2013	**Curie**	Marie Curie	Shader Model 3.0, SLI multi-GPU.
2006–2010	**Tesla**	Nikola Tesla	Unified shaders, CUDA launched.
2007–2013	**Tesla 2.0**	—	Professional/HPC variants.
2010–2016	**Fermi**	Enrico Fermi	ECC, scalable parallelism, SMs.
2010–2013	**VLIW Vec4**	— (Integrated GPUs)	Used in Tegra/embedded chips.
2012–2018	**Kepler**	Johannes Kepler	Energy efficiency, GPU Boost.
2013–2015	**Kepler 2.0**	—	GK110-based refinement.
2014–2017	**Maxwell**	James Clerk Maxwell	Efficiency & NVENC, low power.
2016–2018	**Pascal**	Blaise Pascal	HBM2, NVLink, deep learning.
2017–2020	**Volta**	Alessandro Volta	Tensor Cores introduced.
2018–2021	**Turing**	Alan Turing	Real-time ray tracing (RTX), DLSS.
2020–2022	**Ampere**	André-Marie Ampère	2nd-gen RT, 3rd-gen Tensor Cores.
2022–2024	**Ada Lovelace**	Ada Lovelace	DLSS 3.0, enhanced ray tracing.
2022–Present	**Hopper**	Grace Hopper	Designed for AI/HPC: H100 GPU.
2024–2026	**Blackwell**	David Blackwell	HBM3e, NVLink 5.0, massive AI focus.
2026+	**Rubin (Expected)**	Vera Rubin	Upcoming: more AI/ML advancements.



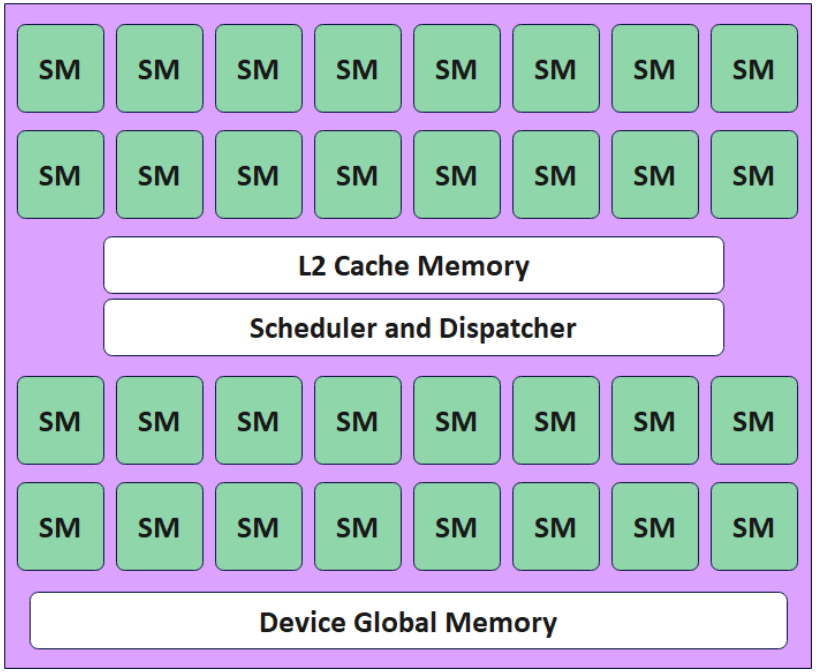
How CPU Uses GPU hardware

- **CPU is the host** → it controls the GPU and launches kernels.
- Data must often be **transferred from CPU memory to GPU memory** before GPU can use it.
- Communication happens through the **PCI Express bus (PCIe)** — not instant!
- After GPU finishes, results may be **copied back to CPU** memory.



GPU Architecture

- **SMs (Streaming Multiprocessors)**: Workhorse units that run threads in parallel
- **Scheduler and Dispatcher**: Controls which thread blocks get sent to which SMs
- **L2 Cache**: Shared across all SMs, acts as buffer between global memory and SMs
- **Device Global Memory**: Main memory on the GPU (like RAM but slower than registers/shared)
- **Each SM has its own local resources**:
  - CUDA cores, Warp schedulers, Registers, Shared memory, Load/store units



Streaming Multiprocessor (SM)

- **SM = core processing unit** of GPU

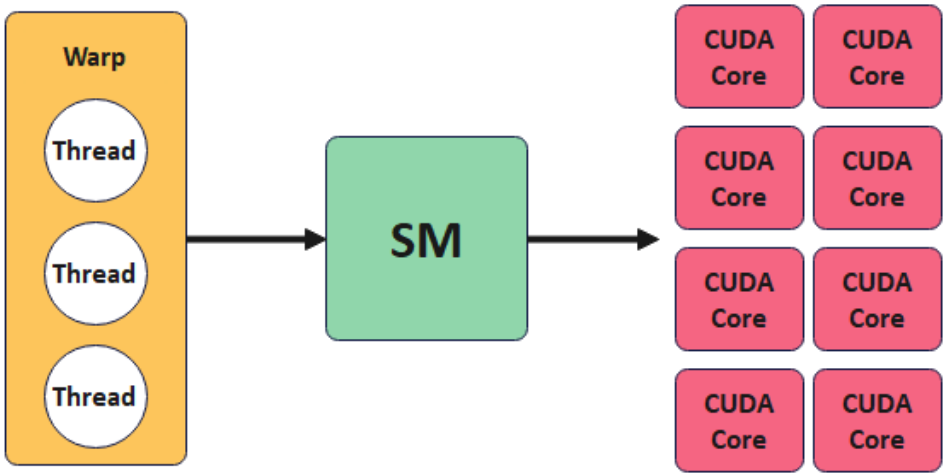
Each SM contains:

- **CUDA cores** (for general ops)
- **Tensor cores** (for matrix math)
- **SFUs** (for special functions)
- **Registers** and **Shared Memory**
- Each GPU may have **tens of SMs**, each capable of running **multiple warps in parallel**

- Think of it as a **"mini GPU within a GPU"**
- Designed for **massive throughput**, not low-latency

CUDA Cores: The Muscle of the SM

- Each **SM contains many CUDA cores**
- CUDA cores = **execution units** for arithmetic and logic operations
- Like **ALUs** in a CPU, but designed for **parallel workloads**
- Execute both **integer and floating-point** operations



How They Work:

- Threads are grouped into **warps** (32 threads per warp)
- A warp is scheduled onto an SM by the **warp scheduler**
- Each CUDA core executes **one thread's instruction per cycle**
- CUDA cores operate in **lockstep** (SIMD-like execution)

CUDA Cores: What They Actually Do

What CUDA Cores Execute:

- **Integer, floating-point, and logical operations**
- **Thread-level execution**: one thread per core, one instruction per cycle
- Common instruction types:
  - Loops & control flow (e.g. if, for, while)
  - Arithmetic: +, -, *, /
  - Memory: load, store, pointer ops
- Each CUDA core is like a **tiny calculator** that runs one part of a thread's job
- They're **general-purpose workers** within the SM — not specialized like SFUs or Tensor Cores
- Cores run instructions in **lockstep** when grouped in a warp.

CUDA: Limitations & Optimizations

CUDA cores work best when:

- **Threads in a warp follow the same execution path**
  Avoid *warp divergence* (e.g. uneven if/else branches)
- **Memory access is coalesced**
  Threads should access **sequential memory addresses**
- **Instructions are simple, repeated, and parallelizable**
  Good: y[i] = x[i] * 2
  Bad: lots of nested logic or varying work per thread

Common Pitfalls:

- **Too many conditionals (if/else)** inside kernels
- **Uncoalesced memory access** (e.g. strided loads)
- **Using printf() inside kernels** → slows execution & pollutes output

Do This Instead:

- **Unroll loops manually or with compiler hints**
- **Use shared memory** when threads need to communicate
- Keep thread workloads **balanced and uniform**

Warp Scheduler: Thread Boss

What It Is:

- Each **SM includes 2–4 warp schedulers**
- Think of it as a **shift manager**: picks which “team” (warp) gets to run next
- Controls warp-level execution — **1 warp = 32 threads**

What It Does:

- **Chooses which warp to execute next**
- Keeps **CUDA cores busy** to avoid idle cycles
- Helps **hide memory latency** (e.g., if one warp waits on memory, it runs another)
- Manages **hundreds or thousands of warps** on standby
- Coordinates **scheduling of instructions** across the SM's resources

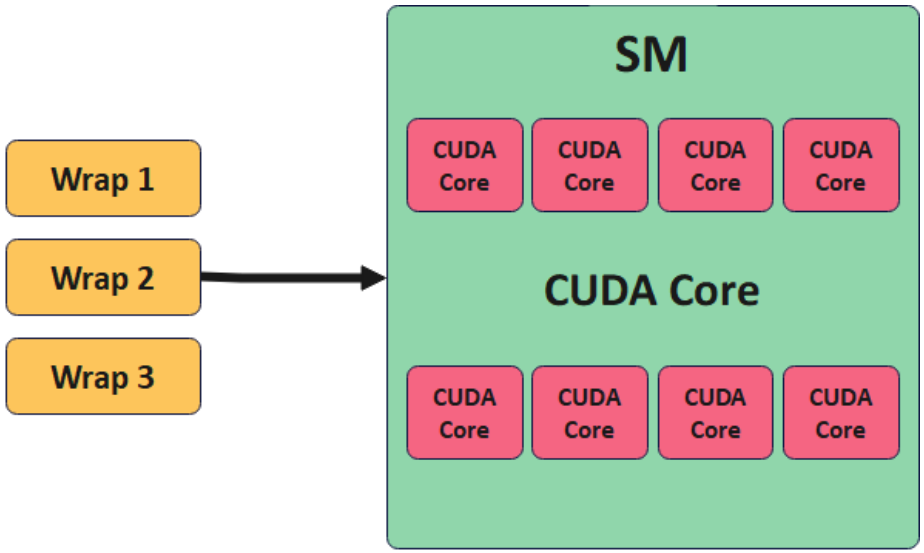
Optimization Tip:

- Good scheduling hides latency, boosts throughput
- Divergence and bad memory access patterns can break scheduler flow

Why Warp Scheduling Matters

What It Enables:

- **Hides memory latency**: switches to a ready warp if one is stalled
- Keeps **CUDA cores busy** with no need for extra hardware
- Enables **concurrent execution** of warps
- Each SM can maintain **many active warps** (e.g., 64+)



Why It's Smart:

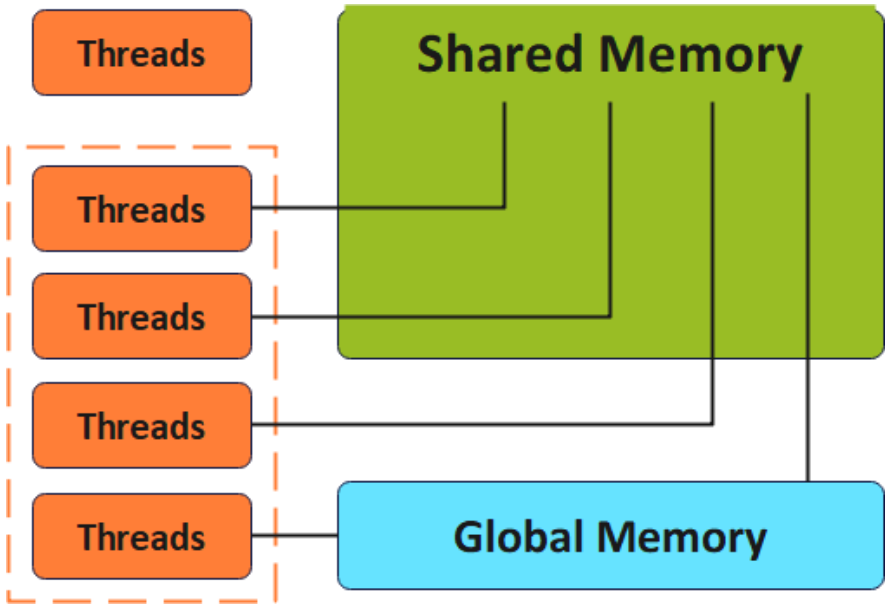
- Boosts efficiency using **smart scheduling** strategies:
  - Round-robin
  - Least recently used
  - Warp-ready prioritization
- Avoids pipeline stalls and keeps **throughput high**

Memory Inside SM – Registers & Shared Memory

Registers (Per Thread):

- **Fastest memory** on the GPU
- Private to each thread → used for temporary variables
- Small size → **compiler decides what goes in**
- Replaced by local memory if register spills occur (bad for perf!)

- **Shared Memory (Per Block):**
- Shared between **all threads** in the same thread block
- Enables **cooperation & fast communication**
- Much faster than global memory
- Manually allocated by programmer (__shared__ in CUDA)



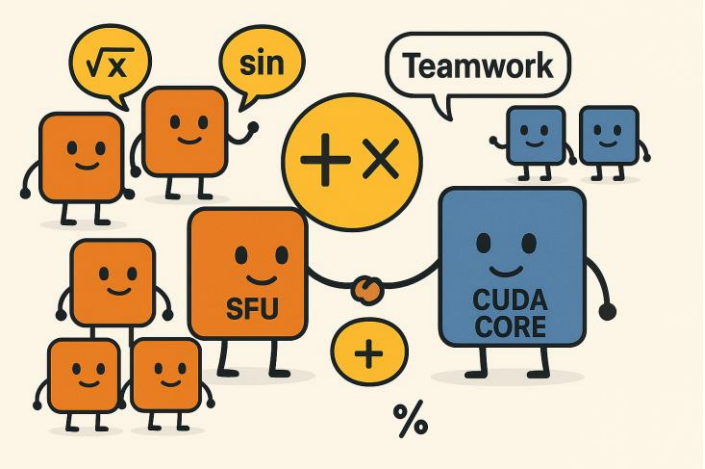
Why Shared Memory Rocks:

- Cuts down on redundant global memory reads
- Supports patterns like **parallel reduction**, **tiling**, **stencil ops**
- Boosts performance by keeping data close to the threads

SFUs: Math Ninjas

What Are SFUs?

- **Special Function Units** – dedicated hardware inside each SM
- Handle **non-basic math ops** like:
  - sin(), cos(), sqrt(), rsqrt(), exp(), log(), etc.
- **Optimized for transcendental functions**



Why Use SFUs?

- **Much faster** than using CUDA cores for the same operations
- Offloads **complex calculations**, freeing CUDA cores for general ops
- CUDA cores = the main team;
  SFUs = math wizards that jump in for tricky stuff

How to Use SFUs Effectively

SFUs Work Automatically:

- No need to manually schedule them — just use functions like sin(), cos(), sqrt()
- CUDA cores will **auto-offload** to SFUs behind the scenes

What to Watch Out For:

- SFUs are **shared across all threads in an SM**
- Limited count per SM (e.g. 4–8 per SM), so...
- **Too many threads using SFUs = bottleneck**

Optimization Tips:

- Try to **spread out** special math ops across time or threads
- Avoid putting heavy SFU usage **in tight inner loops**
- Profile your kernels — if you see slowdown on sin(), sqrt(), etc., **SFU contention** might be the cause

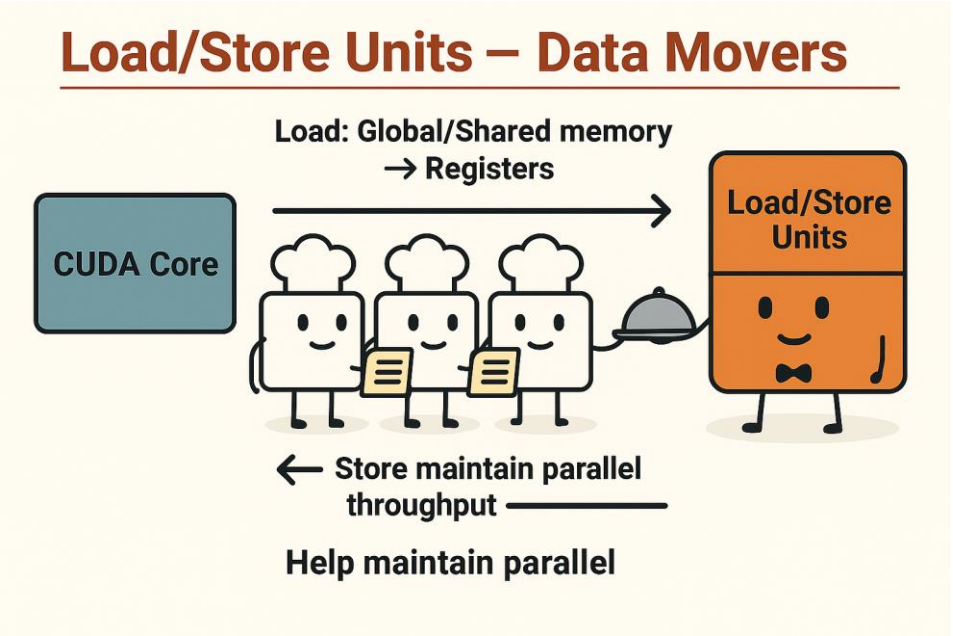
Load/Store Units – Data Movers of the SM

What They Do:

- **Move data between memory and CUDA cores**
- Load: Global/Shared memory → **Registers**
- Store: Registers → **Global/Shared memory**

How They Work:

- LSUs operate **behind the scenes**, managed by hardware + compiler
- **CUDA cores can't access memory directly** — they rely on LSUs to move data
- They're like the **invisible delivery trucks** that keep the math factory running



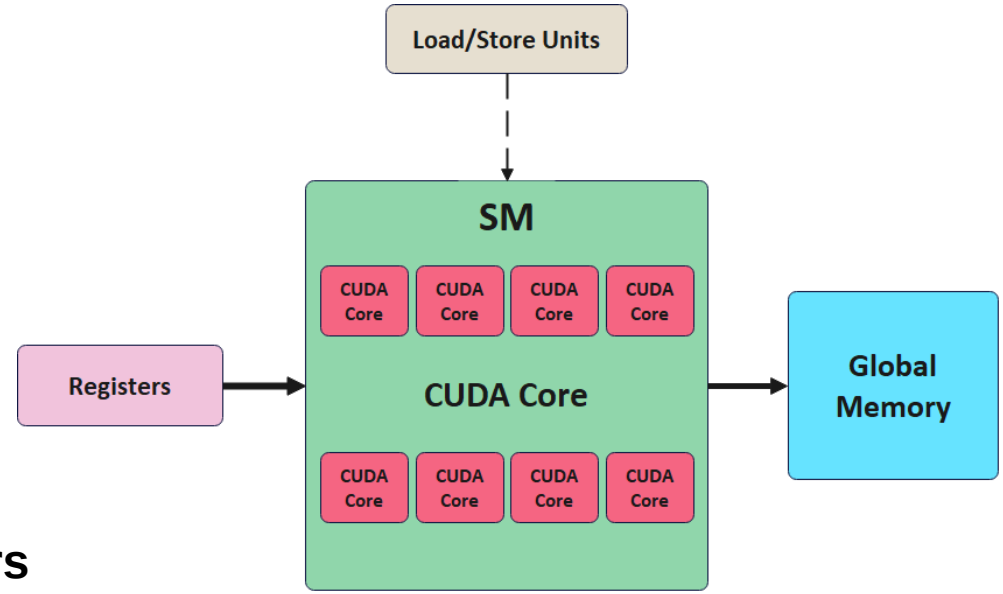
Why They Matter:

- **Efficient memory movement = better performance**
- Good LSU usage **prevents CUDA core stalls**
- Help maintain **parallel throughput**

Load/Store Units – Data Movers of the SM

Avoid Memory Bottlenecks by:

- **Coalescing global memory accesses** (threads should access memory in order, no weird jumps)
- **Minimizing loads/stores inside loops** (especially nested ones)
- **Using shared memory** or **registers** (avoid unnecessary global memory trips)
- **Avoiding bank conflicts** in shared memory
- **Uncoalesced global memory** = traffic jam



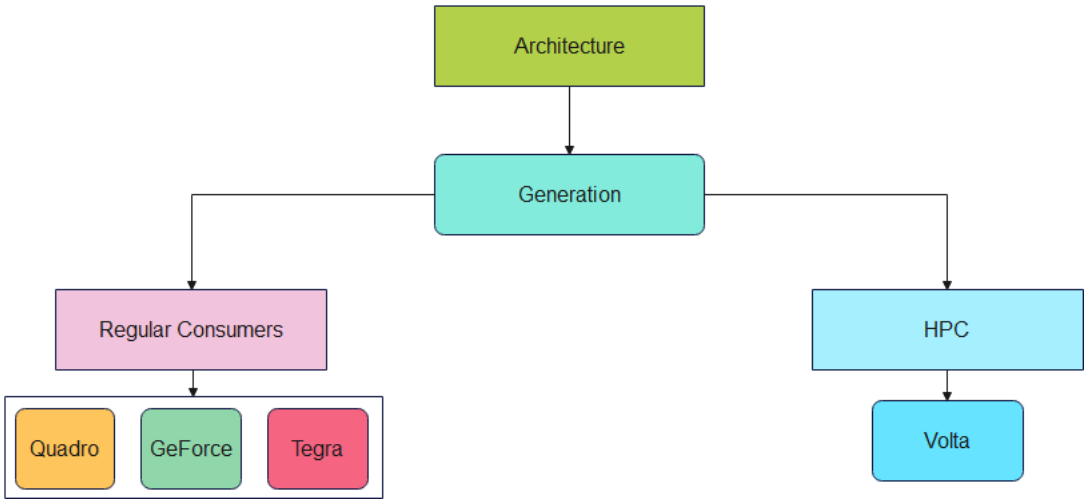
Two Main GPU Categories

- **Standard GPUs** focus on **graphics**, gaming, and productivity.
- **HPC GPUs** are optimized for **parallel computation**, AI workloads, scientific computing.
- Example applications for HPC: Used by **Meta**, **Amazon**, **Google**, **Microsoft**, etc.

Category	Designed For	Product Families (Generations)	Example GPUs
Standard GPUs	Everyday users: laptops, desktops, workstations, gaming setups	**GeForce**, **Tegra**, **Quadro**	RTX 4090, RTX 3090, RTX 4060, GTX 1080
HPC GPUs	High-Performance Computing: AI training, datacenters, cloud	**Tesla**, **A-Series**, **H-Series**	H100, A100, V100, A40, T4

Types of NVIDIA GPUs and Target Markets

GPU Line	Details	Market Examples
Tegra	Mobile GPUs for smartphones, tablets, embedded systems	Nintendo Switch, Microsoft Zune
GeForce	Gaming and consumer graphics, video editing	RTX 3090, gaming PCs
Quadro (now RTX A-series)	Professional cards for design, engineering, animation	RTX A4000, RTX A6000, HP Z4 G4
Tesla (merged into Data Center GPUs)	High-performance computing, scientific simulations	A100, data centers



What are Tensor Cores?

CUDA Cores vs Tensor Cores:

- **CUDA Cores** = general-purpose math
- **Tensor Cores** = built for **matrix multiplications** (e.g. deep learning)

Why Tensor Cores Exist:

- Designed for **AI/ML workloads** (especially DNN training/inference)
- Introduced in **NVIDIA Volta** and later architectures
- **Massively accelerate** matrix-multiply-accumulate (MMA) ops (key to neural nets!)

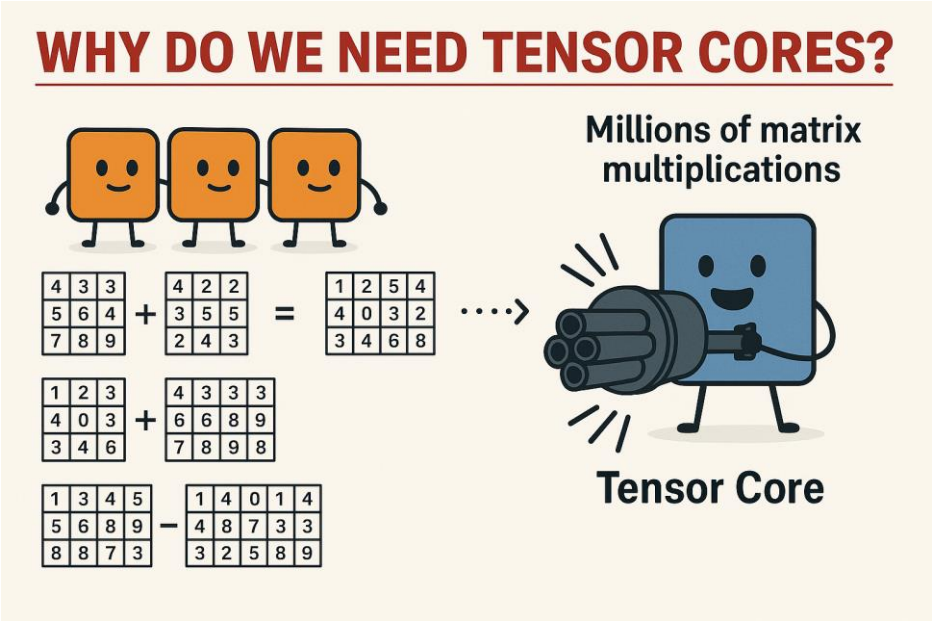
Tensor Cores Work With:

- Reduced-precision formats:
  - FP16, BF16, INT8, TF32
- Trade precision for **speed + throughput**

Why Do We Need Tensor Cores?

Problem:

- Machine Learning (ML) & Deep Learning (DL) = **millions of matrix multiplications**
- Example: 4x4 × 4x4 matrix multiply looks small…
  But modern networks = **tens of thousands** of these per second



Multiply–Accumulate Overload:

- Tensor cores do **multiple MAC (Multiply-Accumulate) ops per clock**
- Built for exactly this kind of work — **no wasted effort**

Insane Throughput:

- NVIDIA **Ampere** Tensor Cores can hit **1024 operations per clock per core**

How Tensor Cores Work

Operation:

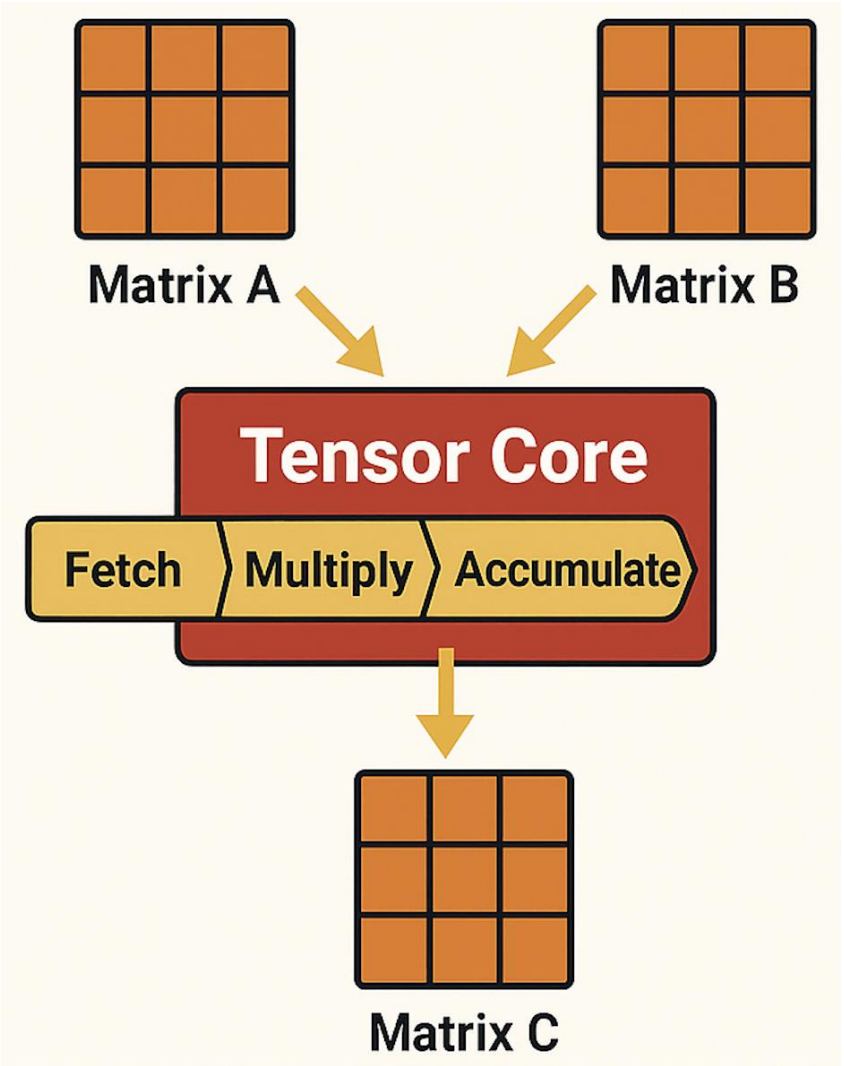
- Tensor Cores perform: **Matrix A × Matrix B + Matrix C → Result Matrix**
- This is called a **FMA (Fused Multiply-Add)**

Tensor Core Pipeline:

- **Fetch** operands (Matrix A, B, C)
- **Multiply** A × B
- **Accumulate** result into C
- **Write Back** to memory

Why It's Efficient:

- FMA does **2 operations in 1 instruction** (multiply + add)
- No intermediate memory reads/writes = **blazing fast**



How Tensor Cores Work (Example)

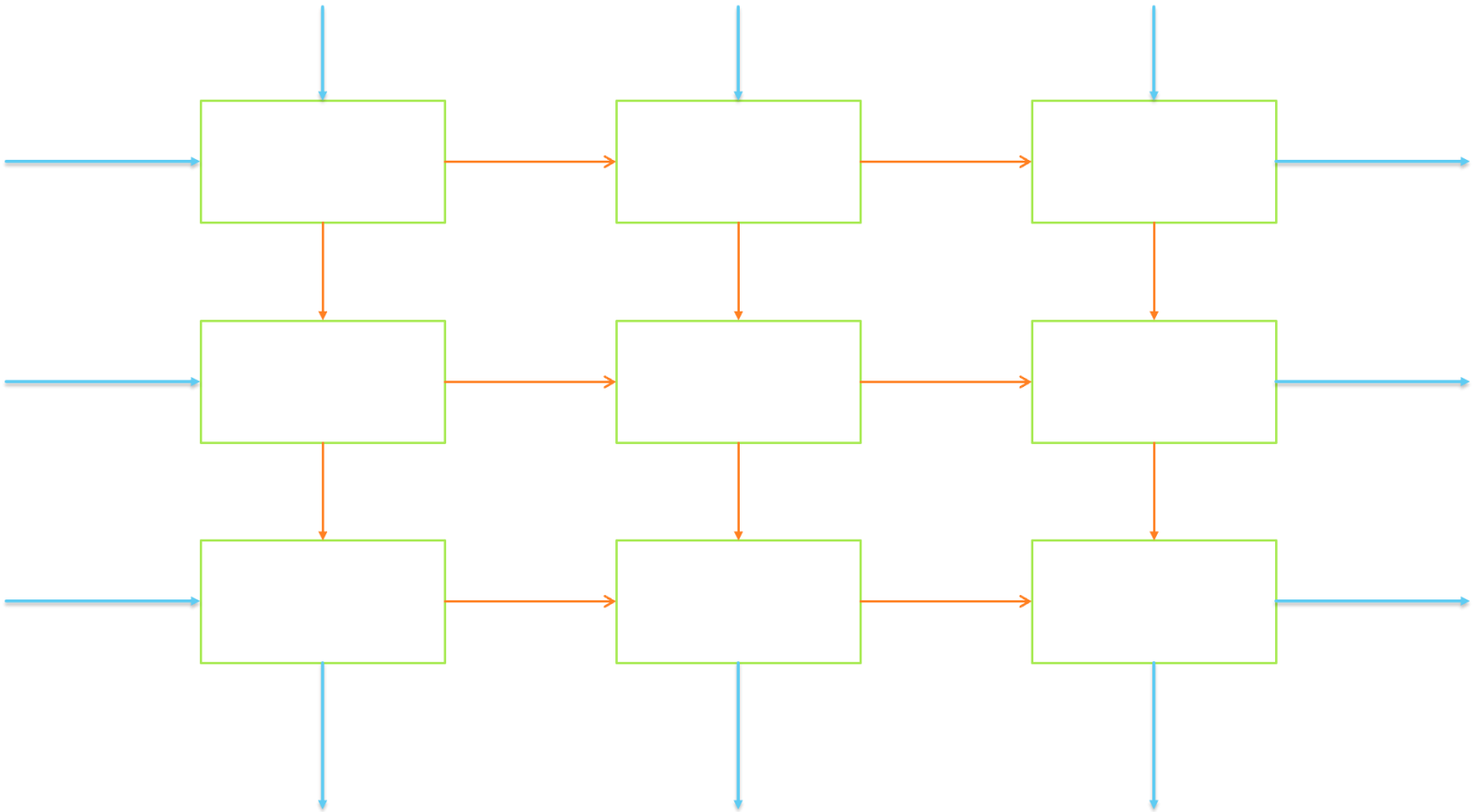
3x3 Matrix Multiplication

$$A = \begin{matrix} a00 & a01 & a02 \\ a10 & a11 & a12 \\ a20 & a21 & a22 \end{matrix}$$

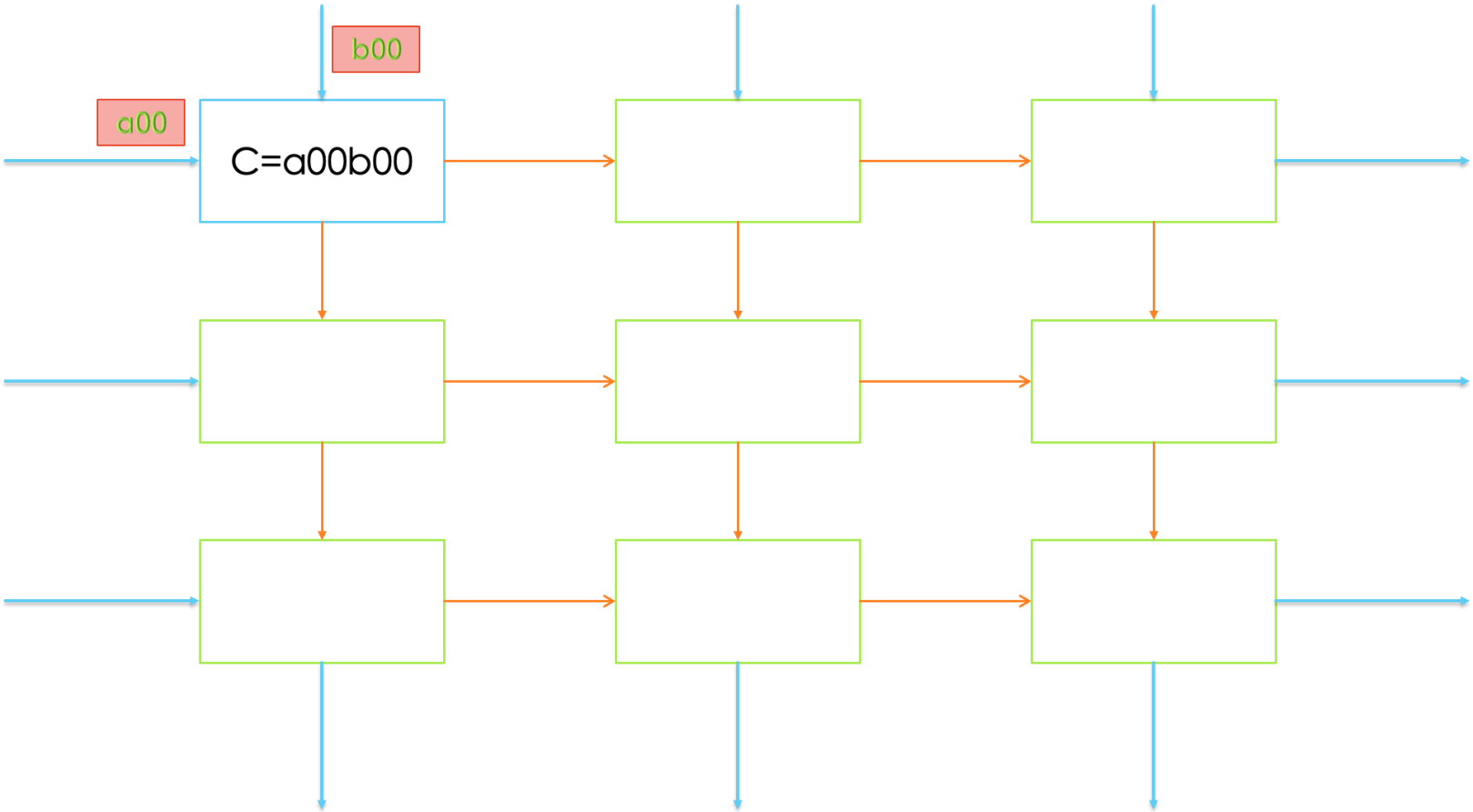
$$B = \begin{matrix} b00 & b01 & b02 \\ b10 & b11 & b12 \\ b20 & b21 & b22 \end{matrix}$$

$$C = \begin{matrix} a00b00 + a01b10 + a02b20 & a00b01 + a01b11 + a02b21 & a00b02 + a01b12 + a02b22 \\ a10b00 + a11b10 + a12b20 & a10b01 + a11b11 + a12b21 & a10b02 + a11b12 + a12b22 \\ a20b00 + a21b10 + a22b20 & a20b01 + a21b11 + a22b21 & a20b02 + a21b12 + a22b22 \end{matrix}$$

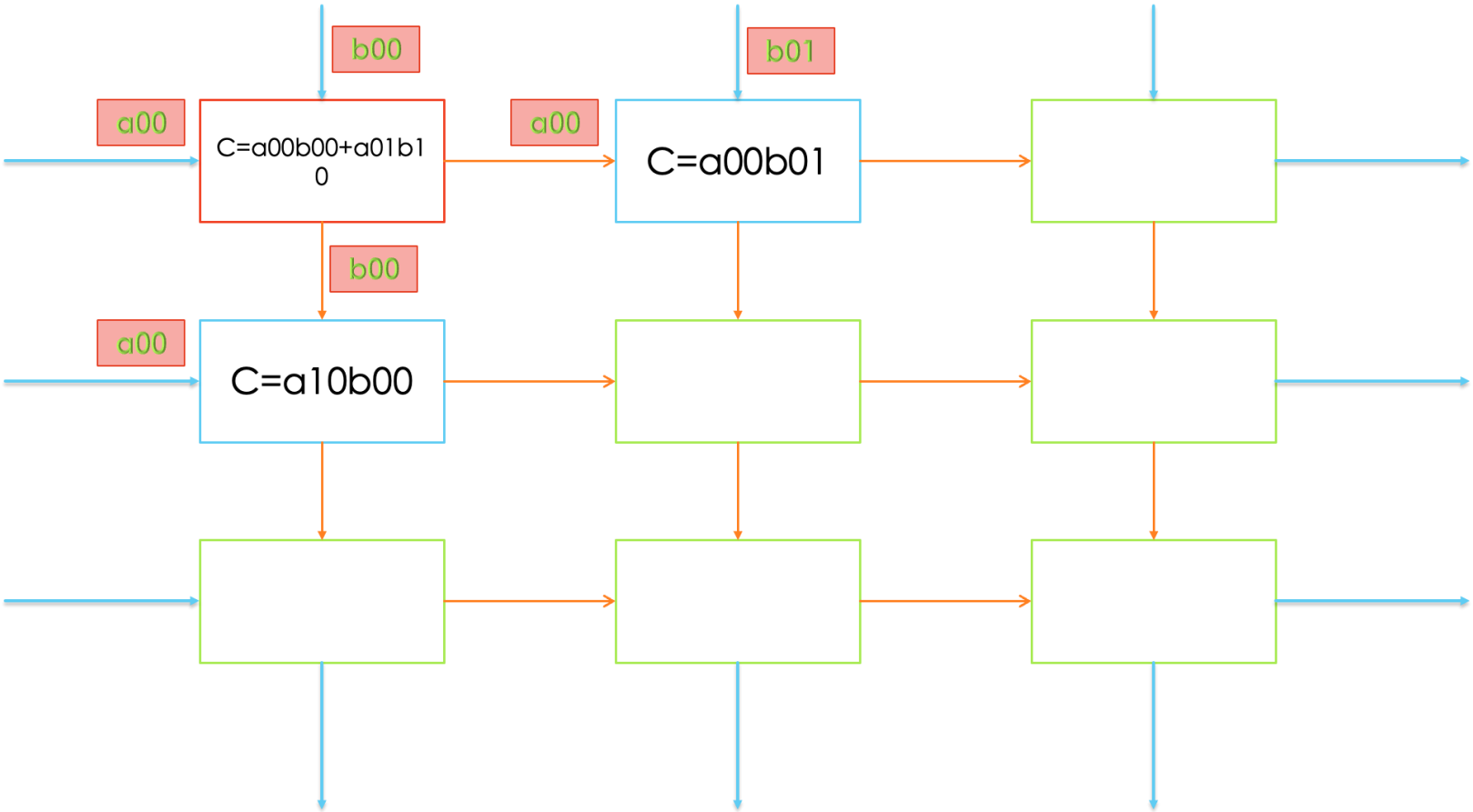
How Tensor Cores Work (Example)



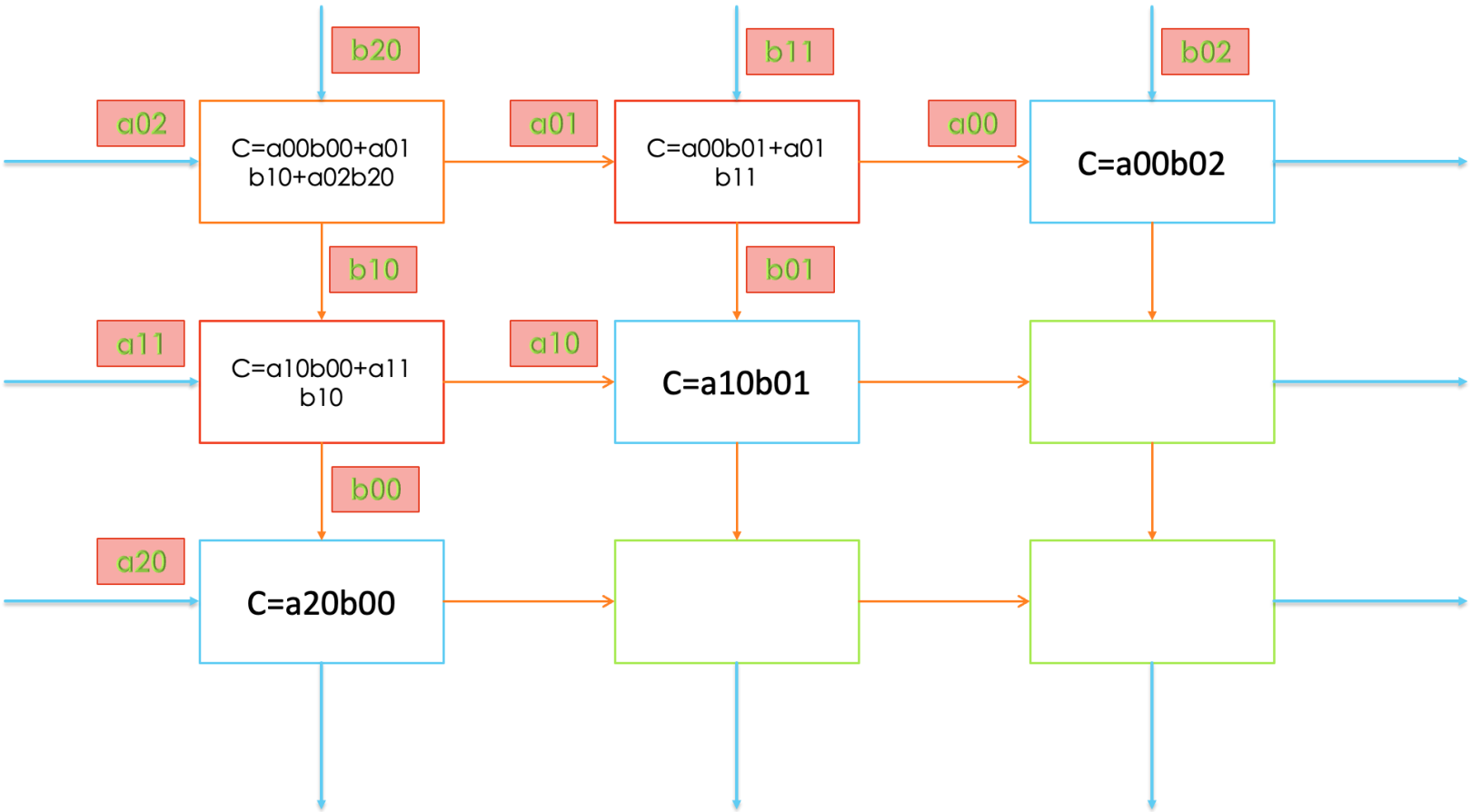
How Tensor Cores Work (Example)



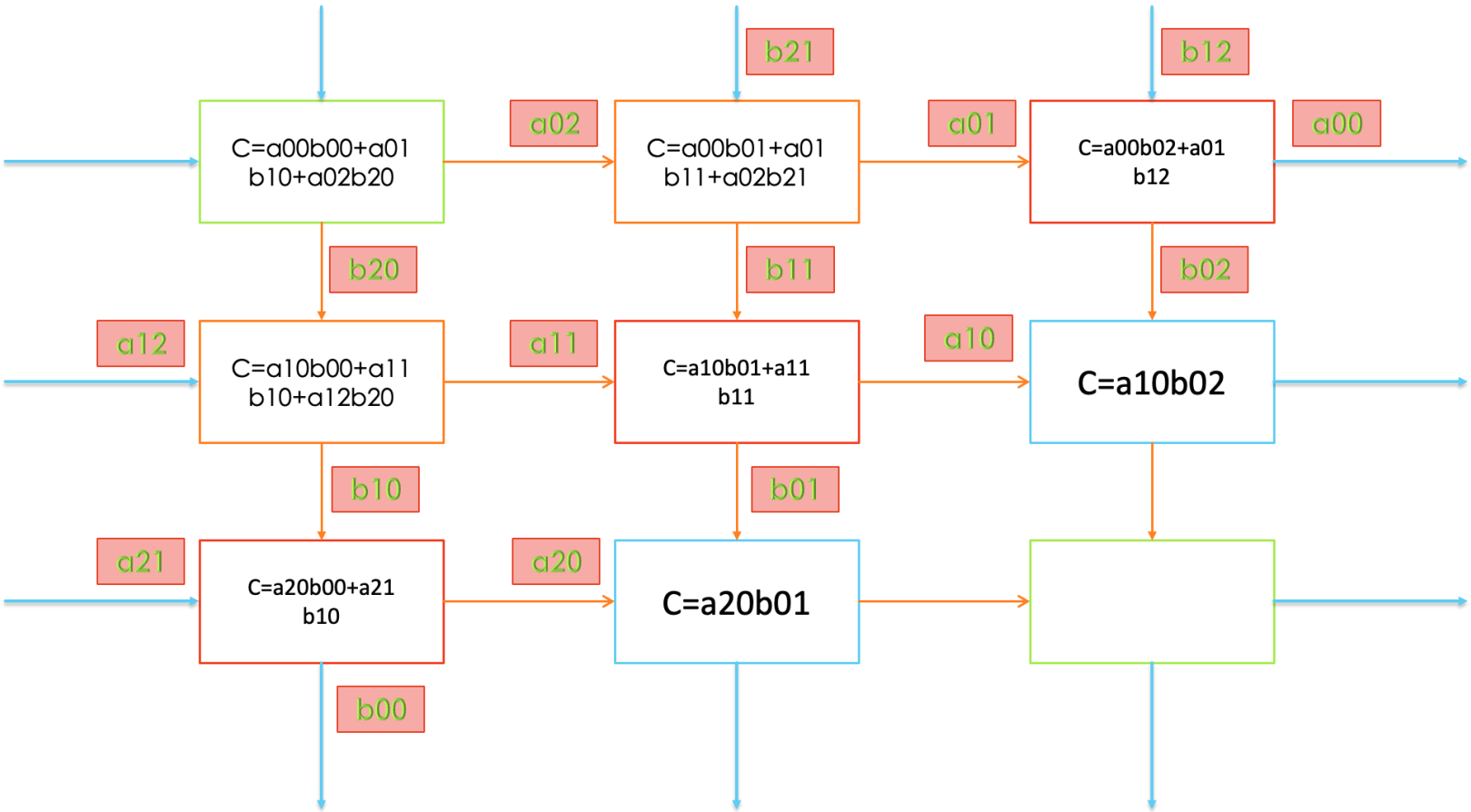
How Tensor Cores Work (Example)



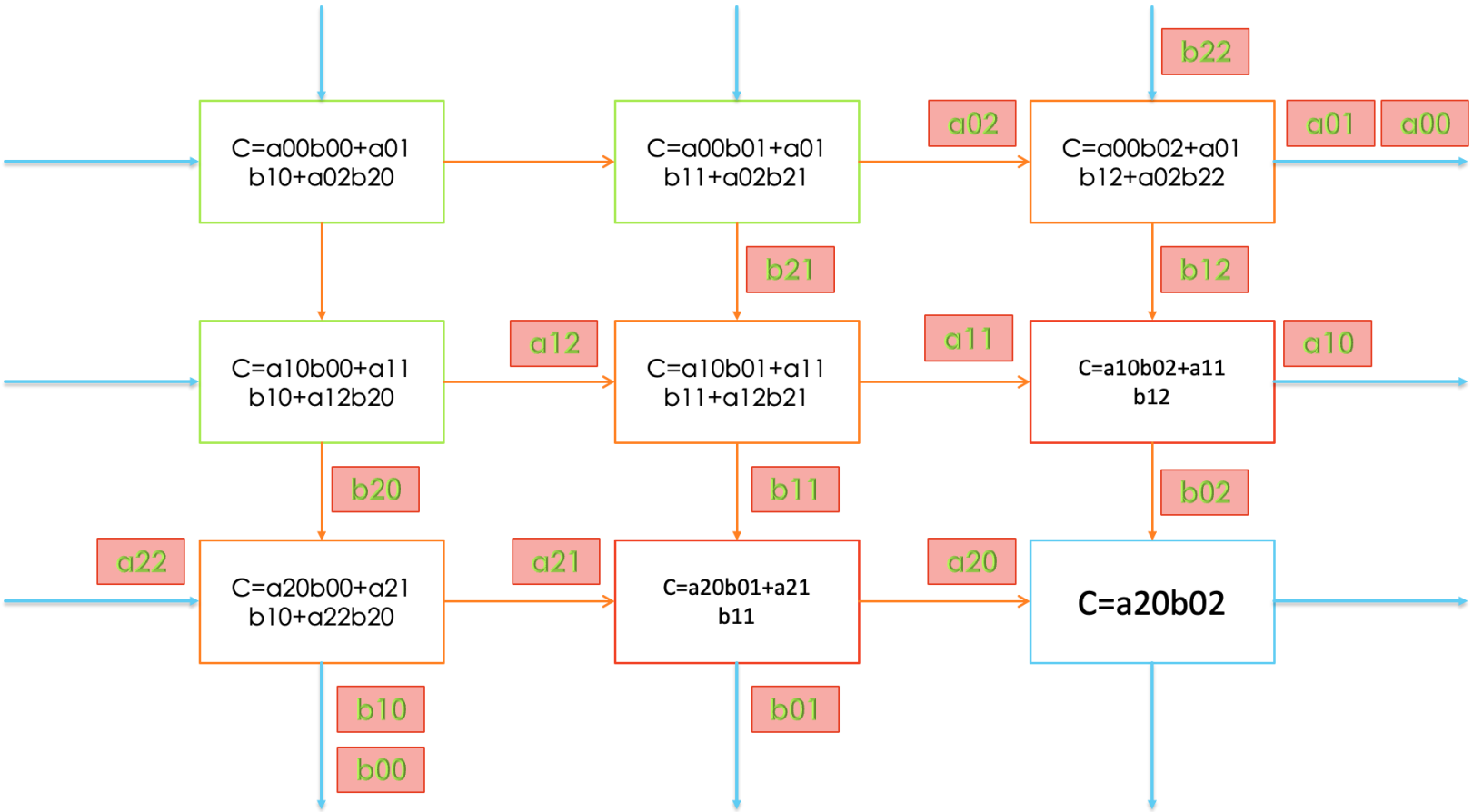
How Tensor Cores Work (Example)



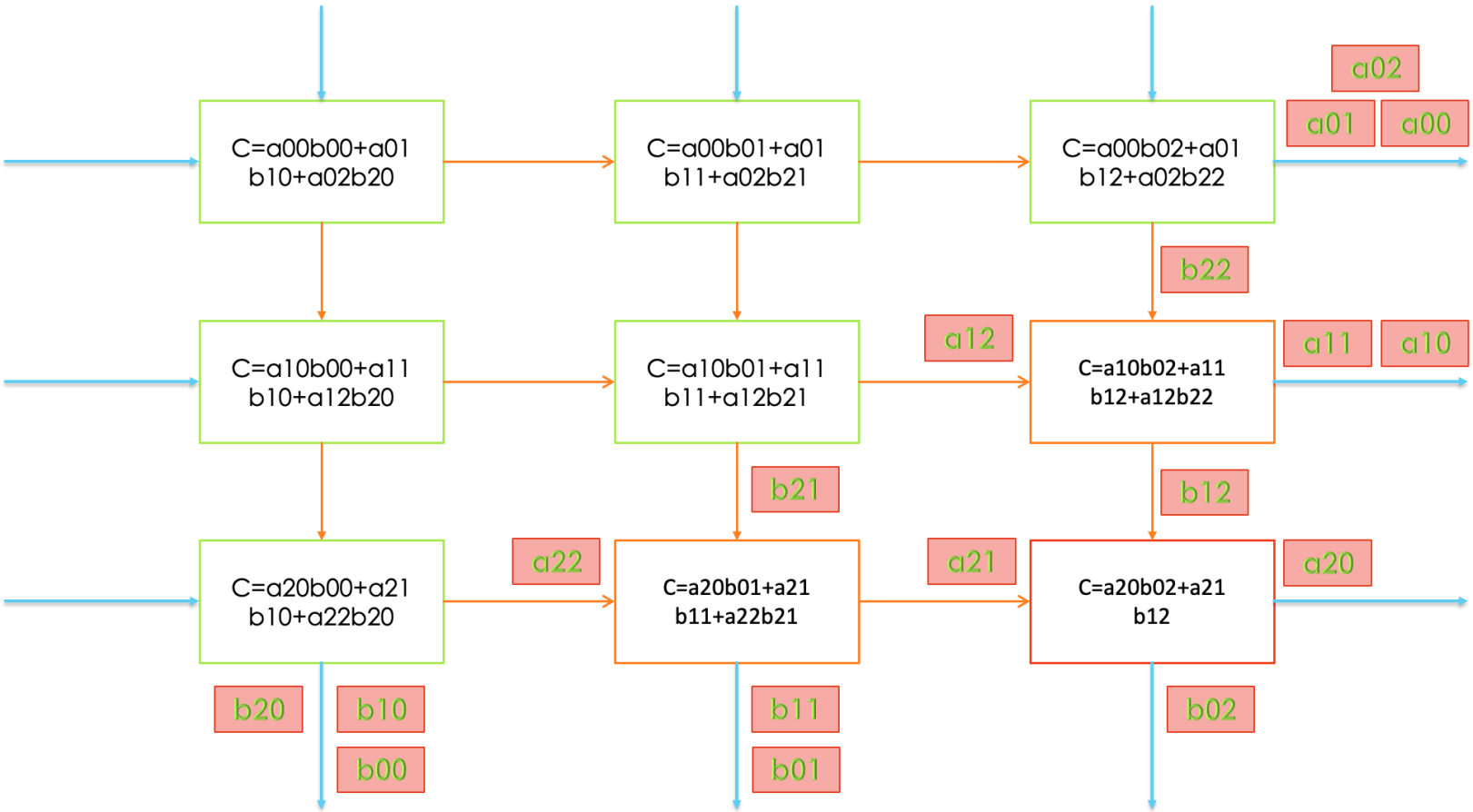
How Tensor Cores Work (Example)



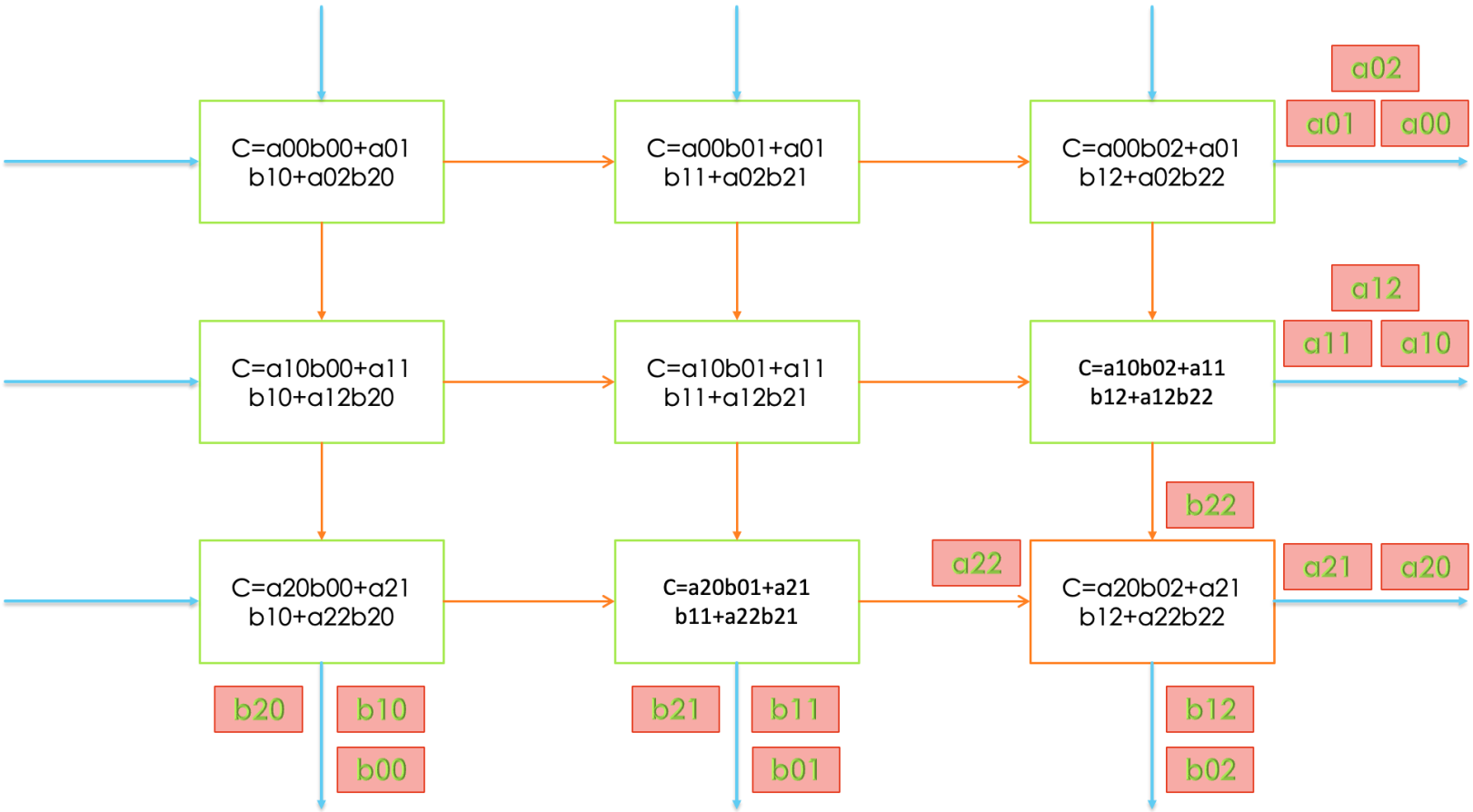
How Tensor Cores Work (Example)



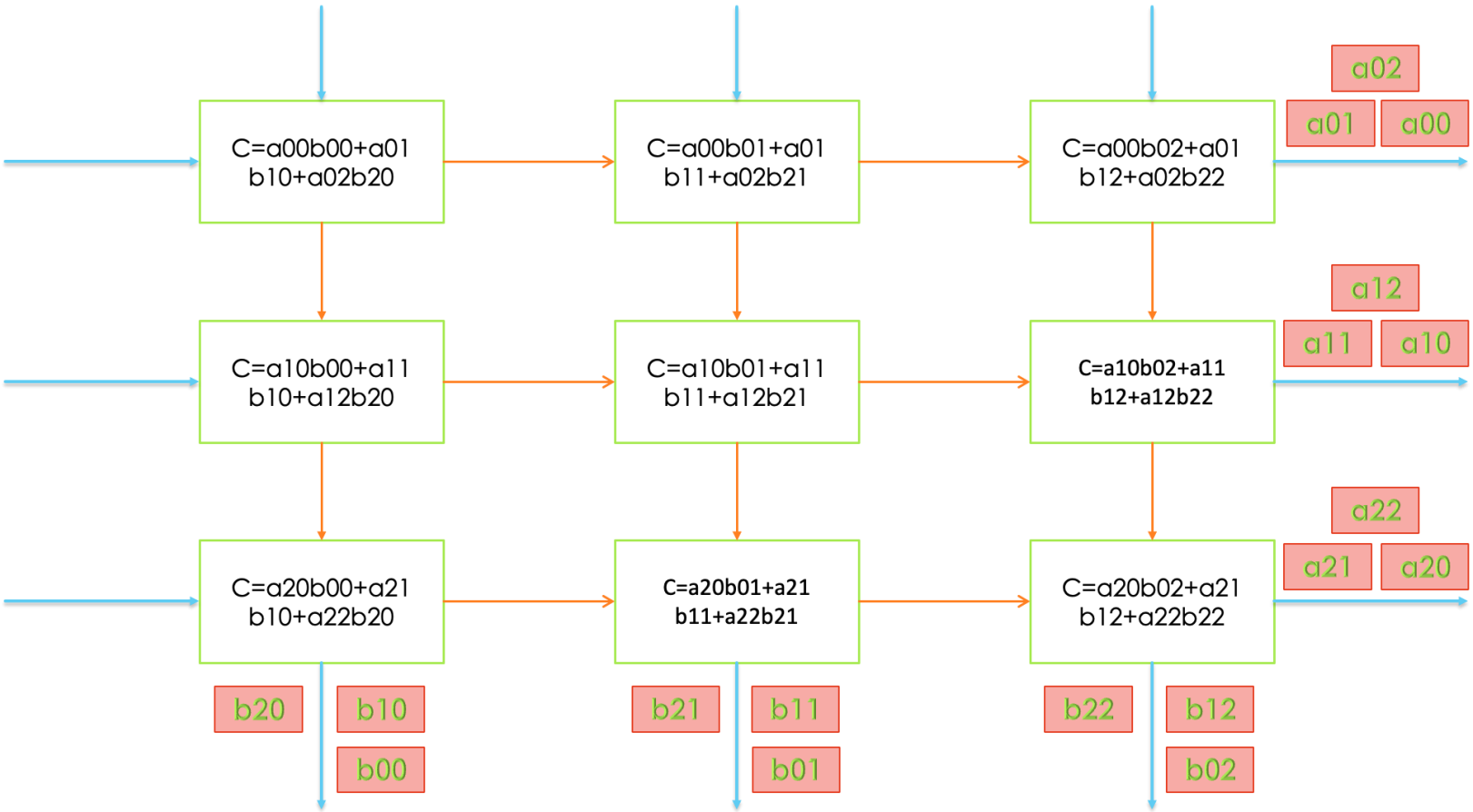
How Tensor Cores Work (Example)



How Tensor Cores Work (Example)



How Tensor Cores Work (Example)



Tensor Cores vs CUDA Cores: Who Does What?

Tensor Cores:

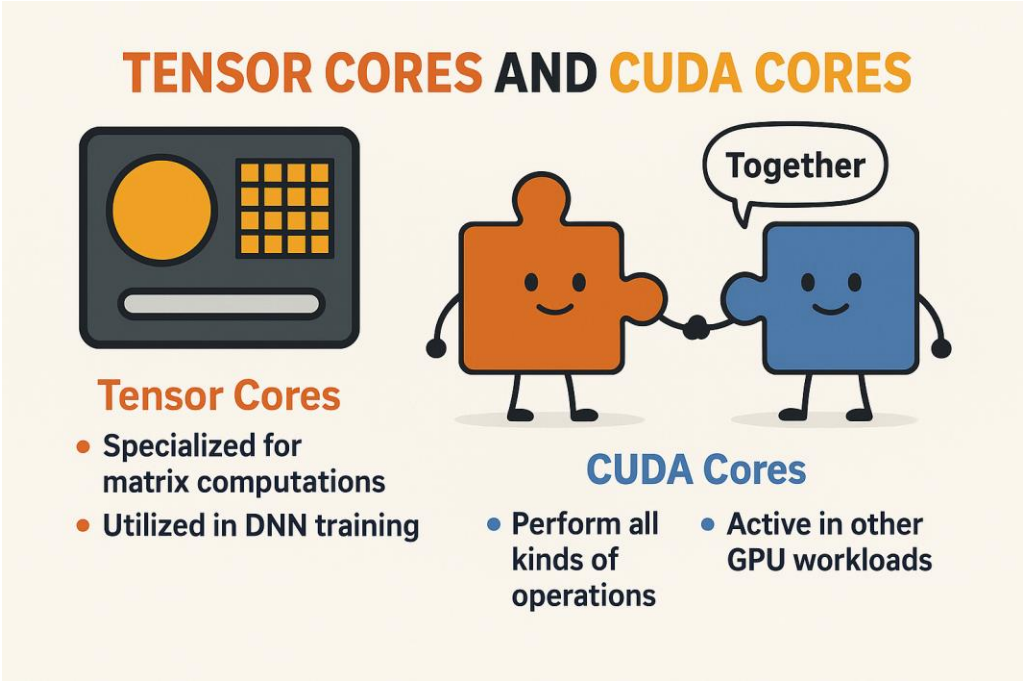
- Built for **ultra-fast matrix math**
- Ideal for **deep learning ops** (training/inference)
- Operate on **FP16, BF16, INT8**
- Used in **mixed precision training** (automatically)

CUDA Cores:

- General-purpose logic units
- Handle loops, control flow, logic ops
- Do **memory ops**, activation functions, tensor prep
- Stay active even when tensor cores are busy

Together:

- **Teamwork = max performance**
- Tensor cores = speed
- CUDA cores = brains & support



FLOPS (Floating Point Operations Per Second)

FLOPS= Number of Cores × Clock Speed (Hz) × Operations per Cycle

Typically:

- **FP32** (single-precision): 12.12
- **FP64** (double-precision): 12.1212
- **Tensor Cores** (for FP16, BF16, TF32): Use separate matrix-multiplication units

FP32 Performance

- 6,912×1.41×109×2=19.5 TFLOPS

FP64 Performance

- 6,912×1.41×109×1=9.75 TFLOPS

Tensor Cores (FP16 / BF16)

- **Without Sparsity:**
  - 19.5×16=312 TFLOPS (FP16 dense)
- **With Sparsity (2x gain):**
  - 312×2=624 TFLOPS

INT8 (used in inference)

- Reported up to 624 TOPS (TeraOps per second) with sparsity

NVIDIA A100 Specs (Ampere)

Precision	Performance (Theoretical)	Notes
FP32	~19.5 TFLOPS	General compute
FP64	~9.75 TFLOPS	Scientific compute
TF32	156 / 312 TFLOPS	Deep learning
FP16	312 / 624 TFLOPS	Tensor ops
BF16	~same as FP16	Less precision
INT8	~624 TOPS	Inference tasks

Feature	Value
CUDA Cores	6,912
SM Count	108
Clock Speed	~1.41 GHz
FP64 Cores	1/2 of FP32 (A100 uses native FP64 on CUDA cores)
Tensor Core (FP16/BF16/TF32)	432
Ops/Cycle (FP32, FP64)	2
Tensor Ops (FP16)	512 FP16 FMA ops per core per cycle

What You Learned Today?

- **GPUs are built for parallel processing.**
  Way more cores than CPUs, perfect for AI, graphics, and simulation.

- **CUDA Cores** = the "worker bees" that run your code
  They do logic, math, loops, memory ops.

- **SM (Streaming Multiprocessor)** = mini multi-core CPU inside GPU
  Each SM contains CUDA cores, SFUs, Tensor cores, schedulers, etc.

- **Warps** = groups of 32 threads managed by warp schedulers
  Schedulers keep the cores busy and hide memory latency.

- **SFUs** = special units for math ops like sin(), cos(), sqrt()
  They handle tricky calculations quickly.

- **Tensor Cores** = matrix math monsters
  Do thousands of operations per cycle, key for deep learning.

- **LSUs** = memory movers (delivery trucks)
  Move data between memory and registers/cores.

Thanks for Your Attention!