GPU Architecture for Machine Learning

Workload is the king, as always

Li Shang
lishang@slai.edu.cn

GPGPU was born differently

- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Avoid high frequency clock speed, due to power bound.

Modern GPGPU

Objective：Flexible, programming graphics and high-performance computing

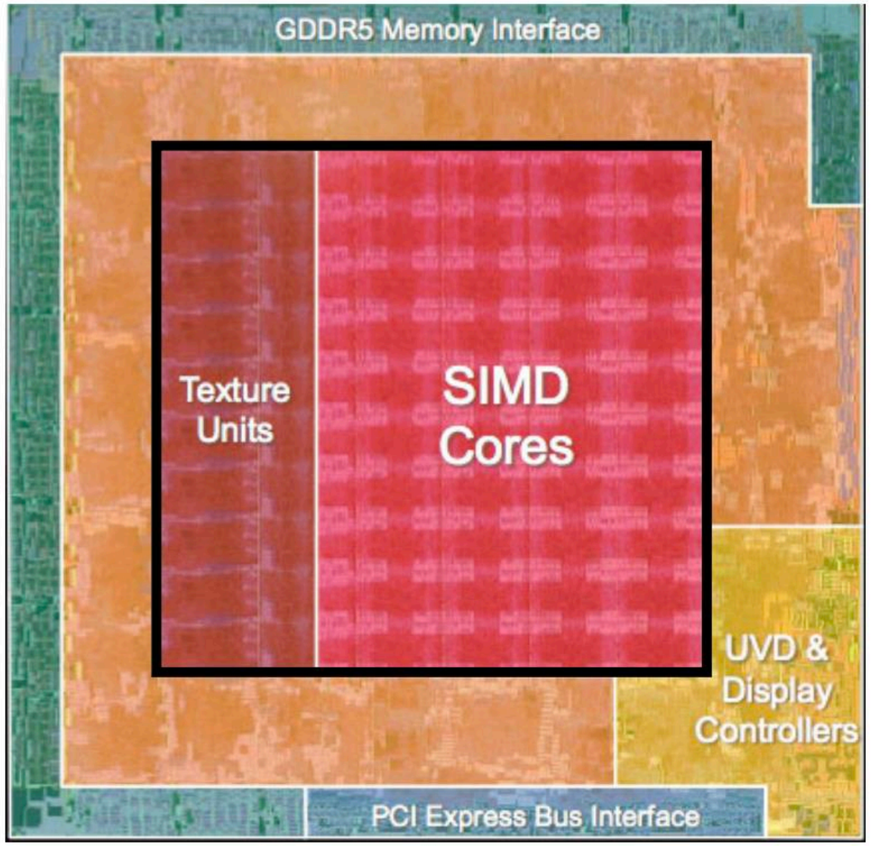
Architecture：Unified graphics and parallel computing architecture

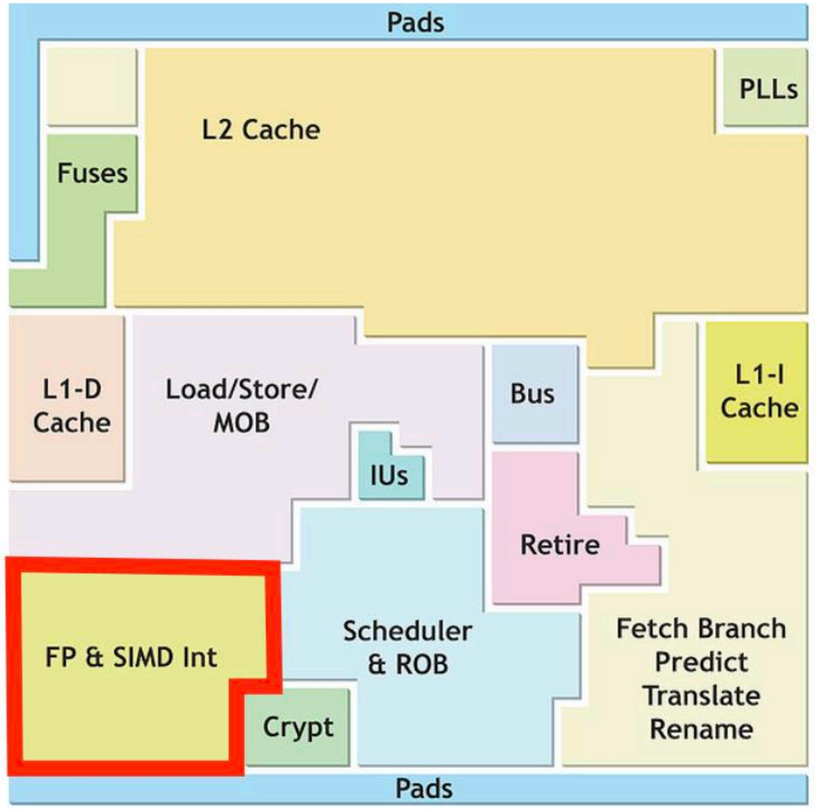
Scalability：Parallel array of processors are massively multithreaded

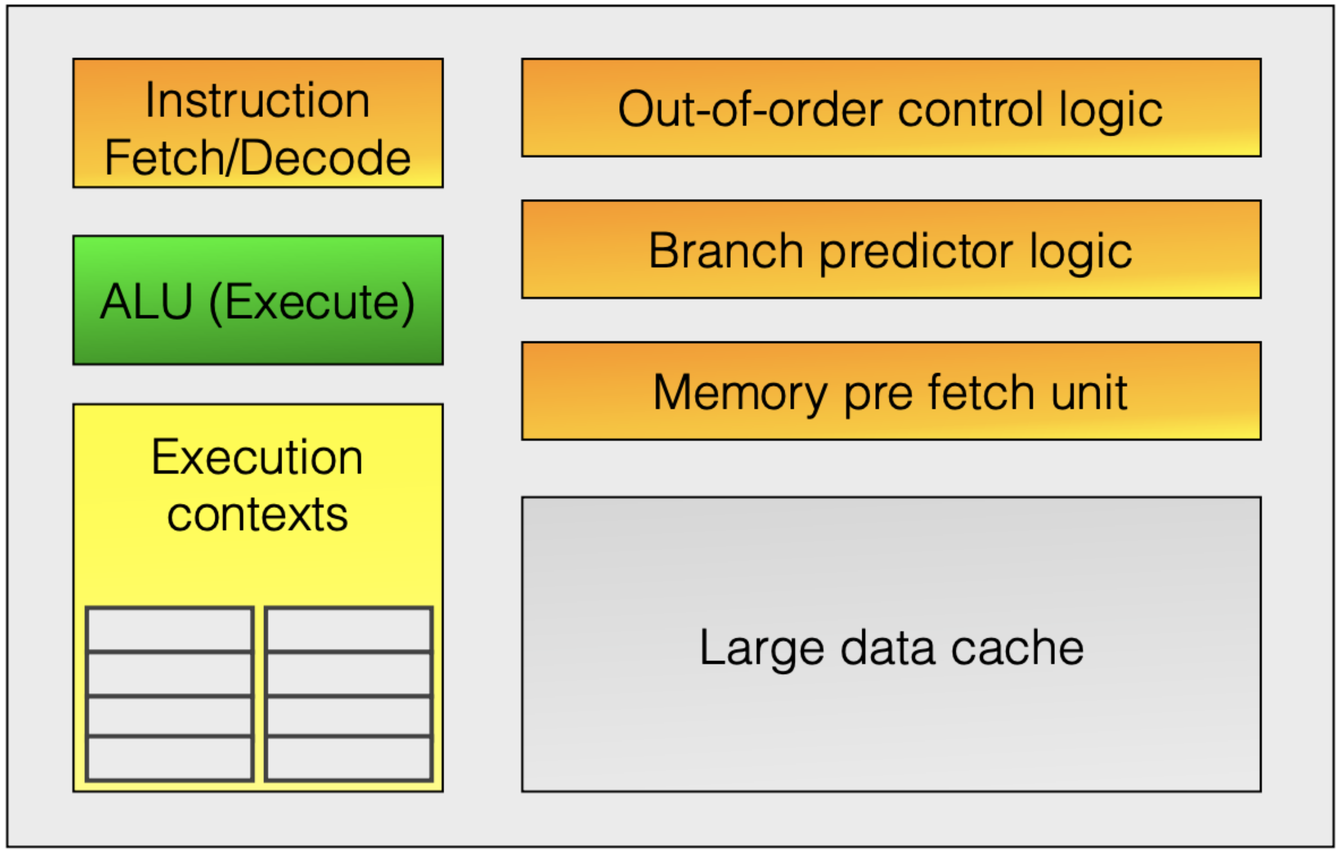
Programming Flexibility：CUDA programming model for high-performance GPGPU programming

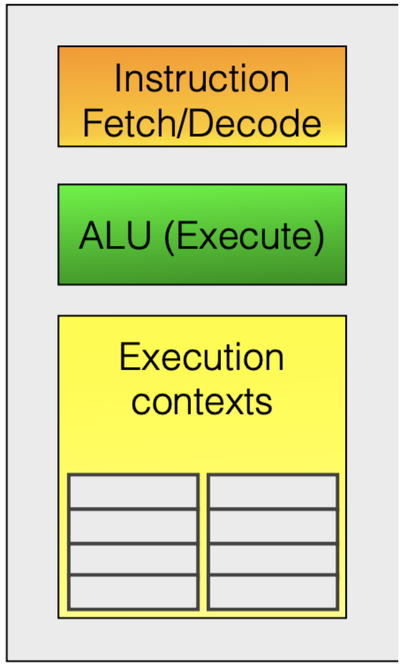
Eric Lindholm, et al., “Nvidia Tesla: A unified graphics and computing architecture, IEEE Micro, 200

CPU vs. GPU

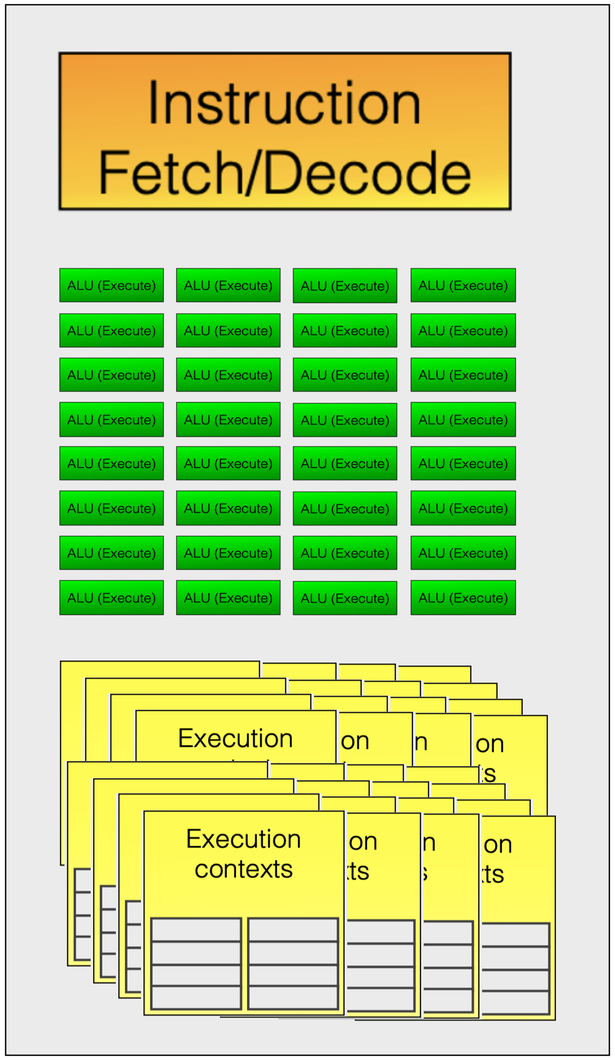

GDDR5 Memory Interface
Texture Units
SIMD Cores
UVD & Display Controllers
PCI Express Bus Interface


Pads
PLLs
L2 Cache
Fuses
L1-D Cache
Load/Store/ MOB
Bus
L1-I Cache
IUs
Retire
FP & SIMD Int
Scheduler & ROB
Fetch Branch Predict Translate Rename
Crypt
Pads


Instruction Fetch/Decode
ALU (Execute)
Execution contexts
Out-of-order control logic
Branch predictor logic
Memory pre fetch unit
Large data cache


Instruction
Fetch/Decode
ALU (Execute)
Execution
contexts


Instruction Fetch/Decode
ALU (Execute)
Execution contexts


Instruction Fetch/Decode
ALU (Execute)
Execution contexts


Instruction Fetch/Decode
ALU (Execute)
Execution contexts

GPU Architecture

- GPGPU: From computer graphics to deep learning
- SIMT: single instruction multiple thread computing
- Memory hierarchy: Global/shared/register storage
- Advanced GPU Features and Future Directions

GPU: Three Generations

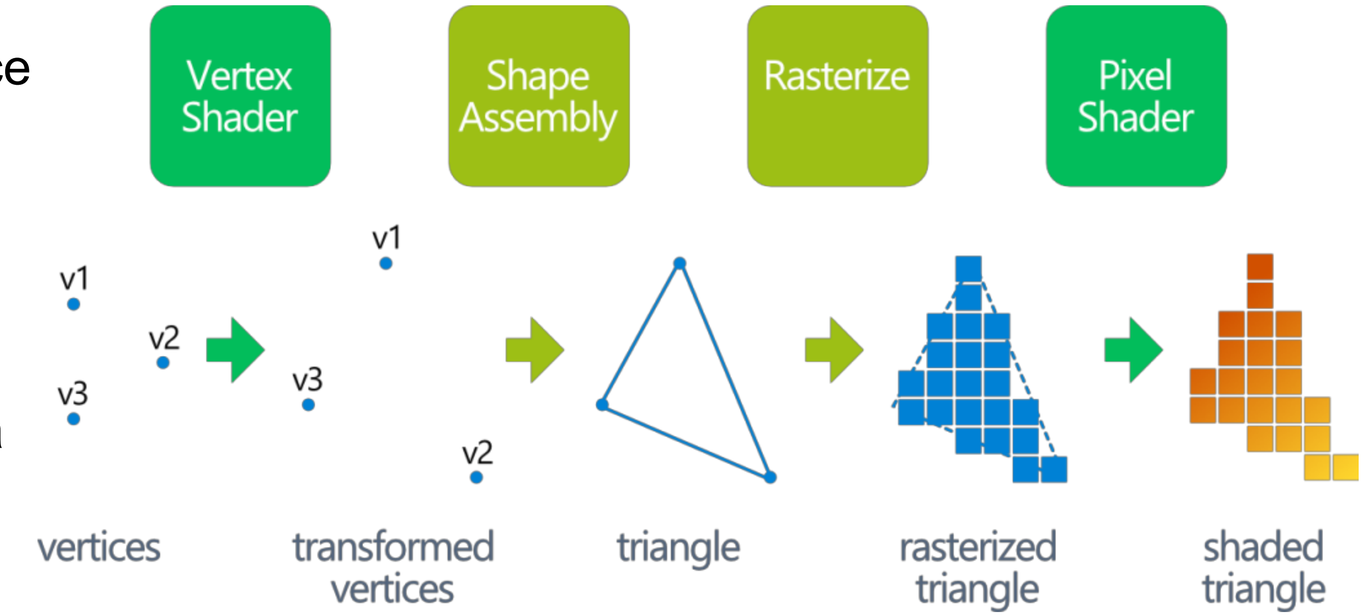
- GPU evolution: From fixed-function accelerator to programmable processor.
- Early adoption in high-performance learning: The massive parallel nature suitable for linear algebra.
- GPU in the deep learning : Architecture evolution driven by rapid growth of deep learning.

Phase	Era	Key Features	Examples
1️⃣ **Hardwired GPU (Fixed-Function Graphics)**	**1980s – Early 2000s**	- Specialized **fixed-function pipelines** for rendering - No programmability, only hardware-based transformations & rasterization	- **1981:** IBM CGA (First consumer graphics card) - **1999:** NVIDIA GeForce 256 (First GPU)
2️⃣ **Programmable GPU (Shader-Based Graphics Processing)**	**Early 2000s – 2010s**	- **Vertex & Pixel Shaders** introduced for **custom effects** - Unified Shader Architecture allows **software-defined rendering**	- **2001:** NVIDIA GeForce 3 (First programmable vertex shader) - **2006:** NVIDIA Tesla (First Unified Shader)
3️⃣ **GPGPU for AI/ML (General-Purpose GPU Computing)**	**2010s – Present**	- **CUDA/OpenCL** enable **parallel computing** for AI, HPC - **Tensor Cores & AI Accelerators** introduced	- **2007:** NVIDIA CUDA (GPGPU programming) - **2017:** NVIDIA Volta (First Tensor Cores for ML)

GPU is a Massive Shader

The graphics rendering pipeline consists of several stages, each responsible for specific tasks in transforming 3D models into 2D images.

- Vertex Transformation Stage: Transforms individual triangle vertices from 3D world space to 2D screen space.
- Shape Assembly Stage: A fixed-function stage that assembles transformed vertices into geometric shapes, typically triangles.
- Rasterization Stage: Another fixed-function stage that converts assembled triangles into a set of pixels or fragments.
- Pixel Shader Stage: Determines the color of each pixel. Programmed using a pixel shader (also known as a fragment shader in OpenGL and Metal), which is executed once per pixel.



Programmable GPU (GeForce 7800@2005)

Vertex Shader Engine: Handles per-vertex computations such as geometry transformations, vertex lighting, and vertex displacement.

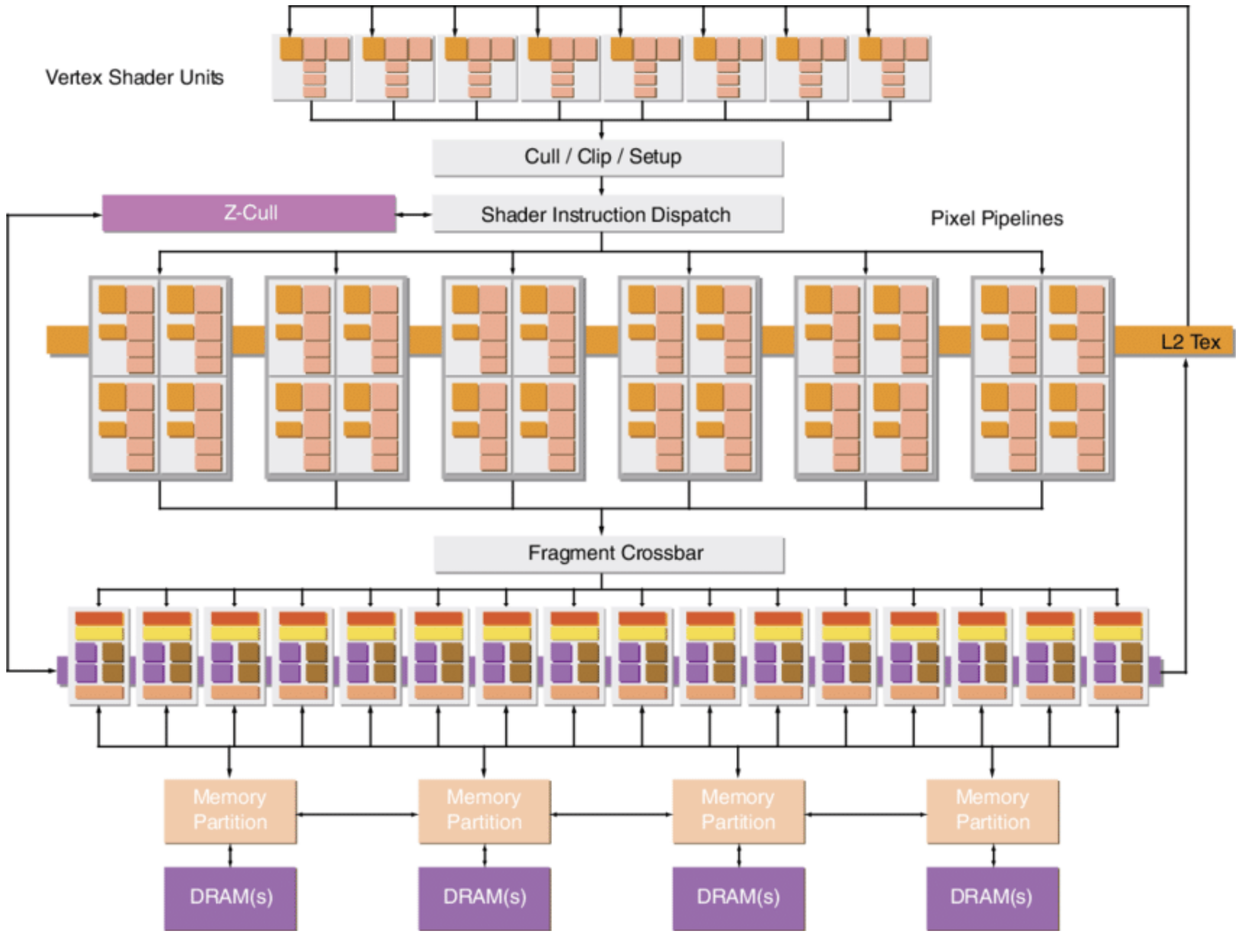
- Coordinate transformations (model → world → screen space)
- Per-vertex lighting calculations
- Vertex morphing and animation

Pixel Shader (Fragment Shader) Engine: Processes individual pixels (fragments), handling color calculations, texture blending, lighting effects, and other pixel-level operations.

- Per-pixel lighting and shading
- Texture mapping, filtering, and blending
- Complex visual effects (reflections, shadows, bump mapping)

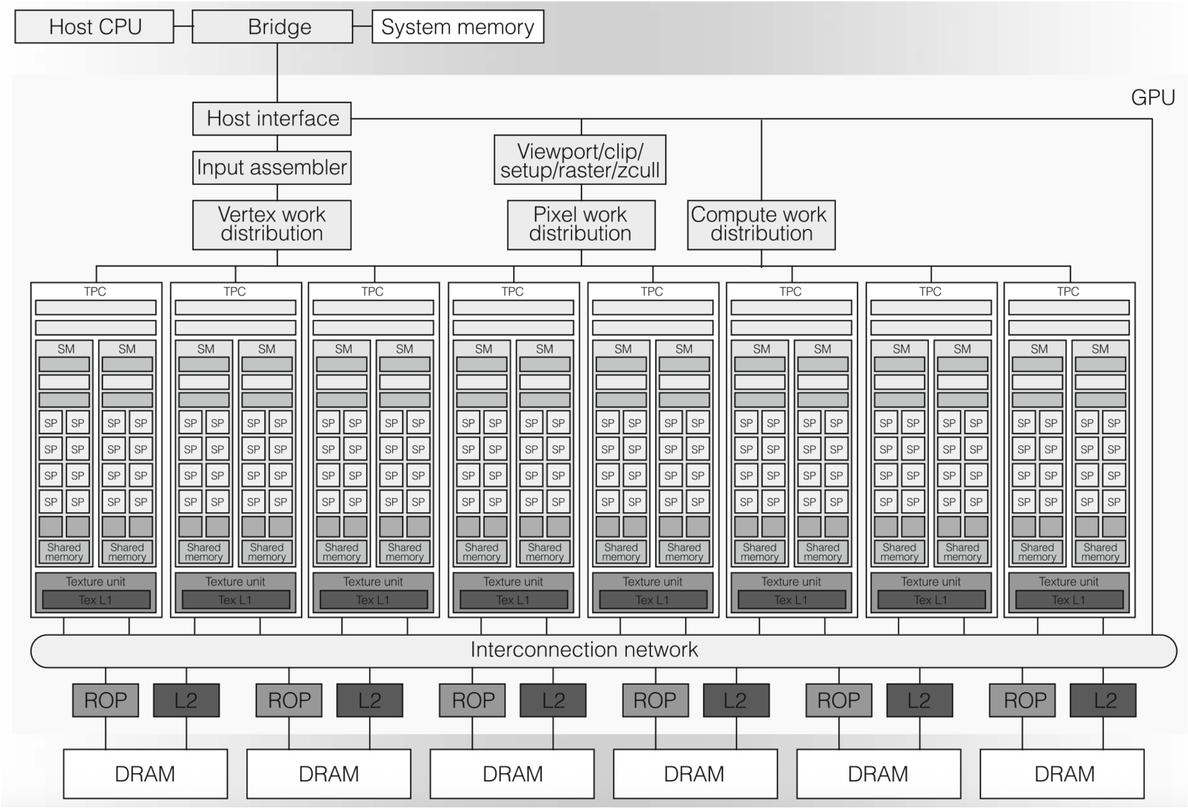
ROP (Raster Operations Pipeline): Final stage of the rendering pipeline, converting fragment outputs into pixels stored in the framebuffer.

- Depth (Z) testing and stencil operations
- Color blending and transparency
- Anti-aliasing (AA)
- Writing final pixel values to memory



Tesla 2006: Unified shader architecture

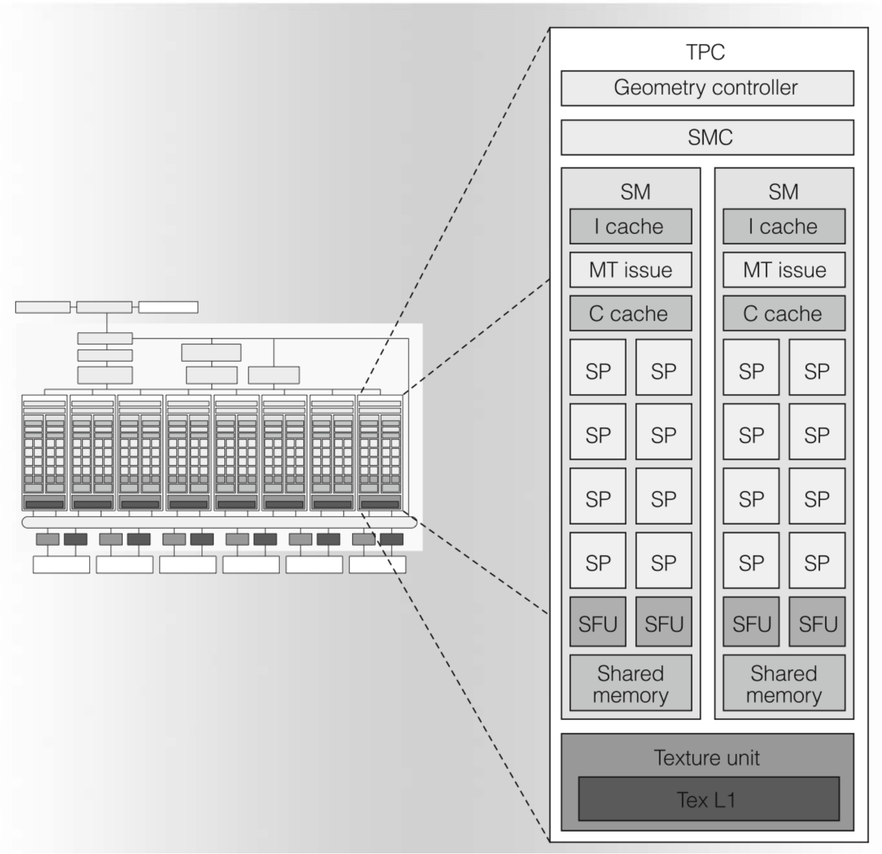
Overall Design	First CUDA-capable GPU (90nm, 2006), unified graphics and compute with five subsystems: control, compute, cache, memory, interconnect. Work distribution units dispatched tasks to scalable TPCs.
Key Modules	Each TPC had a Geometry Controller, SM Controller, 2 SMs, 1 Texture Unit + L1 cache, and 4 ROPs. Shared L2 caches, DRAM partitions, and display interface supported the whole chip.
SM Design	Each SM: 8 scalar SPs (FP32/INT32), 2 SFUs (math functions), 16 KB shared memory, instruction & constant caches. Scalar SP design simplified CUDA C support.
Workflow	CPU sends tasks → Input Assembler builds primitives → Vertex Shaders run → Rasterization & Z-cull → Pixel Shaders process fragments → ROPs handle blending/ depth/AA → output to memory/display.



In 2006, NVIDIA introduced the GeForce 8800. This design featured a “unified shader architecture” with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance.

Tesla 2006: Unified shader architecture

Overall Design	First CUDA-capable GPU (90nm, 2006), unified graphics and compute with five subsystems: control, compute, cache, memory, interconnect. Work distribution units dispatched tasks to scalable TPCs.
Key Modules	Each TPC had a Geometry Controller, SM Controller, 2 SMs, 1 Texture Unit + L1 cache, and 4 ROPs. Shared L2 caches, DRAM partitions, and display interface supported the whole chip.
SM Design	Each SM: 8 scalar SPs (FP32/INT32), 2 SFUs (math functions), 16 KB shared memory, instruction & constant caches. Scalar SP design simplified CUDA C support.
Workflow	CPU sends tasks → Input Assembler builds primitives → Vertex Shaders run → Rasterization & Z-cull → Pixel Shaders process fragments → ROPs handle blending/ depth/AA → output to memory/display.



Starting from now, software developers can write C code on GPU.

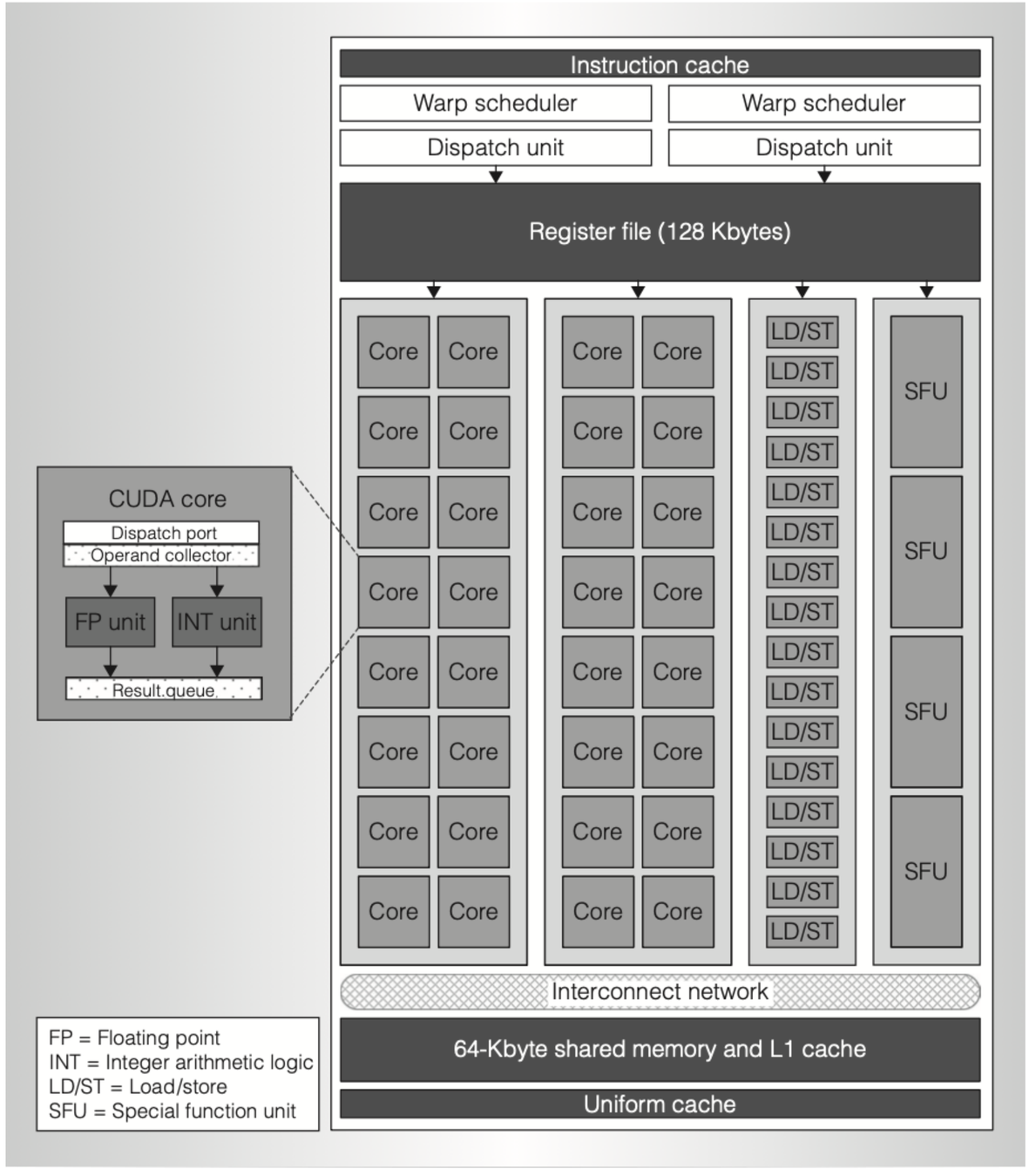
Fermi case study

Streaming Multiprocessors (SM): Each SM includes 32 CUDA processor cores, 16 load/ store units, and four special function units (SFUs). It also possesses a 64-Kbyte configurable shared memory+L1 cache, 128-Kbyte register file, instructions cache, and two multi-threaded wrap schedulers and two instruction dispatch units.

CUDA Cores: Each pipelined CUDA core executes an instruction per clock for a thread. With 32 cores architecture, an SM can execute up to 32 thread instructions per clock. Executable instructions include scalar floating-point instruction, implemented by floating-point unit (FP unit), and integer instruction, implemented by integer unit (INT unit).

Special functional units (SFU): The SFUs are in charge of executing 32-bit floating-point instructions for fast approximations of reciprocal, reciprocal square root, sin, cos, exp, and log functions. The approximations are precise to better than 22 mantissa bits.

Load/store units: The streaming multiprocessor load/store units execute load, store, and atomic memory access instructions. A warp of 32 active threads presents 32 individual byte addresses, and the instruction accesses each memory address. The load/store units coalesce 32 individual thread accesses into a minimal number of memory block accesses.



CUDA Core

CUDA Cores: General-purpose shader cores in NVIDIA GPUs, designed to handle a wide range of programmable tasks, including graphics rendering, general-purpose computing (GPGPU), and traditional compute workloads.

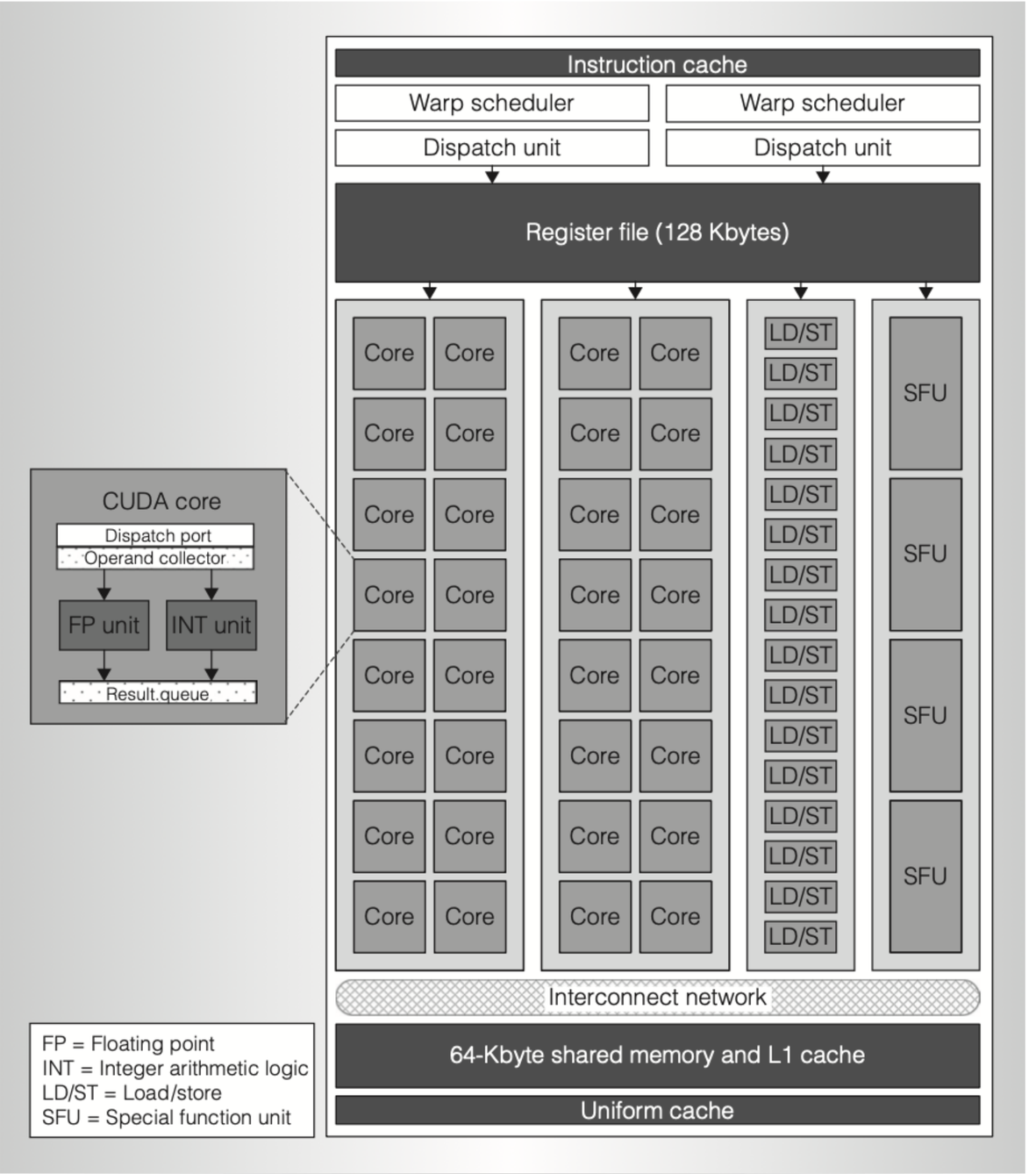
Purpose: The backbone of NVIDIA's unified shader architecture, introduced with the GeForce 8 series in 2006. They execute a variety of instructions, including floating-point (FP32, FP64), integer, and bitwise operations.

Structure: Each CUDA core is a scalar processor capable of performing one operation per clock cycle on a single data element. Multiple CUDA cores are grouped into Streaming Multiprocessors (SMs), which manage thread scheduling and shared memory.

Instruction Set: Supports a broad instruction set, enabling flexibility for graphics (e.g., vertex, geometry, fragment shaders) and compute tasks (e.g., physics simulations, ray tracing).

Precision: Primarily FP32 (single-precision floating-point) with limited FP64 (double-precision) support, depending on the GPU generation (e.g., higher-end GPUs like the A100 offer robust FP64).

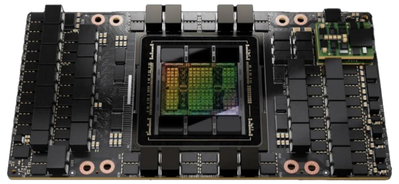
Limitations: Less efficient for matrix-intensive operations due to lack of specialized hardware, relying on software libraries (e.g., cuBLAS) to optimize such workloads.



Single-Chip Inference Performance - 1000X in 10 years

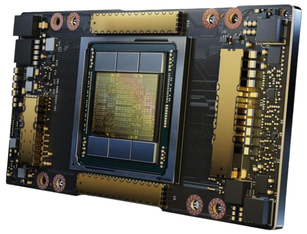
B200
20,000

NVIDIA introduced the GPU in 1999, and from 2010 to 2024 developed nine major architectures—Fermi through Blackwell. Over 15 years, CUDA evolved into NVIDIA's core advantage, with key innovations like Tensor Cores, NVLink, NVSwitch, and the Transformer Engine, establishing the company as a leader in AI, HPC, graphics, and autonomous technologies.



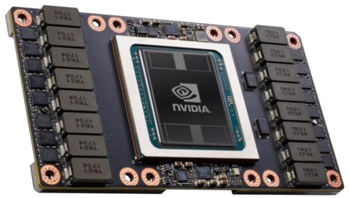
H100
4000.00

Sparsity

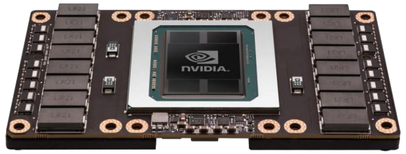


A100
1248.00

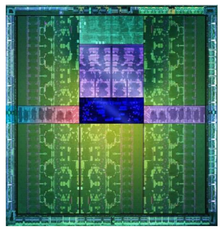
FP16
HMMA



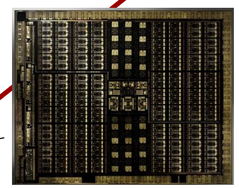
FP16
HDP4



FP32
FMA



INT8
IMMA



Q8000
261.00

V100
125.00

P100
21.20

M40
6.84

K20X
3.94

Int 8 TOPS

4500.00
4000.00
3500.00
3000.00
2500.00
2000.00
1500.00
1000.00
500.00
0.00

4/1/12
8/14/13
12/27/14
5/10/16
9/22/17
2/4/19
6/18/20
10/31/21
3/15/23

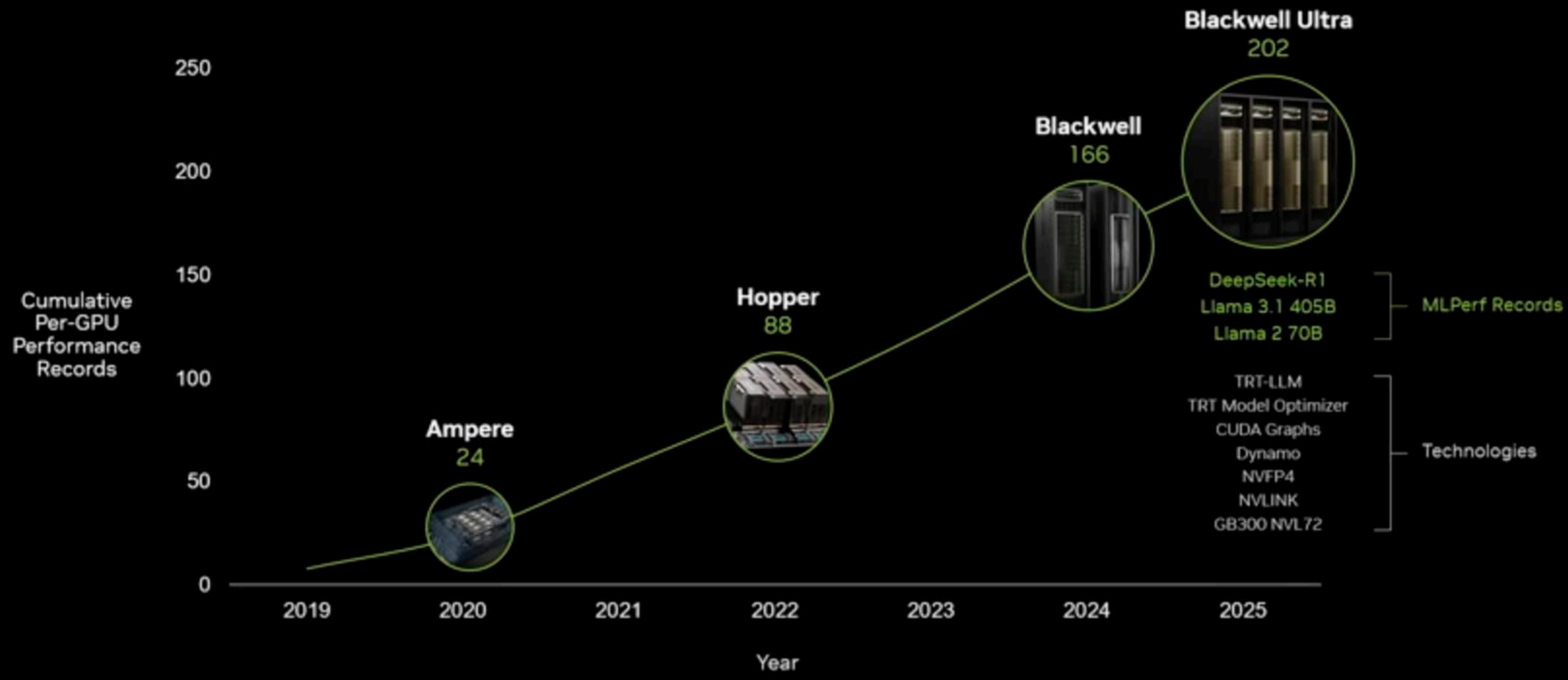
Increasing Massive Parallelism

Architecture	Release Year	CUDA Cores per SM	Total SMs	Total CUDA Cores
Tesla	2006	8	30	240
Fermi	2010	32	16	512
Kepler	2012	192	15	2880
Maxwell	2014	128	16	2048
Pascal	2016	128	20	2560
Volta	2017	64	80	5120
Turing	2018	64	72	4608
Ampere	2020	128	84	10752
Ada Lovelace	2022	128	144	18432
Blackwell	2024	128	170	21760

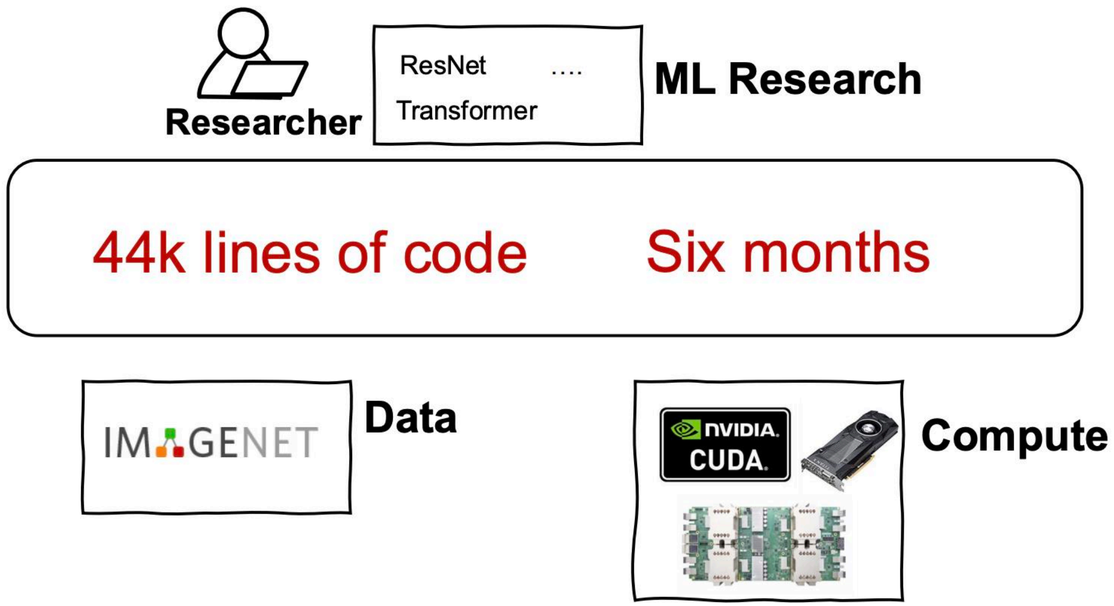
Ian Buck
NVIDIA
VP, Hyperscale and
HPC Computing

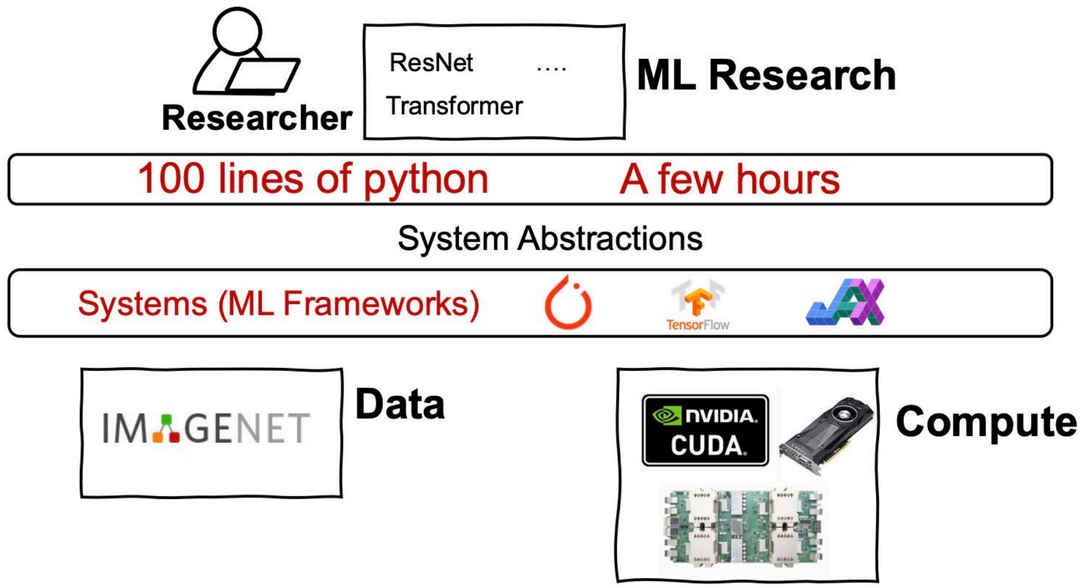
Hundreds of MLPerf Inference Records

Highest performing architectures on every model, every scenario



The Programming Challenge





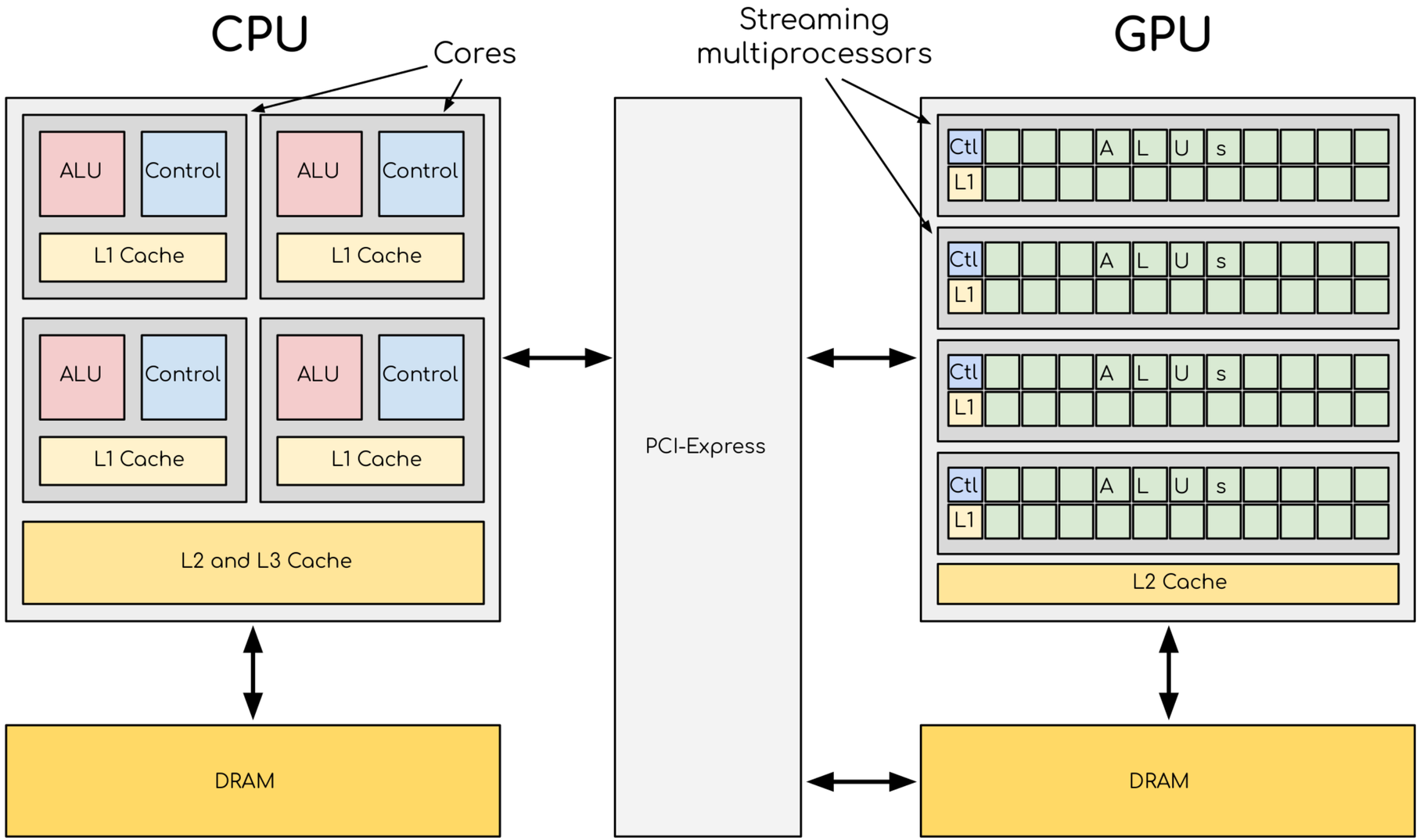
CUDA Compute Unified Device Architecture

- **Programming Model:** It provides a programming model that allows for the development of software that can execute parallel computations efficiently on GPUs.
- **Parallel Computing Platform:** CUDA enables developers to harness the parallel processing power of NVIDIA GPUs for general-purpose computing tasks, facilitating significant performance improvements in various applications.
- **API Support:** CUDA offers an API that allows software developers to utilize NVIDIA GPUs for general-purpose processing, enabling the development of high-performance applications across various domains.
- **Extensive Ecosystem:** Over the years, NVIDIA has built a comprehensive ecosystem around CUDA, including specialized code libraries and AI models, making it a dominant platform in AI and high-performance computing.

The History of CUDA

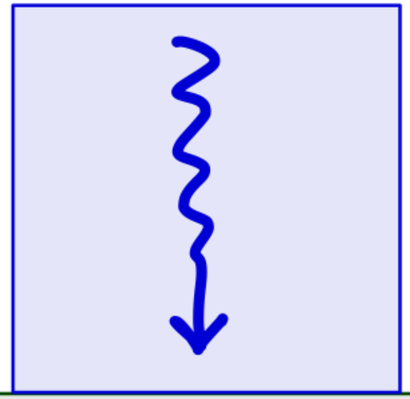
SC08
Conference for High Performance Computing
Austin Texas

How to program GPGPU?



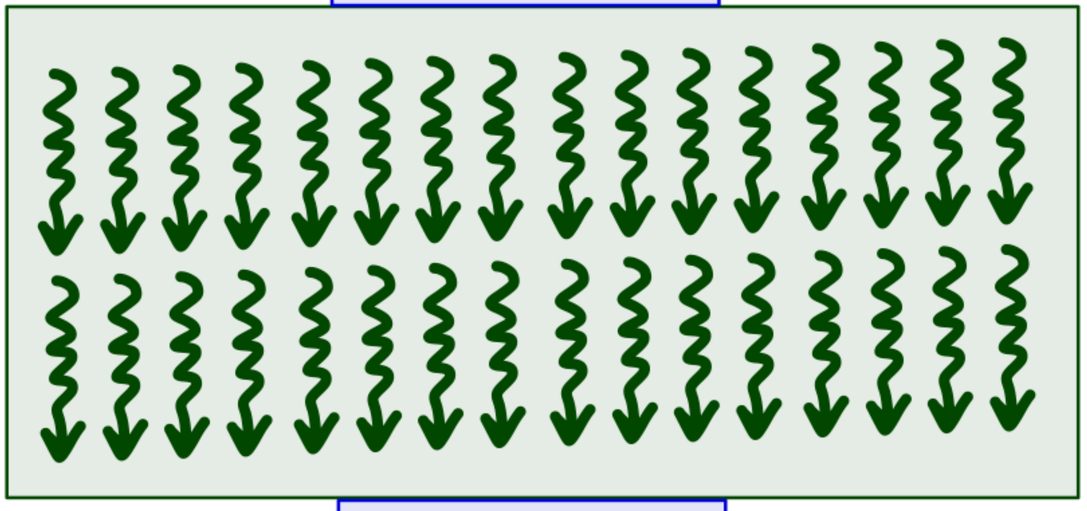
GPGPU as a really powerful accelerator

CPU thread

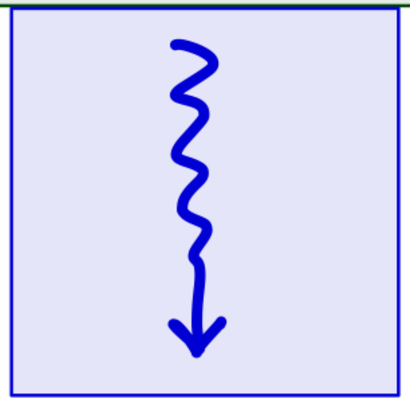


- Prepare data (matrices A, B, C).
- Copy data from RAM to VRAM.

GPU threads



- Parallel Matrix Multiplication.



- Copy results from VRAM to RAM.

CUDA

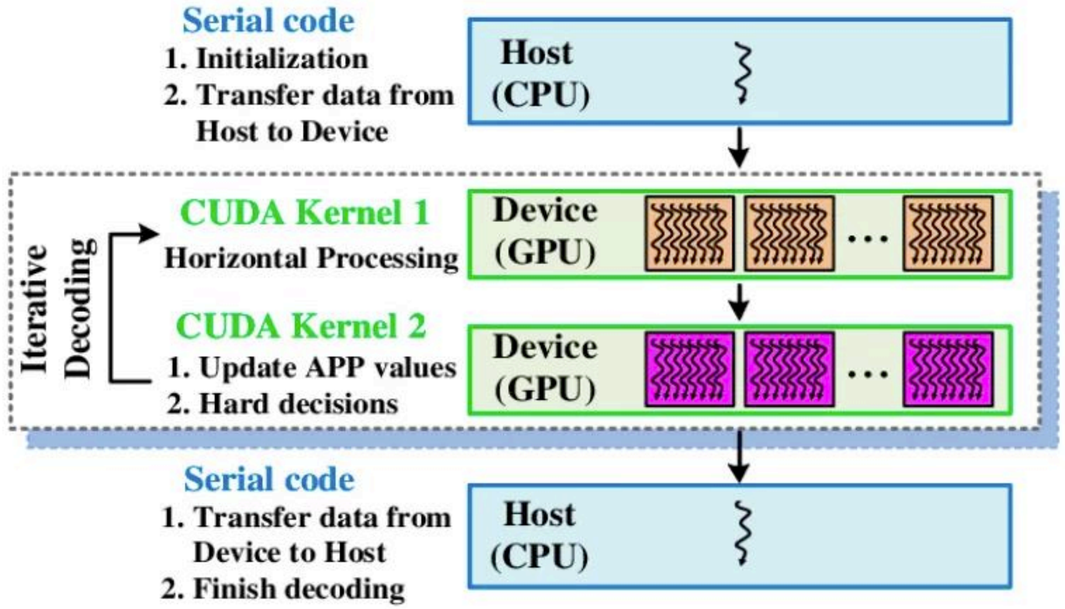
A CUDA program consists of two parts:

1. **Host Code (running on the CPU):**
Written in C/C++ (e.g., `.c` files). Responsible for memory allocation and data transfer.

2. **Device Code (running on the GPU):**
Written in CUDA C/C++ (e.g., `.cu` files). CUDA C++ extends standard C++ functionality so that device code can declare and define a CUDA kernel function using the `__global__` specifier. A kernel function is launched using the CUDA kernel launch operator `<<<...>>>`, which starts a GPU parallel computing task. The kernel is then executed *N* times in parallel by *N* different CUDA threads.

As shown below, the host code executes serially, while the device code executes in parallel. At the boundary between host code and device code, synchronization functions must be called to synchronize (the data).

Note: Excessive synchronization function calls can lead to performance degradation in CUDA programs.

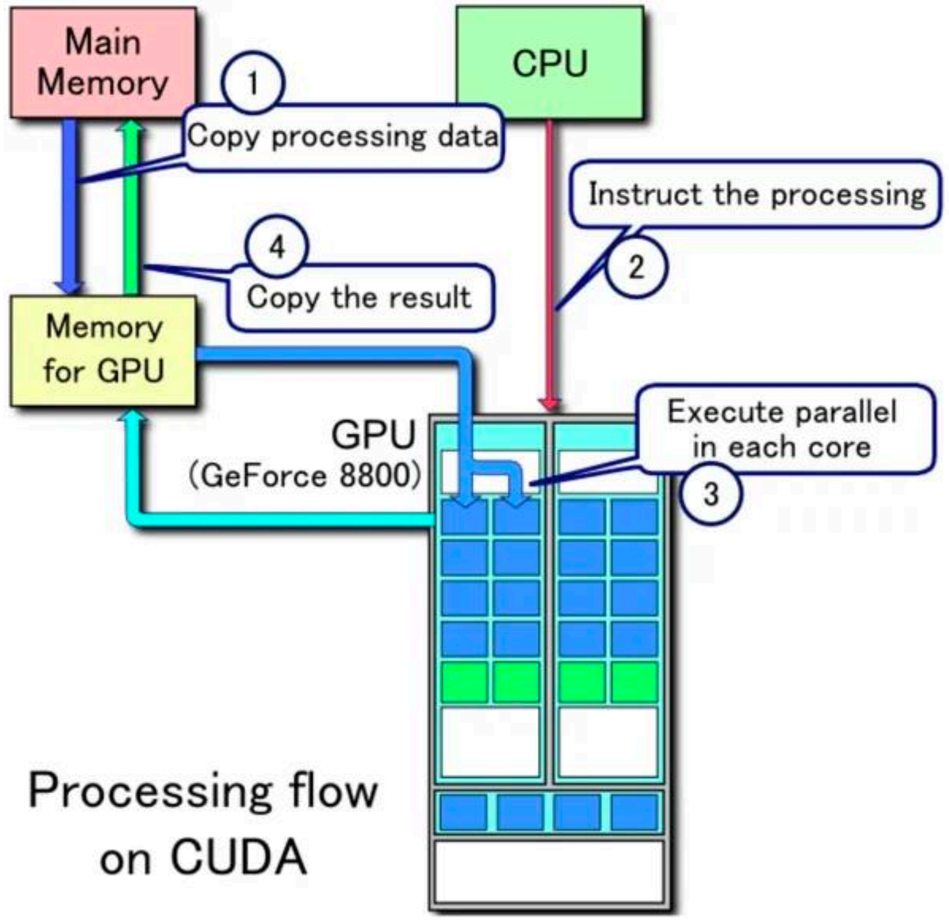


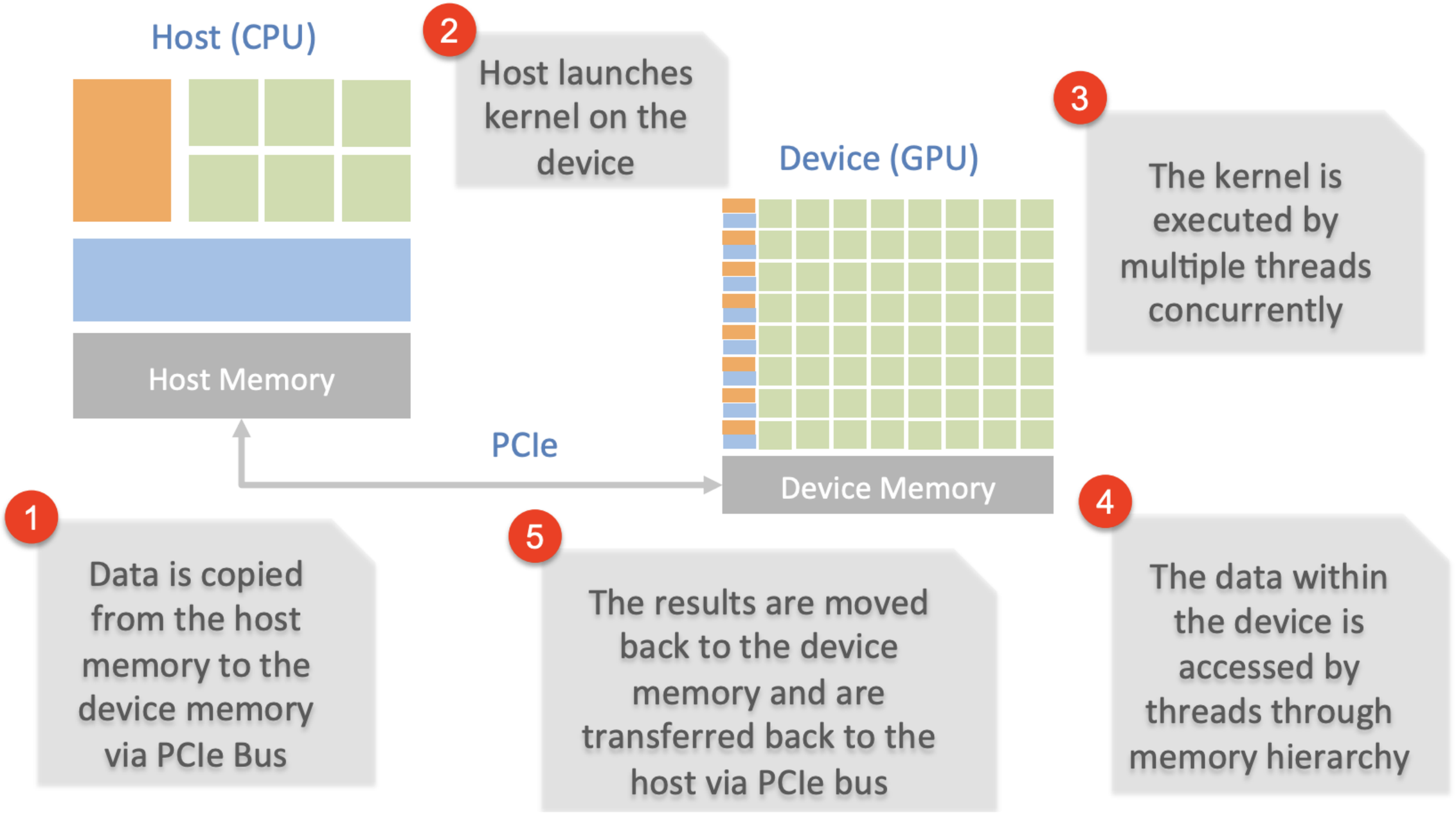
CUDA

A typical CUDA program processing flow is as follows: Since GPU memory and CPU main memory are physically separate, the host program (Host Code) must first allocate both main memory and GPU memory space. Before the GPU can execute the Device Code, the code and data must be transferred from main memory to GPU memory. Only then can the GPU begin executing the kernel function, as illustrated below:

1. The Host Code copies the Device Code and data from main memory into GPU memory.
2. The Host Code launches the kernel.
3. The GPU executes the kernel function in parallel, accessing data from GPU memory during execution.
4. Once execution is complete, the GPU writes the results from GPU memory back to main memory.

Note: In large language model (LLM) training, massive amounts of sample data, checkpoint data, and weight outputs all require persistent storage. The storage path inevitably involves GPU memory, main memory, and disk/network. Therefore, accelerating data transfer across these paths is also one of the key aspects of performance optimization.




Host (CPU)
Host Memory
PCIe
Device (GPU)
Device Memory
1
Data is copied from the host memory to the device memory via PCIe Bus
2
Host launches kernel on the device
3
The kernel is executed by multiple threads concurrently
4
The data within the device is accessed by threads through memory hierarchy
5
The results are moved back to the device memory and are transferred back to the host via PCIe bus

Kernel

A **kernel** is a function written in CUDA C/C++ (or another GPU programming language) that runs on the GPU and is executed **in parallel by many threads**.

- Declared with the `__global__` qualifier.
- Launched from the host (CPU) using the special syntax:

```
myKernel<<<numBlocks, threadsPerBlock>>>(arguments);
```

- When launched, CUDA creates a **grid** of threads (organized into blocks), and *each thread executes the same kernel function*, but on different data.

```cpp
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Kernel

```
// HOST code to queue kernel
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

__global__ specifies kernel

ThreadIdx & blockIdx determine thread rank that is mapped to array index

```
__global__ void simpleKernel(int N, float *d_a){

    // Convert thread and thread-block indices into array index
    const int n  = threadIdx.x + blockDim.x*blockIdx.x;

    // If index is in [0,N-1] add entries
    if(n<N)
      d_a[n] = n;
}
```

Action performed by each thread

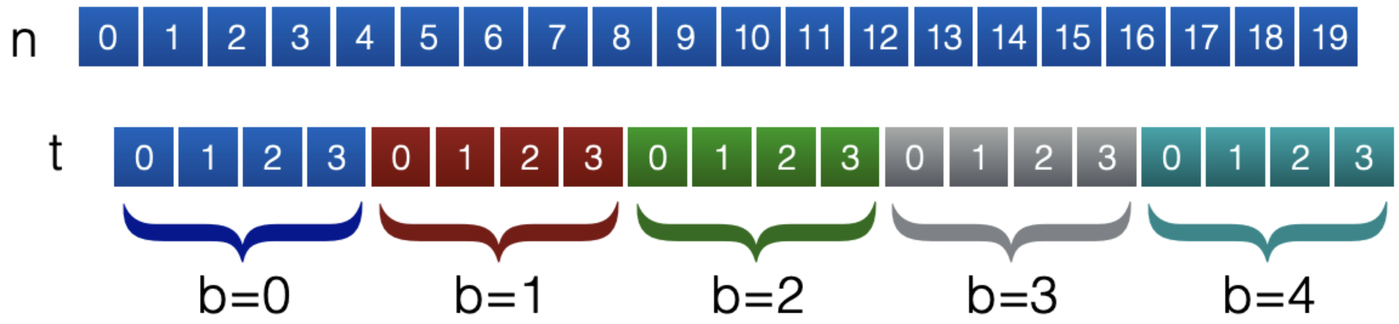
Key observation: the loops are implicitly executed by thread parallelism and *do not* appear in the CUDA kernel code.

Kernel

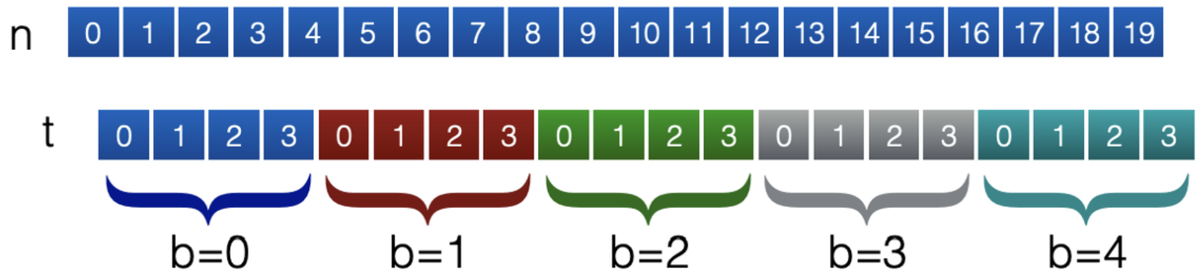
- **Entry point for GPU computation**: The kernel defines what each thread does.
- **Single Program, Multiple Data (SPMD)** style: All threads run the same program (kernel), but each thread is distinguished by its thread/block ID.
- **Massively parallel**: Thousands of threads can be executing the kernel simultaneously.
- **Scoped execution**: A kernel has one associated grid per launch; within that grid, threads are grouped into blocks.

```
void serialSimpleKernel(int N, float *d_a){

  for(n=0;n<N;++n){ // loop over N entries

    d_a[n] = n;

  }
}
```

```
void serialSimpleKernel(int N, float *d_a){

  for(n=0;n<N;++n){ // loop over N entries

    d_a[n] = n;

  }
}
```



```
void tiledSerialSimpleKernel(int N, float *d_a){

  for(blockIdx.x=0;blockId.x<gridDim.x;++blockIdx.x){ // loop over blocks

    for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block

      // Convert thread and thread-block indices into array index
      const int n  = threadIdx.x + blockDim.x*blockIdx.x;

      // If index is in [0,N-1] add entries
      if(n<N)
        d_a[n] = n;
    }
  }
}
```



Grids, Blocks and Threads

Array traversal

```
int index = threadIdx.x + blockDim.x * blockIdx.x;
```

```
blockDim.x  = 4
blockIdx.x  = 0
threadIdx.x = 0, 1, 2, 3
Index       = 0, 1, 2, 3
```

```
blockDim.x  = 4
blockIdx.x  = 1
threadIdx.x = 0, 1, 2, 3
Index       = 4, 5, 6, 7
```

Grids, Blocks and Threads

A similar approach is used for 3D threads and 2D / 3D grids; can be very useful in 2D / 3D finite difference applications.

How do 2D / 3D threads get divided into warps?

1D thread ID defined by

```
threadIdx.x +
threadIdx.y * blockDim.x +
threadIdx.z * blockDim.x * blockDim.y
```

and this is then broken up into warps of size 32.

Example

```
int main(int argc,char **argv){
    int N = 3789; // size of array for this DEMO

    float *d_a;  // Allocate DEVICE array
    cudaMalloc((void**) &d_a, N*sizeof(float));

    int B = 512;
    dim3 dimBlock(B,1,1);          // 512 threads per thread-block
    dim3 dimGrid((N+B-1)/B, 1, 1); // Enough thread-blocks to cover N

    // Queue kernel on DEVICE
    simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);

    // HOST array
    float *h_a = (float*) calloc(N, sizeof(float));

    // Transfer result from DEVICE array to HOST array
    cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Print out result from HOST array
    for(int n=0;n<N;++n) printf("h_a[%d] = %f\n", n, h_a[n]);
}
```

1. Allocate array space on DEVICE:

```
float *d_a; // Allocate DEVICE array (pointers used as array handles)
cudaMalloc((void**) &d_a, N*sizeof(float));
```

2. Design thread-array:

```
dim3 dimBlock(512,1,1);          // 512 threads per thread-block
dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
```

3. Queue compute task on DEVICE:

```
// specify number of threads with <<< block count, thread count >>>
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

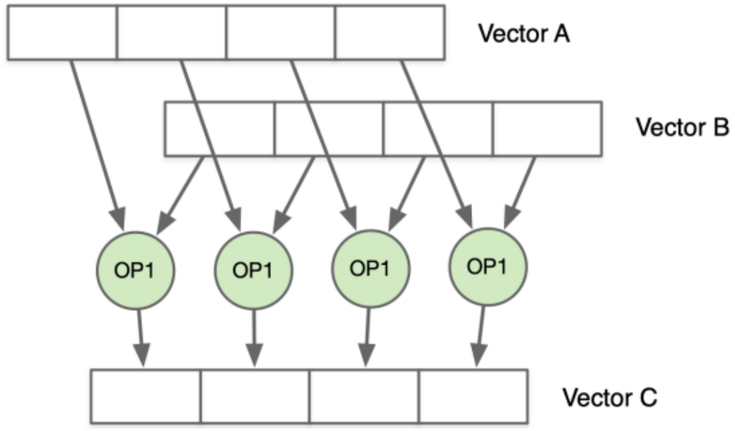
4. Copy results from DEVICE to HOST:

```
float *h_a = (float*) calloc(N, sizeof(float));
cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost)
```

SIMT

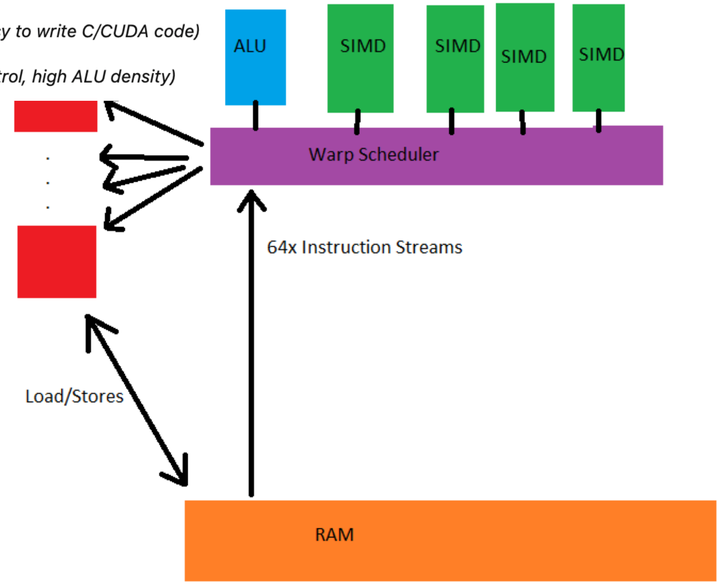
Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading. All instructions in all "threads" are executed in lock-step. The SIMT execution model is the primary programming model for GPUs.

$$C[0:3] = A[0:3] \times B[0:3]$$



(SIMD)

- Programmer's view: *scalar multithreading (easy to write C/CUDA code)*
- Hardware's view: *SIMD efficiency (shared control, high ALU density)*



(SIMT)

SIMT

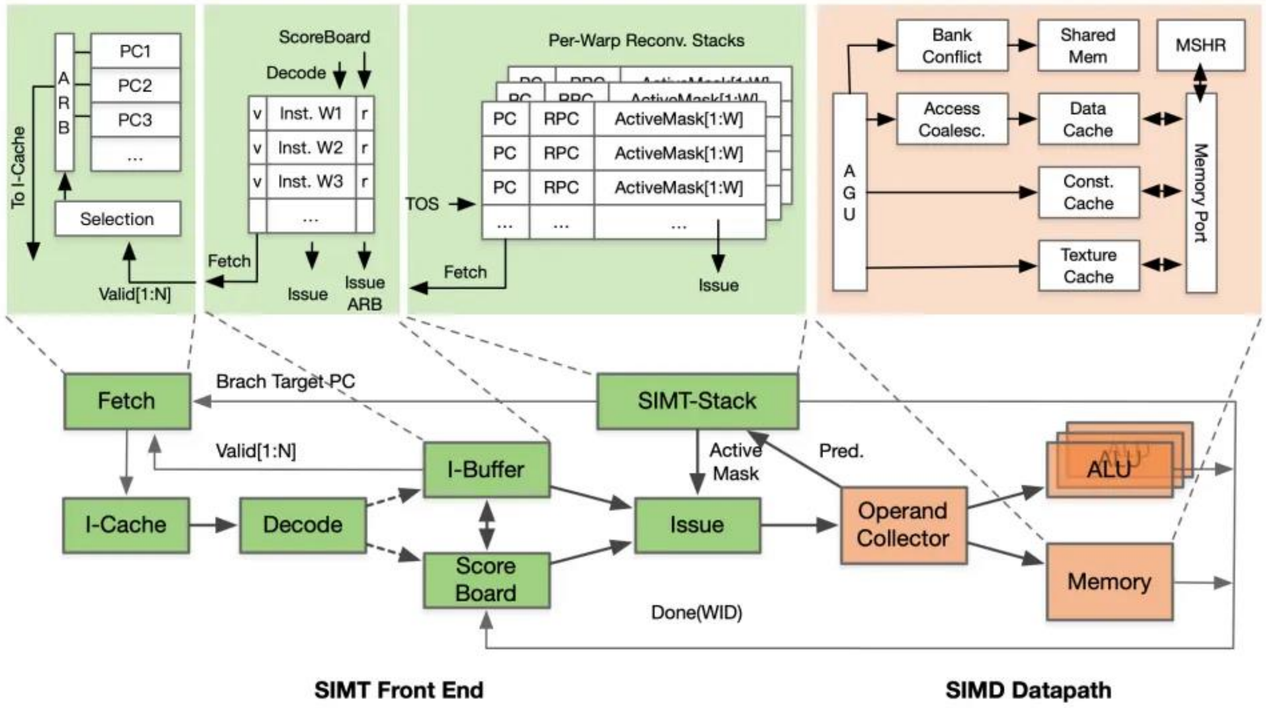
```
void add_arrays_sisd(float* a, float* b, float* c, int n) {
    // 纯串行处理：每个元素独立计算
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

```
void add_arrays_simd(float* a, float* b, float* c, int n) {
    // 每次迭代处理 8 个元素 (AVX2 256-bit = 8x float)
    for (int i = 0; i < n; i += 8) {
        // 加载 256 位数据
        __m256 va = _mm256_loadu_ps(&a[i]);
        __m256 vb = _mm256_loadu_ps(&b[i]);

        // 向量加法 (单指令操作所有8个元素)
        __m256 vc = _mm256_add_ps(va, vb);

        // 存储结果
        _mm256_storeu_ps(&c[i], vc);
    }
```

```
// GPU 核函数 (每个线程处理一个元素)
__global__ void add_arrays_simt(float* a, float* b, float* c, int n) {
    // 计算当前线程处理的元素索引
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // 确保不越界
    if (i < n) {
        // 每个线程独立执行相同的加法指令
        c[i] = a[i] + b[i];
    }
}
```

```
SISD:
  CPU: [a0+b0] → [a1+b1] → [a2+b2] → ... (串行执行)

SIMD:
  CPU: [a0,a1,a2...,a7] + [b0,b1,b2...,b7] = [c0,c1,c2...,c7] (单指令并行)

SIMT:
  GPU:
    Thread0: a0 + b0 = c0
    Thread1: a1 + b1 = c1
    Thread2: a2 + b2 = c2
    ... (所有线程同时执行相同加法指令)
```

SIMT



- Programmer's view: *scalar multithreading (easy to write C/CUDA code)*
- Hardware's view: *SIMD efficiency (shared control, high ALU density)*

SIMT = Front-End Control (Warp/Thread Scheduler) + Back-End Execution (SIMD)

Front-End Control (Warp Scheduler):

- **Fetch stage:** Fetch → I-Cache → Decode → I-Buffer. The Warp Scheduler uses the PC (program counter) to obtain the address of the next instruction from main memory.
- **Instruction issue stage:**
  - **I-Buffer**
  - **Scoreboard:** Tracks resource conflicts and data dependencies.
  - **Issue / MT Issue (Multi-thread issue):** Distributes Warps/Threads to CUDA cores.
  - **SIMT-Stack:** Manages branch divergence caused by if-else statements.

Back-End Execution (SIMD):

- **Execution stage:**
  - **Operand Collector:** Fetches the required operands from the Register Files.
  - **ALU:** Executes arithmetic instructions (e.g., FP32/INT32 operations).
  - **Memory:** Handles LD/ST (load/store) requests.

SIMT

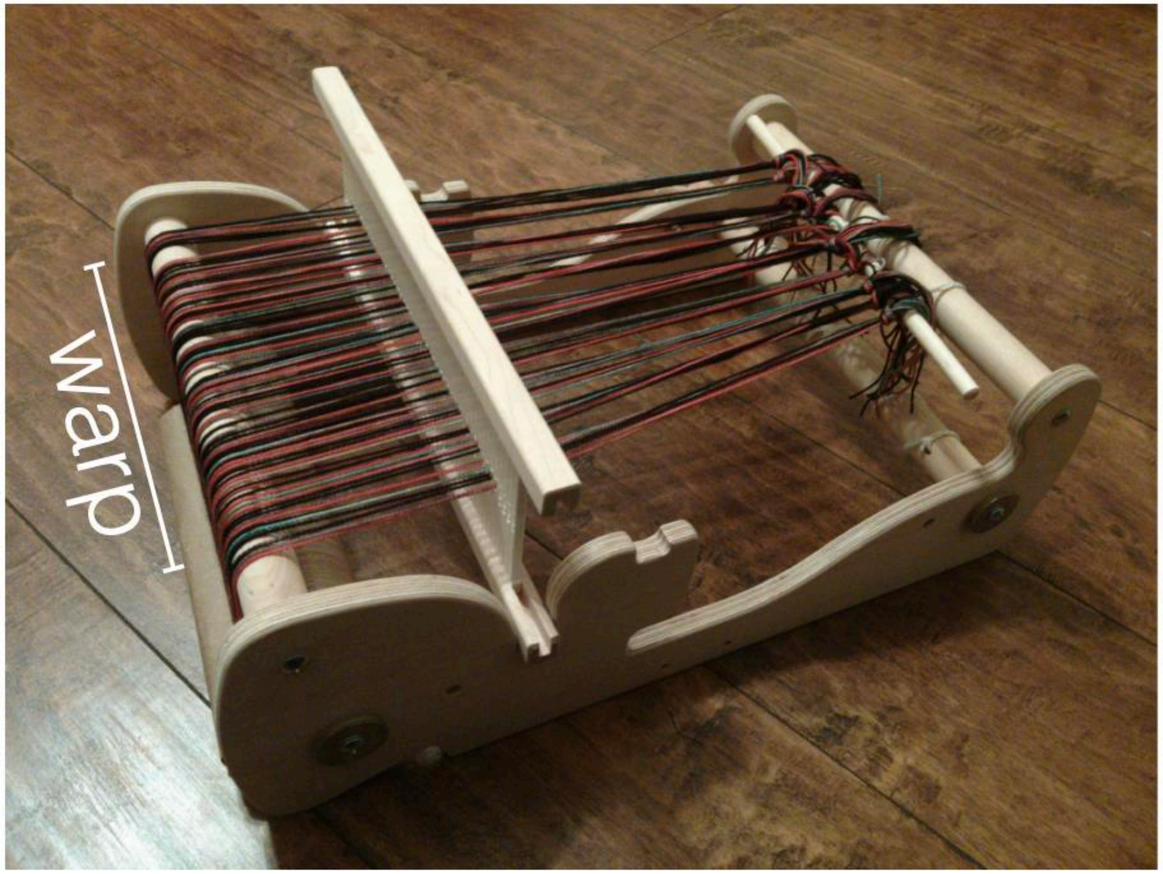
- **Warp Scheduler:** Manage 32 (or more) threads per warp with a shared instruction stream.
- **SIMT Stack:** Track divergence (if/else branches) and reconverge threads at join points.
- **Scoreboard + Operand Collector:** Handle dependencies and resource conflicts across thousands of in-flight threads.
- **Massive Register File:** Provide enough register state for tens of thousands of logical threads, far bigger than CPUs.
- **Latency Hiding:** Dynamically swap warps to cover hundreds of cycles of DRAM latency without stalling pipelines.

Warp

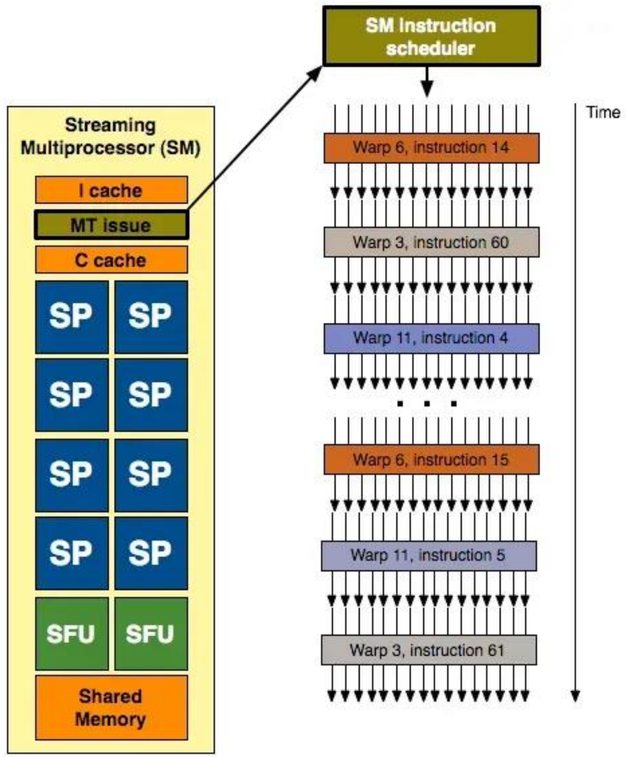
In CUDA programming, a warp is the implementation of the Single Instruction, Multiple Threads (SIMT) execution model. A warp consists of 32 threads that execute the same instruction simultaneously, allowing for efficient parallel processing on NVIDIA GPUs.

- **Memory Sharing:** Threads within a warp can efficiently share data using shared memory, which is a fast, on-chip memory accessible to all threads in the same block. Efficient use of shared memory within a warp can lead to significant performance improvements.
- **Warp Divergence** : When threads within the same warp follow different execution paths due to conditional statements (e.g., if-else branches). In such cases, the warp must serialize the execution of each divergent path, leading to reduced parallel efficiency and potential performance degradation.
- **Performance Considerations:** Optimal performance is achieved when all threads in a warp follow the same execution path. Divergence, where threads take different paths, can lead to performance degradation.

Warp



Warp

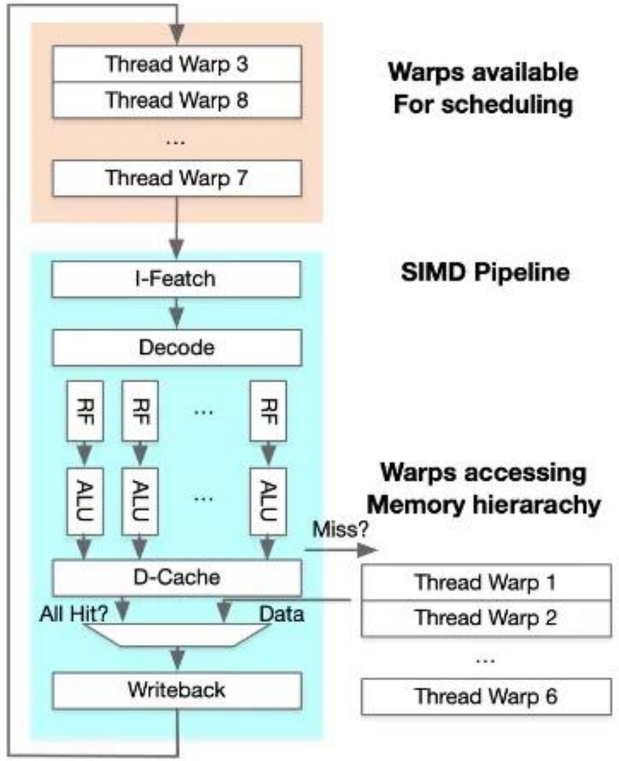


Imagine a block with 128 threads, split into 4 warps (32 threads each):

- Warp 0 might be executing a floating-point operation.
- Warp 1 might be fetching data from global memory (stalled).
- Warp 2 might be performing an integer operation.
- Warp 3 might be idle or waiting for a branch resolution.

The SM can execute instructions from Warp 0 and Warp 2 in parallel on different execution units while Warp 1 waits. This is how parallelism across warps is achieved.

Warp



Imagine a block with 128 threads, split into 4 warps (32 threads each):

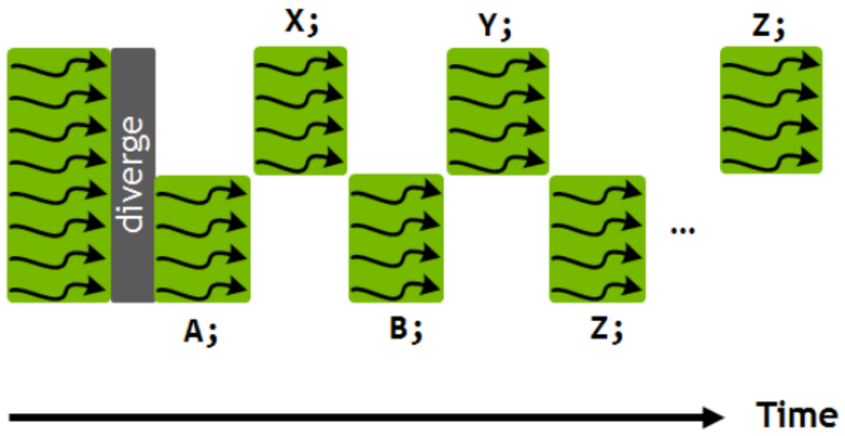
- Warp 0 might be executing a floating-point operation.
- Warp 1 might be fetching data from global memory (stalled).
- Warp 2 might be performing an integer operation.
- Warp 3 might be idle or waiting for a branch resolution.

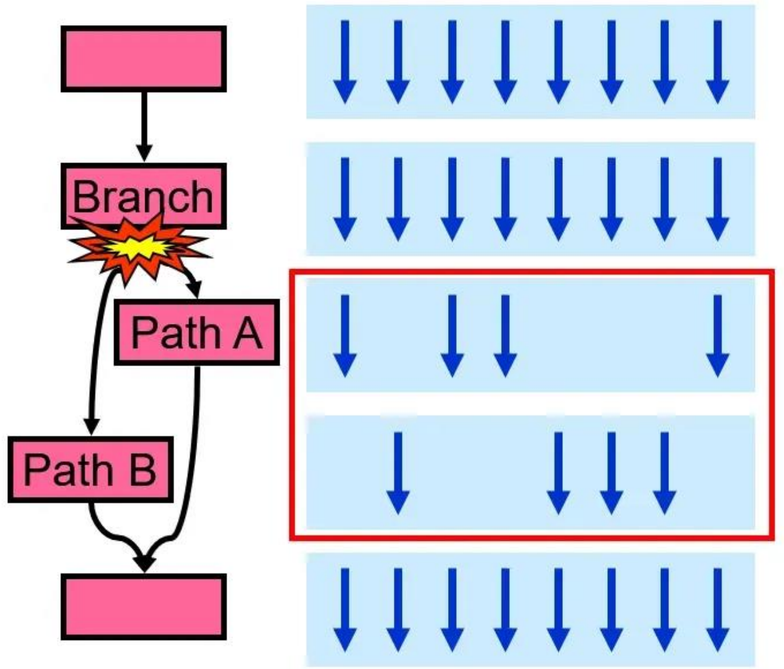
The SM can execute instructions from Warp 0 and Warp 2 in parallel on different execution units while Warp 1 waits. This is how parallelism across warps is achieved.

Divergence

Warp divergence happens when threads in the same warp take different control flow paths (e.g., inside an if/else or loop).

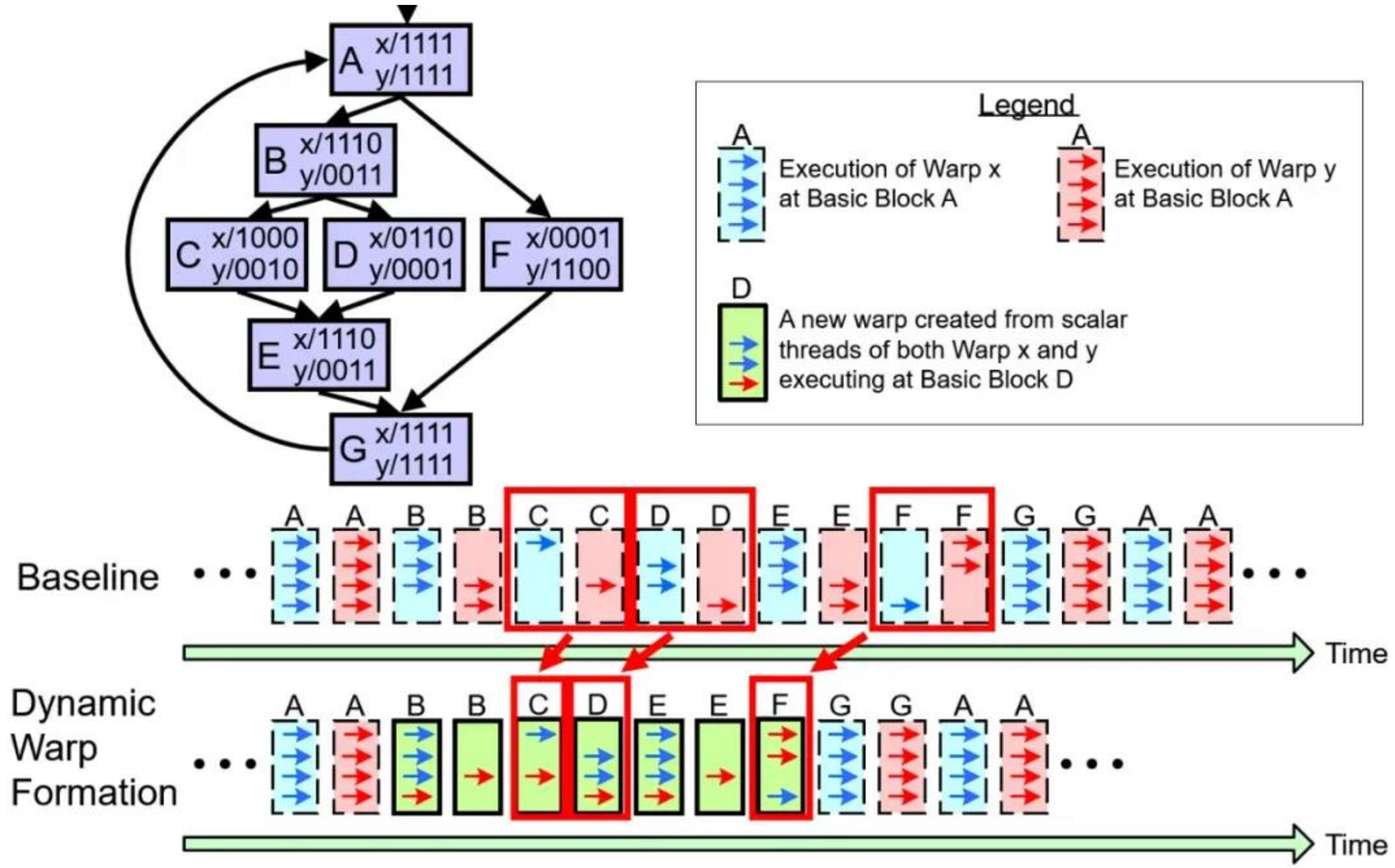
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



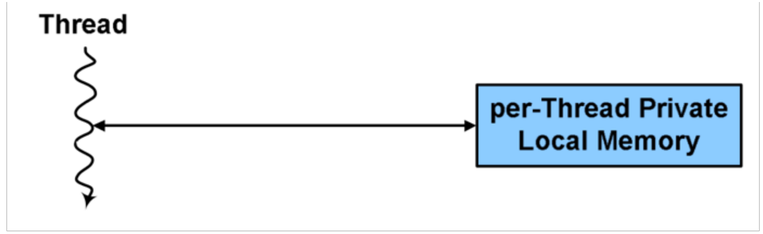


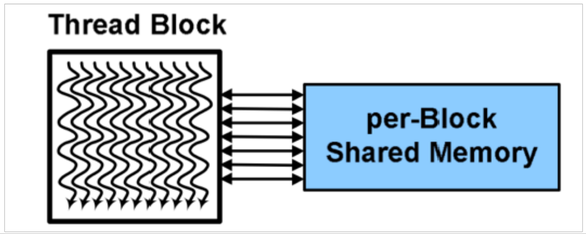
Dynamic Warp Formation

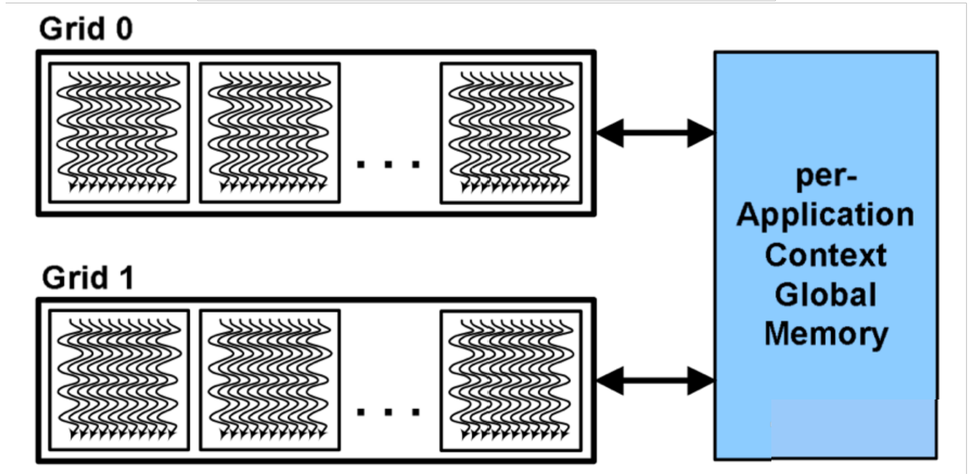
Dynamic Warp Formation (DWF) is a hardware technique to reduce SIMT branch divergence overhead. DWF solves wrap divergence by dynamically regrouping threads that share the same program counter into new warps at runtime, allowing them to execute in parallel again. This improves ALU utilization and overall throughput while preserving the scalar-thread programming model seen by developers.

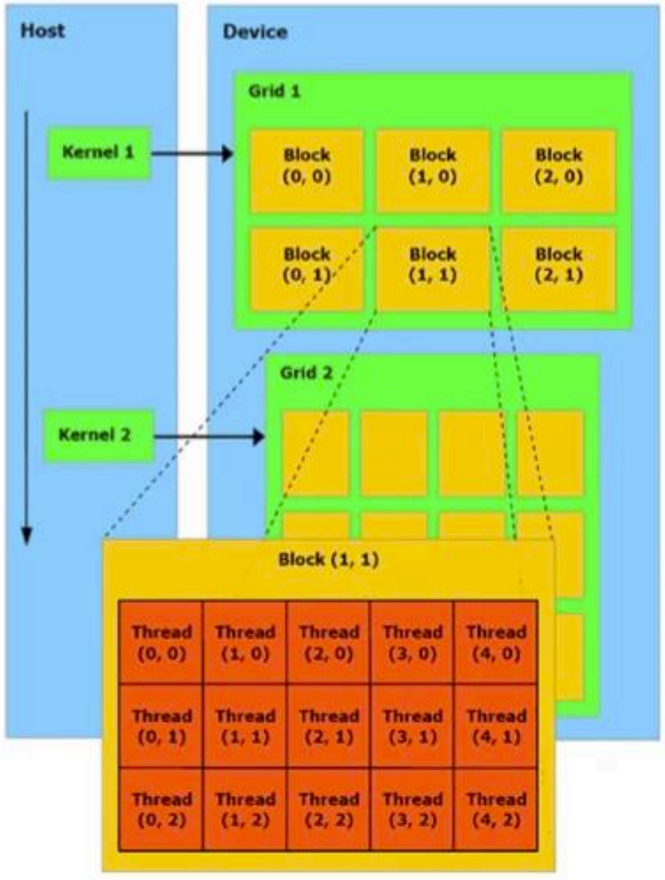


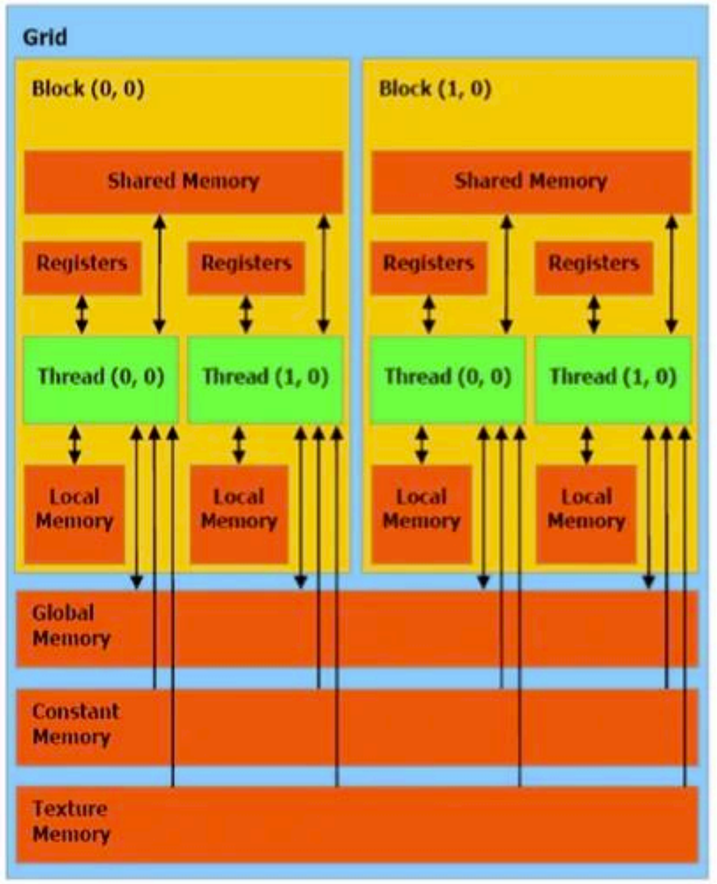
Execution Model





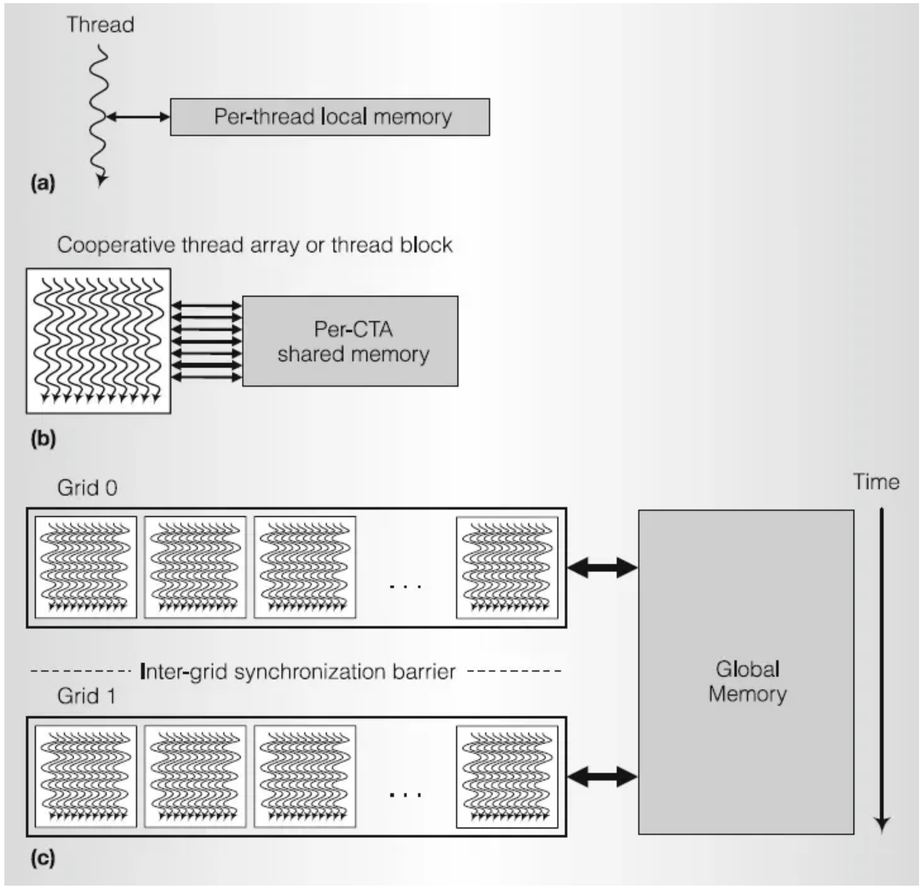






Execution Model

Within each block, the threads are automatically divided into warps (32 threads each) by the MT issue / warp scheduler hardware. For example, a block containing 256 threads will be divided into 8 warps.



CUDA Abstraction	Maps To (Hardware)	Composition	Memory Scope	Notes
Thread	CUDA Core (EU + Registers)	Smallest unit of execution	Registers, Local Memory (private)	Executes one instance of the kernel
Block	Streaming Multiprocessor (SM)	Contains many threads (128–1024)	Shared Memory (fast, on-chip)	Threads cooperate via shared memory
Grid	Entire GPU Device	Contains many blocks	Global Memory (off-chip DRAM)	Grids independent; sync only within blocks

Where is the bottleneck?

–Tom Jerry

Memory Bandwidth

Memory bandwidth – rate at which the data is transferred – is a valuable metric to gauge the performance of an application

Theoretical Bandwidth

Memory bandwidth (GB/s) = Memory clock rate (Hz) × interface width (bytes) / 10^9

Real Bandwidth (Effective Bandwidth)

Bandwidth (GB/s) = [(bytes read + bytes written) / 10^9] / execution time

If real bandwidth is much lower than the theoretical then code may need review

Optimize on Real Bandwidth

May also use profilers to estimate bandwidth and bottlenecks

Matrix Multiplication

Kernel analysis

2 floating point read accesses, 2 x 4 bytes = 8 bytes per one multiply and add that is 2 floating point operations per second (add and multiply). Hence the throughput is 8 bytes / 2 = 4B / FLOPs.

Theoretical peak of Fermi is 530 GFLOPs

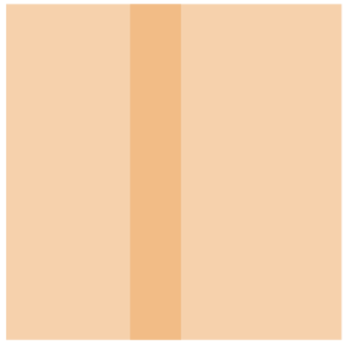
To achieve peak will require bandwidth of 4 x 530 = 2120 GB/s

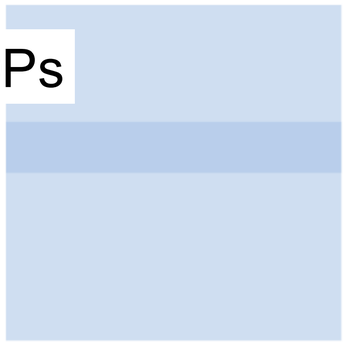
The actual bandwidth is 177GB/s

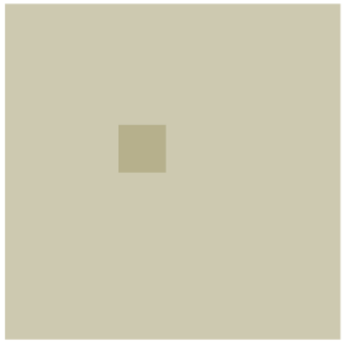
With this bandwidth it yields 177/4 = 44.25 GFLOPs

About 12 times below peak performance.

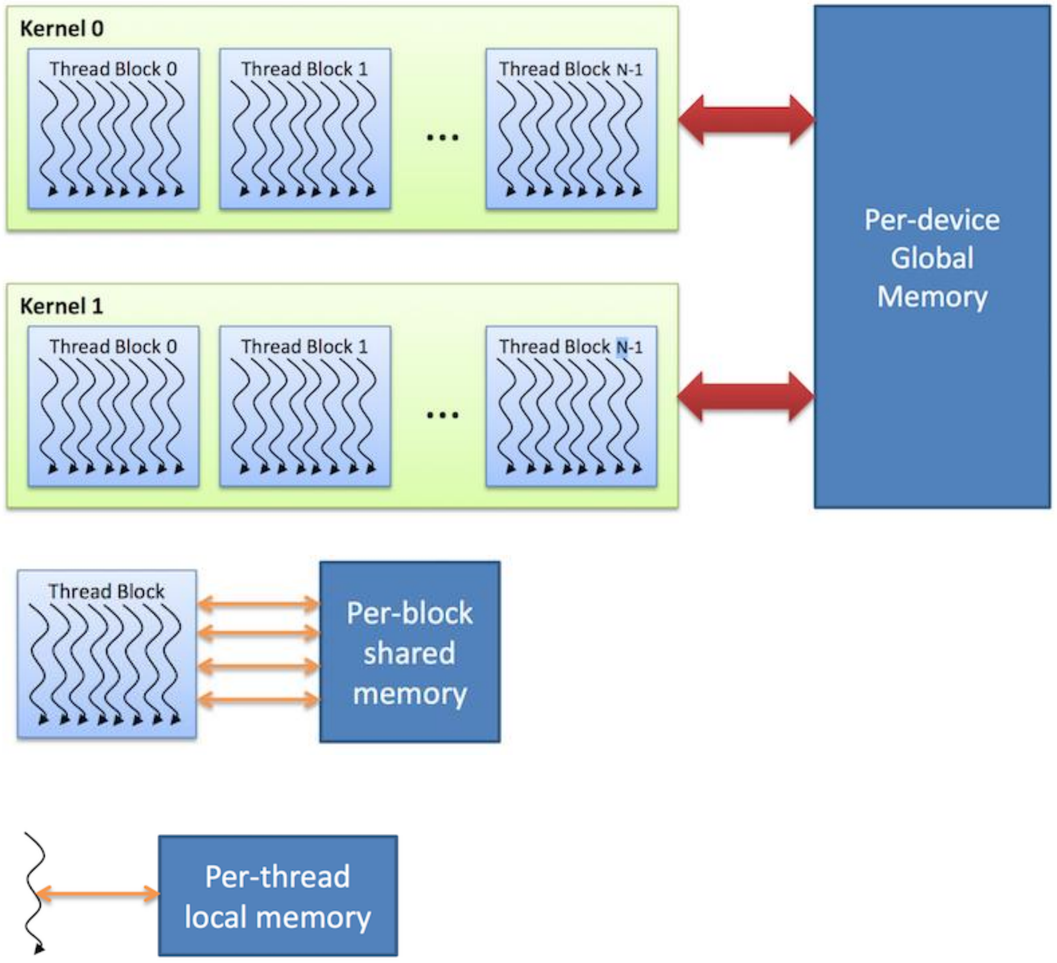
In practice it will be slower.

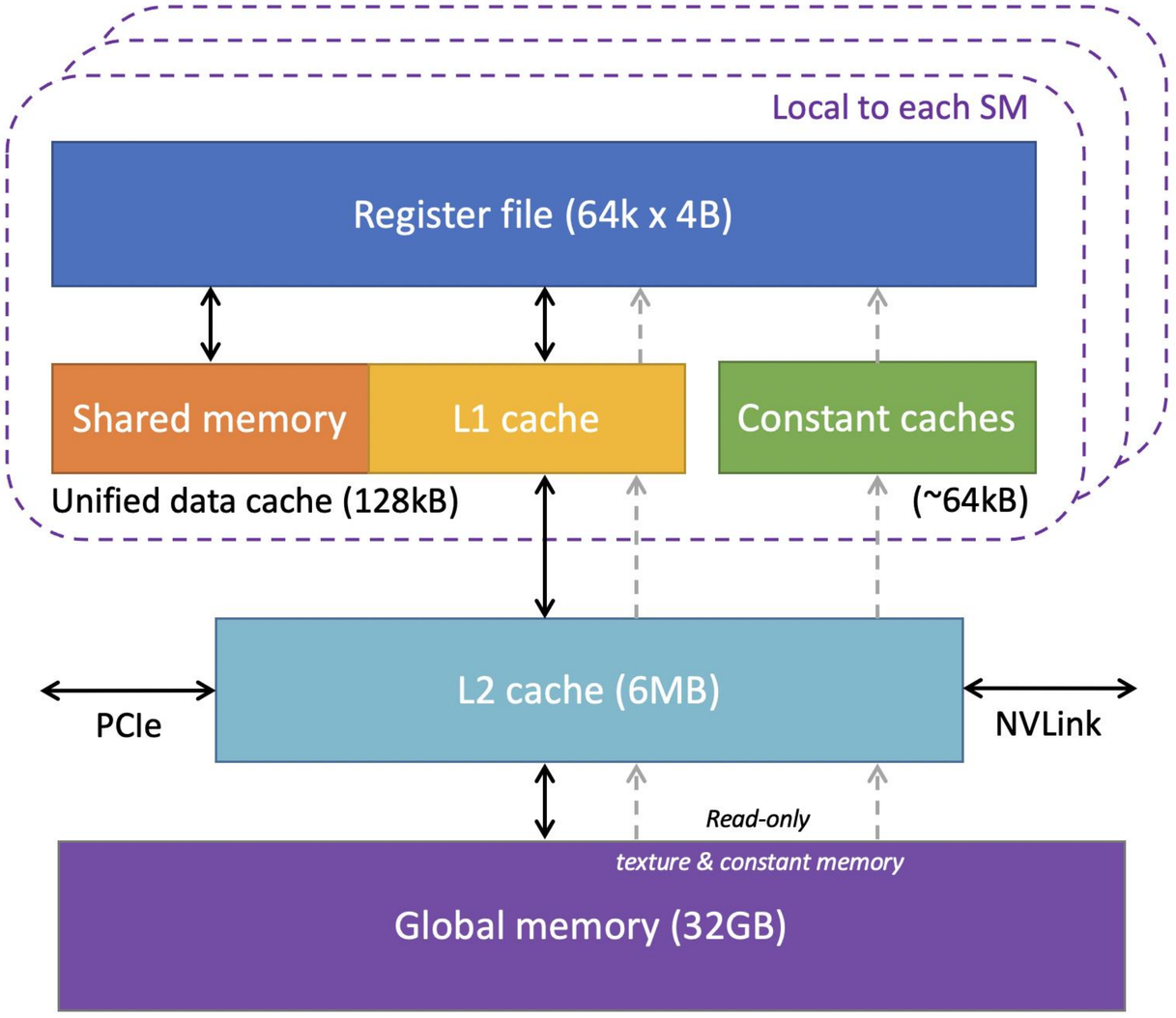






Memory Hierarchy

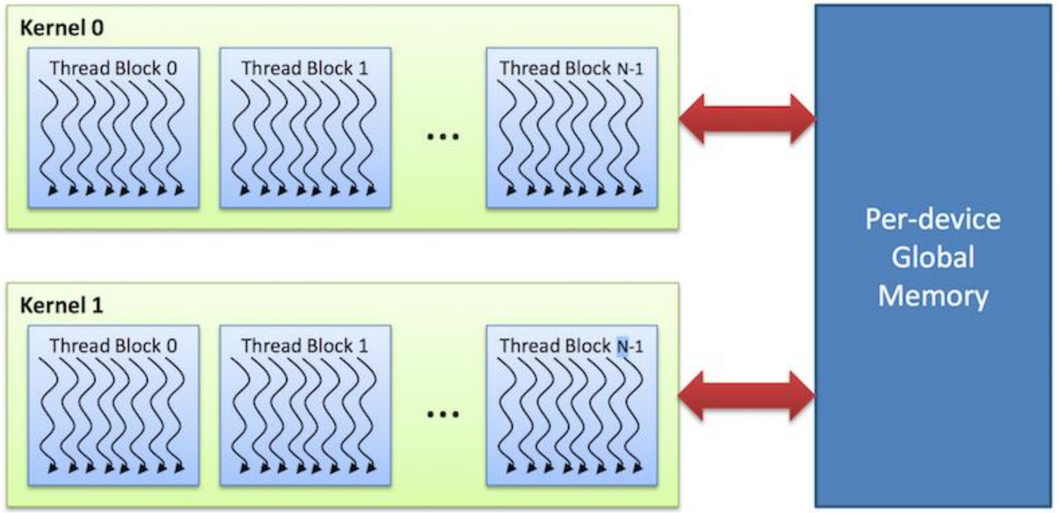




Global Memory

Global Memory:

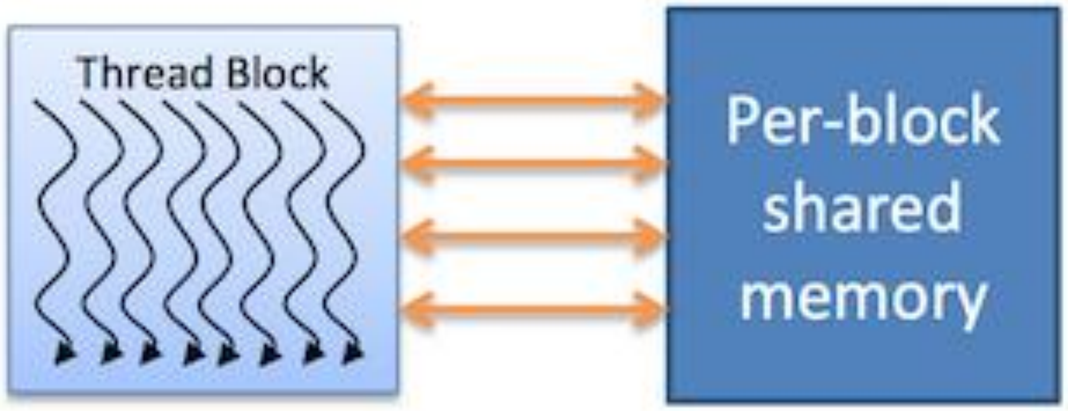
- **Scope:** Accessible by all threads across all thread blocks and by the host (CPU).
- **Lifetime:** Exists for the duration of the application; persists between kernel launches.
- **Performance:** High latency and lower bandwidth compared to other memory types; careful management is required to avoid performance bottlenecks.



Shared Memory

Shared Memory:

- **Scope:** Shared among threads within the same thread block; not accessible by threads in other blocks.
- **Lifetime:** Exists for the duration of the thread block.
- **Performance:** On-chip memory with low latency and high bandwidth; significantly faster than global memory.



Local/Private Memory

Local Memory:

- **Scope:** Private to each thread; used for variables that cannot be stored in registers.
- **Lifetime:** Exists for the duration of the thread.
- **Performance:** Resides in global memory space; accessing local memory can be as slow as accessing global memory.

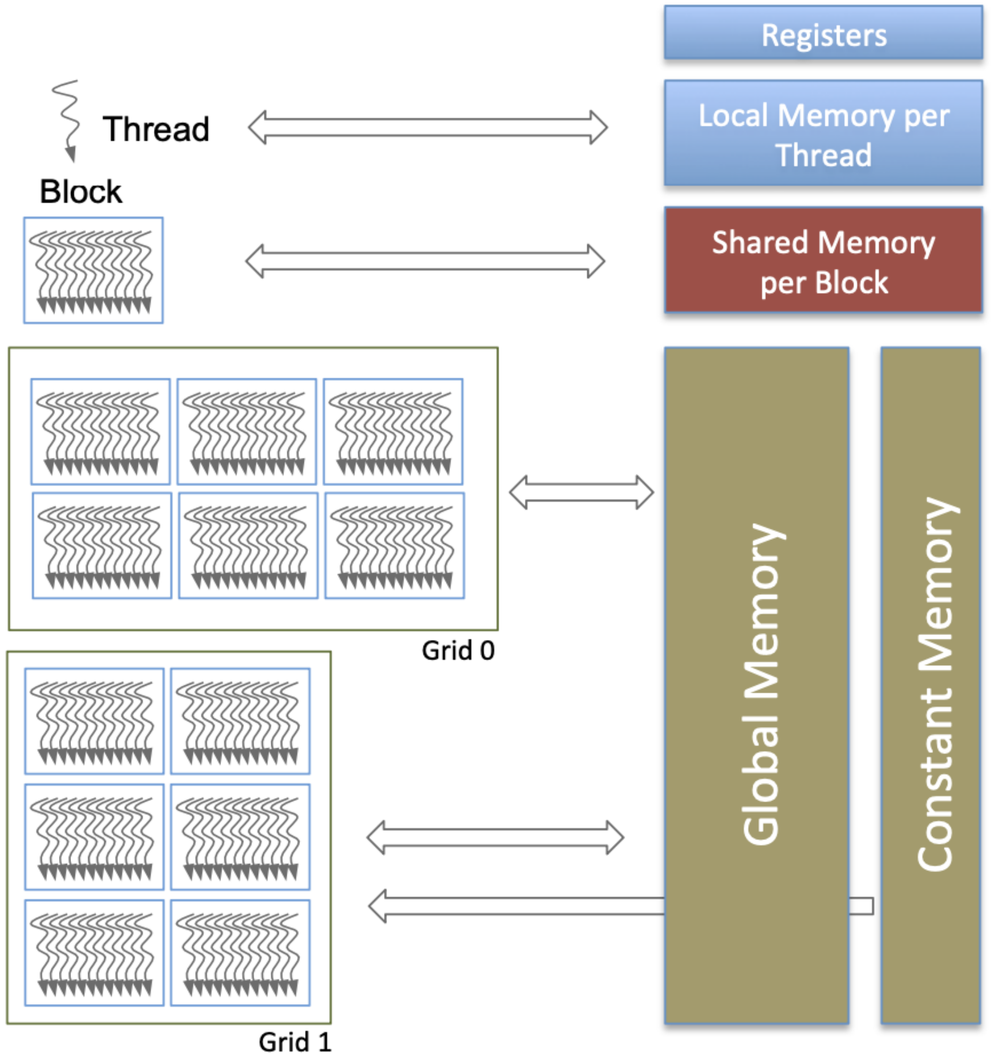
Memory Hierarchy

Private memory
Visible only to the thread

Shared memory
Visible to all the threads in a block

Global memory
Visible to all the threads
Visible to host
Accessible to multiple kernels
Data is stored in row major order

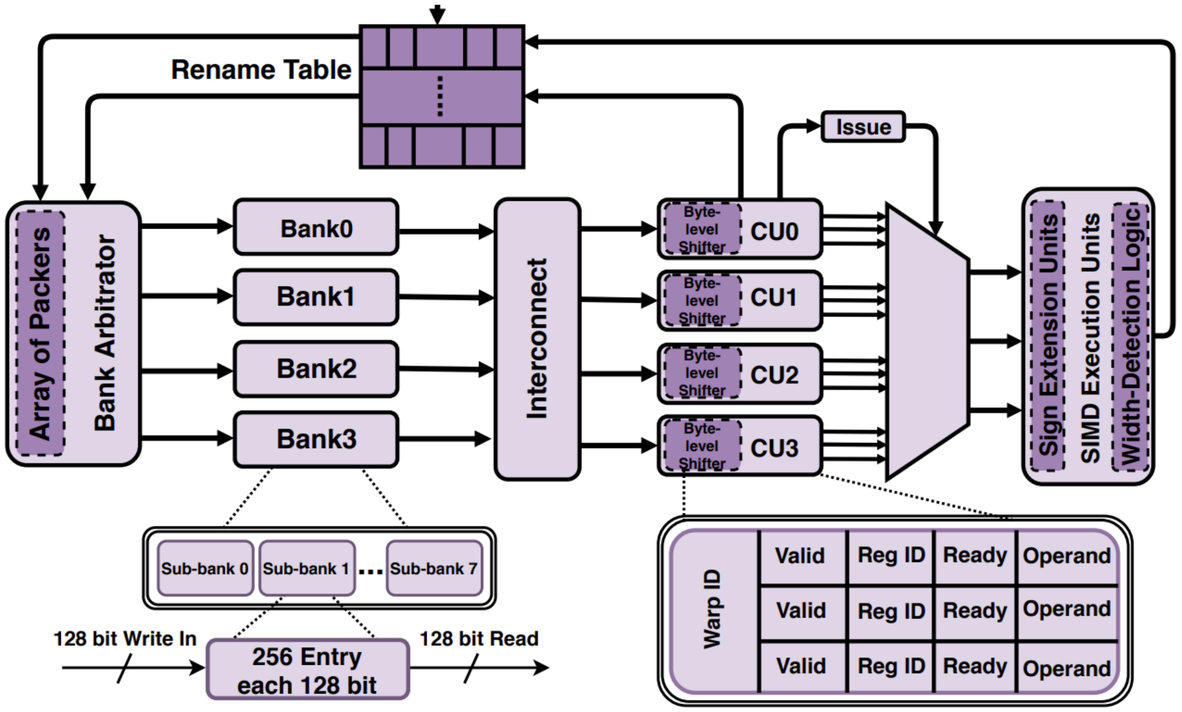
Constant memory (Read Only)
Visible to all the threads in a block



Registers

GPU registers are fast, on-chip memory in NVIDIA GPUs, storing thread-specific data (e.g., variables, results) for rapid access during execution.

- **Characteristics:** Each thread gets its own registers (32-255 per thread), with large register files per SM (e.g., 128K in Ampere). Access is extremely fast (1-2 cycles).
- **Organization:** Registers are allocated per thread within SMs, managed dynamically by the CUDA compiler and hardware scheduler.
- **Limitations:** Fixed register count per SM limits active threads; excess usage causes spilling to slower memory, reducing performance.
- **Performance Impact:** High register use lowers thread occupancy, affecting parallelism. Optimizing usage (e.g., minimizing variables) improves efficiency.
- **CUDA Context:** Registers are implicit in CUDA kernels; their usage influences block scheduling and overall GPU utilization.



Intra-SM Memory

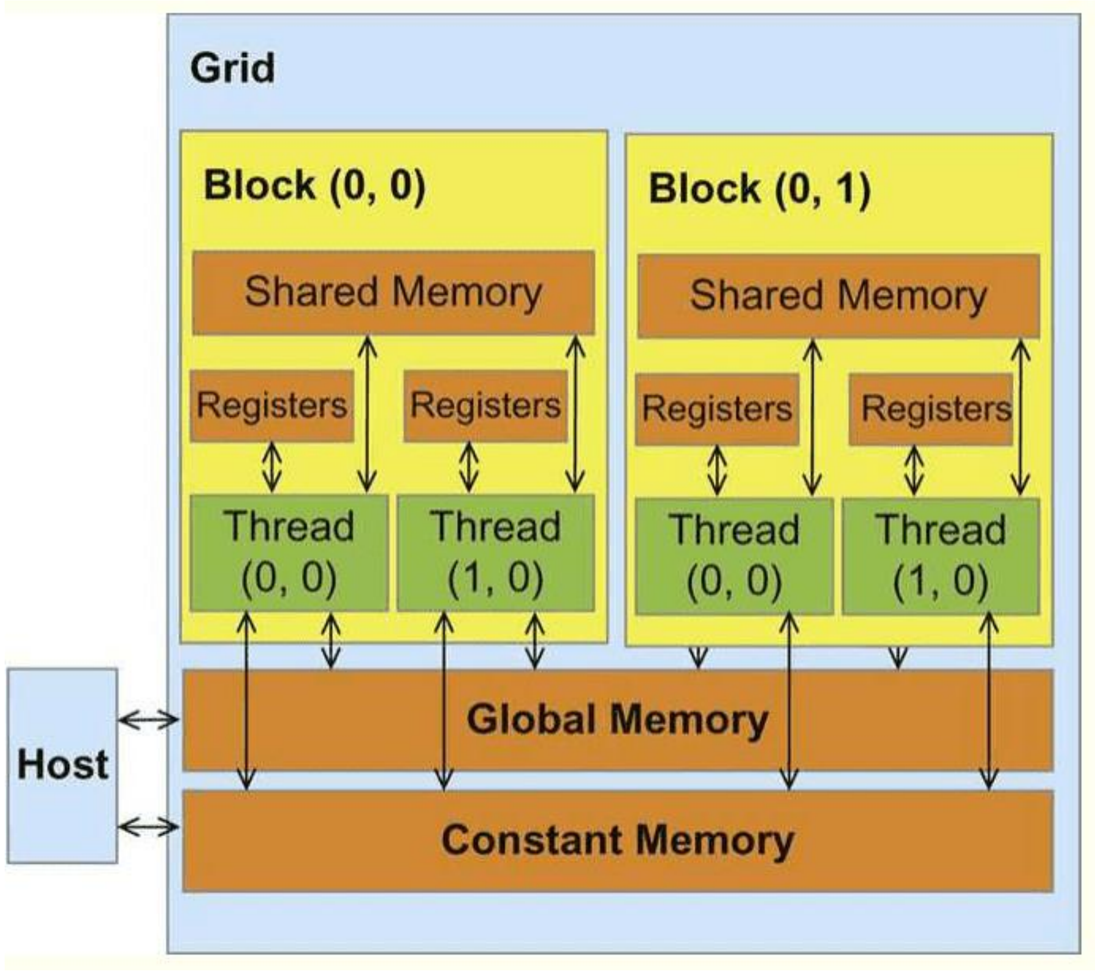
- **Register File** - denotes the area of memory that feeds directly into the CUDA cores. Accordingly, it is organized into 32 banks, matching the 32 threads in a warp. Think of the register file as a big matrix of 4-byte elements, having many rows and 32 columns. A warp operates on full rows; within a given row, each thread (CUDA core) operates on a different column (bank).
- **L1 Cache** - refers to the usual on-chip storage location providing fast access to data that are recently read from, or written to, main memory (RAM). Additionally, L1 serves as the overflow region when the amount of active data exceeds what an SM's register file can hold, a condition which is termed "register spilling". In L1, the cache lines and spilled registers are organized into banks, just as in the register file.
- **Shared Memory** - is a memory area that physically resides in the same memory as the L1 cache, but differs from L1 in that all its data may be accessed by any thread in a thread block. This allows threads to communicate and share data with each other. Variables that occupy it must be declared explicitly by an application. The application can also set the dividing line between L1 and shared memory.
- **Constant Caches** - are special caches pertaining to variables declared as read-only constants in global memory. Such variables can be read by any thread in a thread block. The main and best use of these caches is to broadcast a single constant value to all the threads in a warp

Inter-SM Memory

- **L2 Cache** - is a further on-chip cache for retaining copies of the data that travel back and forth between the SMs and main memory. Like the L1, the L2 cache is intended to speed up subsequent reloads. But unlike the L1 cache(s), there is just one L2 that is shared by all the SMs. The L2 cache is also situated in the path of data moving on or off the device via PCIe or NVLink.
- **Global Memory** - represents the bulk of the main memory of the device, equivalent to RAM in a CPU-based processor. For performance reasons, the Tesla V100 has special HBM2 high-bandwidth memory, while the Quadro RTX 5000 has fast GDDR6 graphics memory.
- **Local Memory** - corresponds to specially mapped regions of main memory that are assigned to each SM. Whenever "register spilling" overflows the L1 cache on a particular SM, the excess data are further offloaded to L2, then to "local memory". The performance penalty for reloading a spilled register becomes steeper for every memory level that must be traversed in order to retrieve it.
- **Texture and Constant Memory** - are regions of main memory that are treated as read-only by the device. When fetched to an SM, variables with a "texture" or "constant" declaration can be read by any thread in a thread block, much like shared memory. Texture memory is cached in L1, while constant memory is cached in the constant caches.

Memory Model

Concept	Description	Memory Access & Sharing
Grid	A collection of thread blocks that execute a kernel function. Grids can be one, two, or three-dimensional, allowing for flexible organization of thread blocks.	Grids do not directly access or share memory; they serve as a container for thread blocks. Each thread within the grid can access global memory.
Block	A group of threads that can cooperate among themselves through shared memory and synchronization. Blocks can be one, two, or three-dimensional.	Threads within a block share access to **shared memory**, which is a fast, on-chip memory. This allows for efficient data exchange and synchronization among threads in the same block. Blocks do not share memory with other blocks. DEVELOPER.NVIDIA.COM
Warp	The basic unit of execution in a Streaming Multiprocessor (SM), consisting of 32 threads that execute instructions in lockstep.	Threads within a warp utilize the shared memory of their parent block. Efficient memory access patterns within a warp can lead to performance improvements. EN.WIKIPEDIA.ORG
Thread	The smallest unit of execution, representing a single sequence of instructions. Each thread has a unique ID within its block.	Each thread has its own local memory and registers. It can access: **Local Memory:** Private to the thread.**Shared Memory:** Shared among threads within the same block.**Global Memory:** Accessible by all threads across the grid. DEVELOPER.NVIDIA.COM



CPU vs. GPU

in GPU programming, the best way to avoid the high latency penalty associated with global memory is to launch very large numbers of threads. That way, at least one warp is able to grab its next instruction from the instruction buffer and go, whenever another warp is stalled waiting for data. This technique is known as latency hiding.

Imagine a block with 128 threads, split into 4 warps (32 threads each):

- Warp 0 might be executing a floating-point operation.
- Warp 1 might be fetching data from global memory (stalled).
- Warp 2 might be performing an integer operation.
- Warp 3 might be idle or waiting for a branch resolution.

The SM can execute instructions from Warp 0 and Warp 2 in parallel on different execution units while Warp 1 waits. This is how parallelism across warps is achieved.

CPU vs. GPU

Memory Type	NVIDIA Tesla V100[1] *- per SM -*		Intel Cascade Lake SP, Skylake SP[2] *- per core -*	
	Latency	Bandwidth	Latency	Bandwidth
L1 cache	28 cycles	128 B/cycle	**4 cycles**	192 B/cycle
Private L2 cache	- N/A -	- N/A -	**14 cycles**	64 B/cycle
Shared L2 or L3	193 cycles	17.6 B/cycle	**50–70 cycles**	14.3 B/cycle
Global or RAM	220–350 cycles	**7.4 B/cycle**	190–220 cycles	1.9–2.5 B/cycle

Latency and available bandwidth per SM or core at each level in the memory hierarchy of the NVIDIA Tesla V100 vs. Intel Xeon SP processors.

Memory Type	NVIDIA Tesla V100 *- per SM -*	Intel Cascade Lake SP, Skylake SP *- per core -*
Register file	**256 kB**	10.5 kB
L1 cache	**128 kB** (max)	32 kB
Constant caches	**64 kB**	- N/A -
L2 cache	0.075 MB	**1 MB**
L3 cache	- N/A -	**1.375 MB**
Global or RAM	0.4 GB	**>3.4 GB**

Available memory per SM or core at each level in the memory hierarchy of the NVIDIA Tesla V100 vs. Intel Xeon SP processors.

Register file: In the NVIDIA Tesla V100, the register file of an SM stores (65536)/(2x32) = 1024 floats per CUDA core. In the Intel Cascade Lake and Skylake chips, a CPU core has a physical register file that stores 168 vector registers of 64B, or 10.5 kB in total. This works out to (16x168)/(2x16) = 84 floats per vector lane, which is an order of magnitude less than what is available to a CUDA core. (The factors of 2 appear because a CPU core can handle 2x16-float vectors/cycle, and a GPU SM can handle 2x32-float warps/cycle.)

CPU vs. GPU

Memory Type	NVIDIA Tesla V100[1] *- per SM -*		Intel Cascade Lake SP, Skylake SP[2] *- per core -*	
	Latency	Bandwidth	Latency	Bandwidth
L1 cache	28 cycles	128 B/cycle	**4 cycles**	192 B/cycle
Private L2 cache	- N/A -	- N/A -	**14 cycles**	64 B/cycle
Shared L2 or L3	193 cycles	17.6 B/cycle	**50–70 cycles**	14.3 B/cycle
Global or RAM	220–350 cycles	**7.4 B/cycle**	190–220 cycles	1.9–2.5 B/cycle

Latency and available bandwidth per SM or core at each level in the memory hierarchy of the NVIDIA Tesla V100 vs. Intel Xeon SP processors.

Memory Type	NVIDIA Tesla V100 *- per SM -*	Intel Cascade Lake SP, Skylake SP *- per core -*
Register file	**256 kB**	10.5 kB
L1 cache	**128 kB** (max)	32 kB
Constant caches	**64 kB**	- N/A -
L2 cache	0.075 MB	**1 MB**
L3 cache	- N/A -	**1.375 MB**
Global or RAM	0.4 GB	**>3.4 GB**

Available memory per SM or core at each level in the memory hierarchy of the NVIDIA Tesla V100 vs. Intel Xeon SP processors.

Cache sizes: In the NVIDIA Tesla V100, an SM has 128 kB (max) in its L1 data cache, and 64 kB in its constant cache. Adding these to its share of the 6 MB shared L2 (6/80 = 0.075 MB) yields 0.26 MB per SM, or 0.0041 MB per CUDA core. In the Intel Cascade Lake and Skylake chips, a CPU core has 32 kB in its L1 plus 1 MB in its L2 data cache. Adding these to its share of the shared L3 cache (1.375 MB) yields 2.4 MB, or 0.75 MB per vector lane, which is two orders of magnitude more than what is available to a CUDA core.

CPU vs. GPU

- **GPU large registers**: Support massive parallel threads simultaneously active.
- **CPU small registers**: Optimized for fewer, high-performance threads with complex logic.
- **GPU small caches**: Rely on parallelism to hide latency, limited benefit from temporal reuse.
- **CPU large caches**: Rely heavily on locality and fast repeated access to small working datasets.

Let’s tackle the problem step by step.

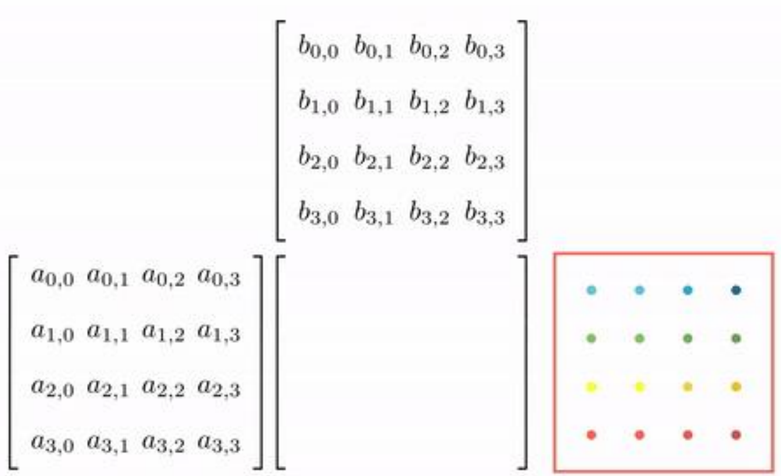
–Tom Jerry

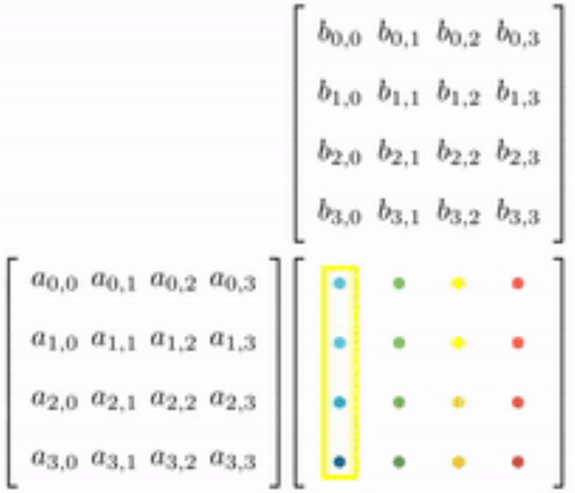
Matrix Multiplication

$$\begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,n-1} \end{bmatrix}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

1. Naive Parallelism

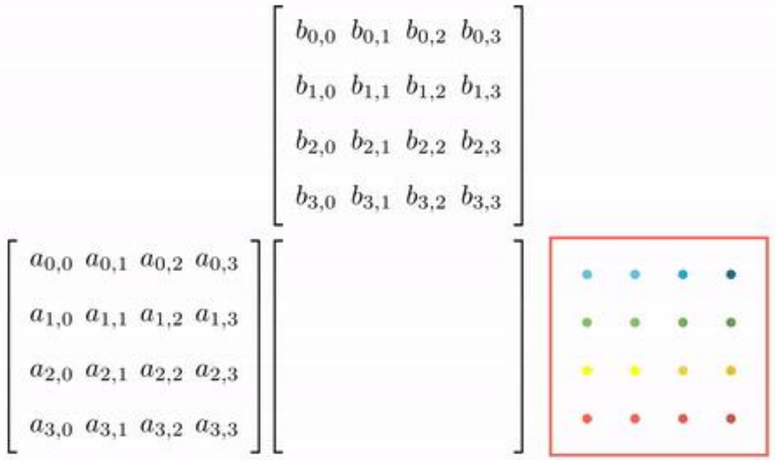


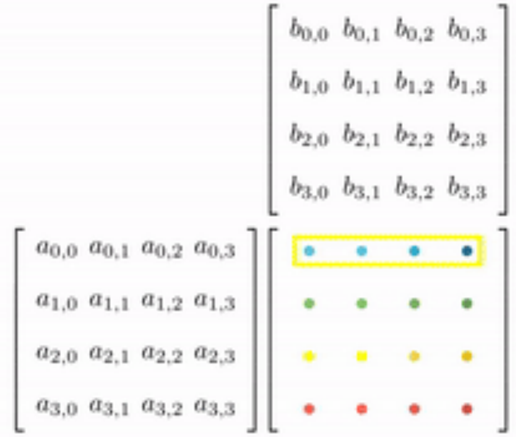


1. Naive Parallelism

```
__global__ void naive_mat_mul_kernel(float *d_A_ptr, float *d_B_ptr,
                                     float *d_C_ptr, int C_n_rows, int C_n_cols, int A_n_cols)
{
    // Working on C[row,col]
    const int row = blockDim.x*blockIdx.x + threadIdx.x;
    const int col = blockDim.y*blockIdx.y + threadIdx.y;

    // Parallel mat mul
    if (row < C_n_rows && col < C_n_cols)
    {
        // Value at C[row,col]
        float value = 0;
        for (int k = 0; k < A_n_cols; k++)
        {
            value += d_A_ptr[row*A_n_cols + k] * d_B_ptr[k*C_n_cols + col];
        }

        // Assigning calculated value (SGEMM is C = α*(A @ B)+β*C and in this repo α=1, β=0)
        d_C_ptr[row*C_n_cols + col] = 1*value + 0*d_C_ptr[row*C_n_cols + col];
    }
}
```

2. Memory Coalescing: Access Global Memory Wisely





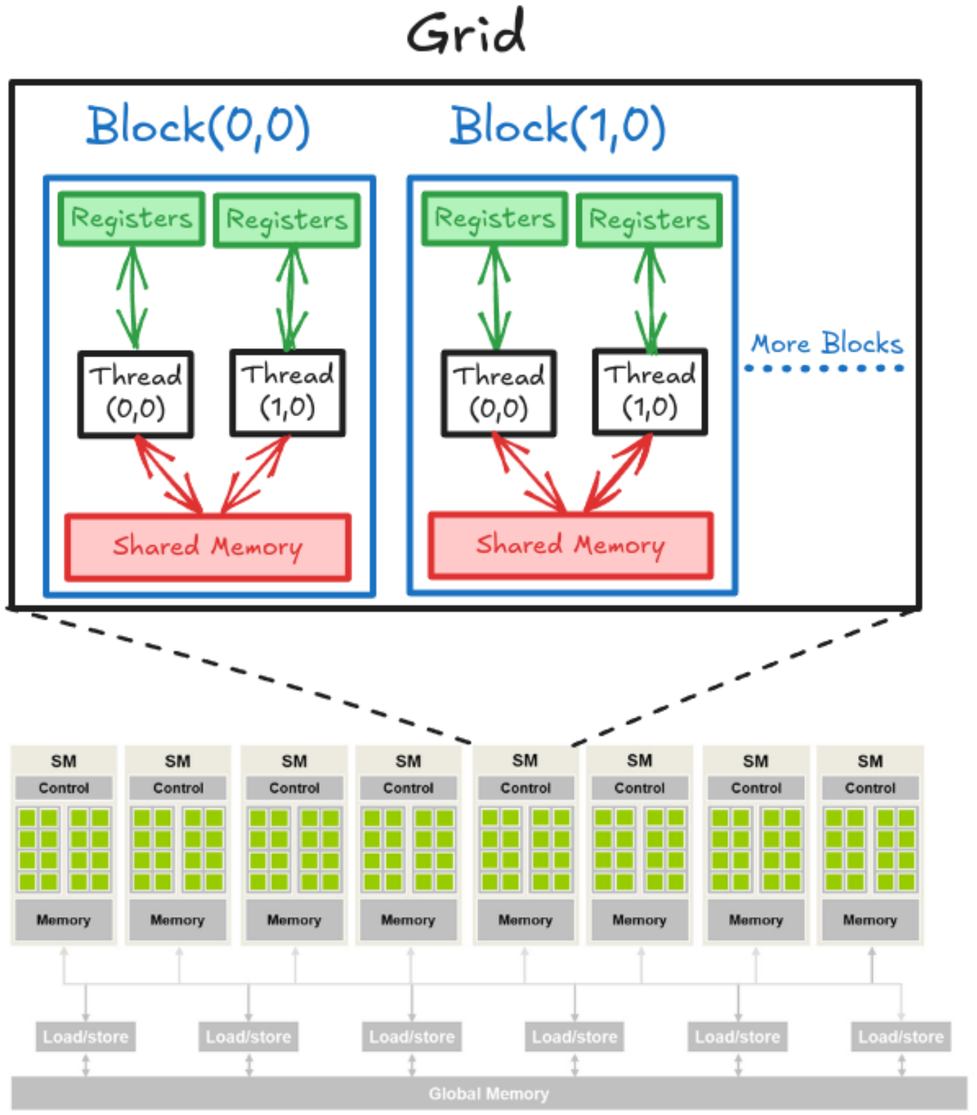
2. Memory Coalescing: Access Global Memory Wisely

```
__global__ void coalesced_mat_mul_kernel(float *d_A_ptr, float *d_B_ptr,
                                         float *d_C_ptr, int C_n_rows, int C_n_cols, int A_n_cols)
{
    // Working on C[row,col]
    const int col = blockDim.x*blockIdx.x + threadIdx.x;
    const int row = blockDim.y*blockIdx.y + threadIdx.y;

    // Parallel mat mul
    if (row < C_n_rows && col < C_n_cols)
    {
        // Value at C[row,col]
        float value = 0;
        for (int k = 0; k < A_n_cols; k++)
        {
            value += d_A_ptr[row*A_n_cols + k] * d_B_ptr[k*C_n_cols + col];
        }

        // Assigning calculated value (SGEMM is C = α*(A @ B)+β*C and in this repo α=1, β=0)
        d_C_ptr[row*C_n_cols + col] = 1*value + 0*d_C_ptr[row*C_n_cols + col];
    }
}
```

3. Shared Memory Through Tiling

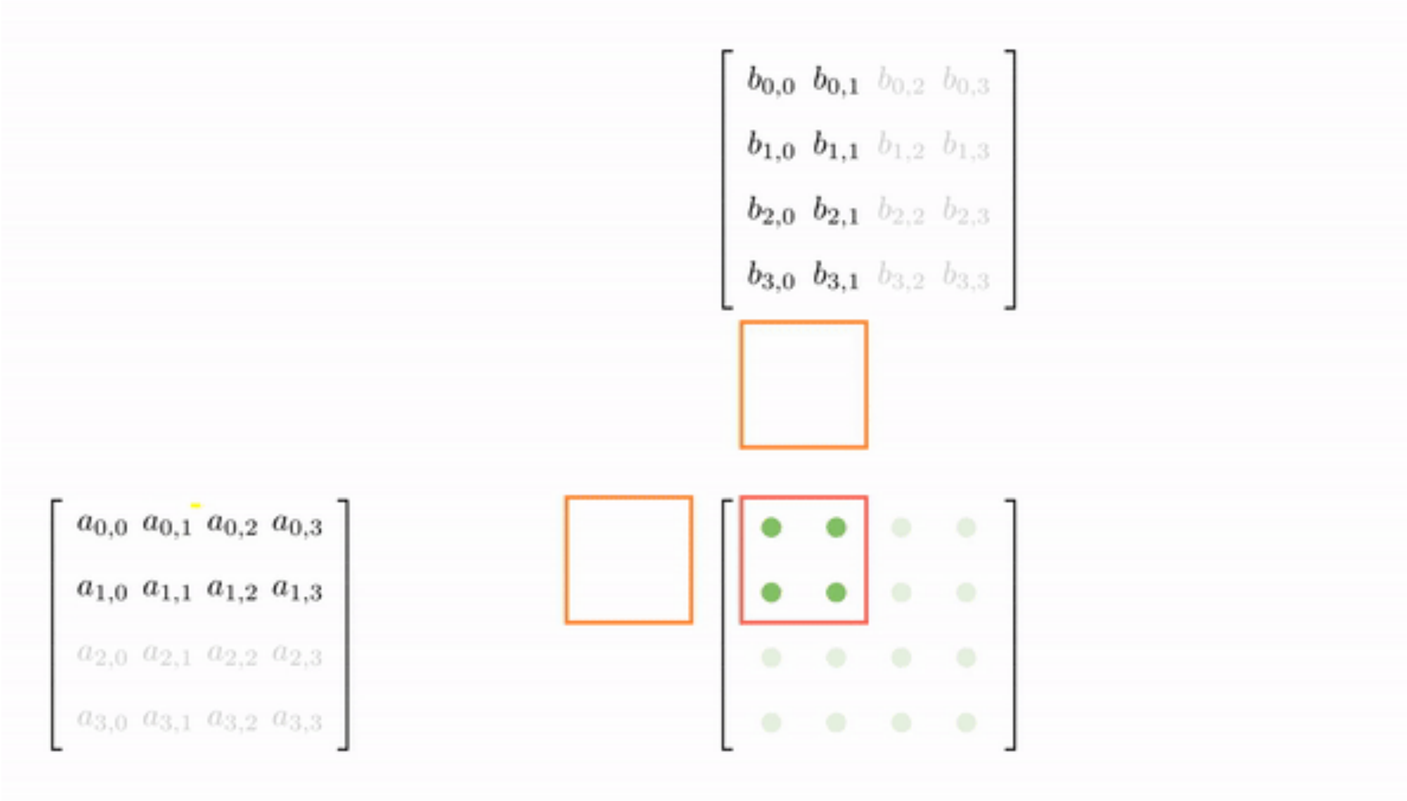
- **Shared Memory:** is a small memory space (~16KB per SM) that resides on-chip and has a short latency with high bandwidth. On a software level, it can only be written and read by the threads within a block.

- To avoid multiple global memory accesses, we can partition the data into subsets called tiles so each tile fits into the shared memory and then perform multiple operations on this data. Shared memory is small, so we can only move very small subsets of data to and from shared memory (one at a time).



3. Shared Memory Through Tiling

k	thread (0,0)	thread (0,1)	thread (1,0)	thread (1,1)
0	A(0,0) B(0,0)	A(0,0) B(0,1)	A(1,0) B(0,0)	A(1,0) B(0,1)
1	A(0,1) B(1,0)	A(0,1) B(1,1)	A(1,1) B(1,0)	A(1,1) B(1,1)
2	A(0,2) B(2,0)	A(0,2) B(2,1)	A(1,2) B(2,0)	A(1,2) B(2,1)



3. Shared Memory Through Tiling

phase	thread (0,0)	thread (0,1)	thread (1,0)	thread (1,1)
0	A(0,0) B(0,0)	A(0,1) B(0,1)	A(1,0) B(1,0)	A(1,1) B(1,1)
1	A(0,2) B(2,0)	----- B(2,1)	A(1,2) -----	----- -----

Original Version

$$C[0,0] = \overbrace{A[0,0] \cdot B[0,0]}^{k=0} + \overbrace{A[0,1] \cdot B[1,0]}^{k=1} + \overbrace{A[0,2] \cdot B[2,0]}^{k=2} + \overbrace{A[0,3] \cdot B[3,0]}^{k=3}$$

Tiled Version

$$C[0,0] = \overbrace{\overbrace{A[0,0] \cdot B[0,0]}^{k_{phase}=0} + \overbrace{A[0,1] \cdot B[1,0]}^{k_{phase}=1}}^{phase=0} + \overbrace{\overbrace{A[0,2] \cdot B[2,0]}^{k_{phase}=0} + \overbrace{A[0,3] \cdot B[3,0]}^{k_{phase}=1}}^{phase=1}$$

3. Shared Memory Through Tiling

Tiling: All threads in a block can access all elements for their computations. As data copying into shared memory is done in parallel by multiple threads (and it's coalesced), we must ensure the complete tile is loaded before moving forward! This is done using __syncthreads() which basically holds the code execution (at this point) until all threads have reached there.

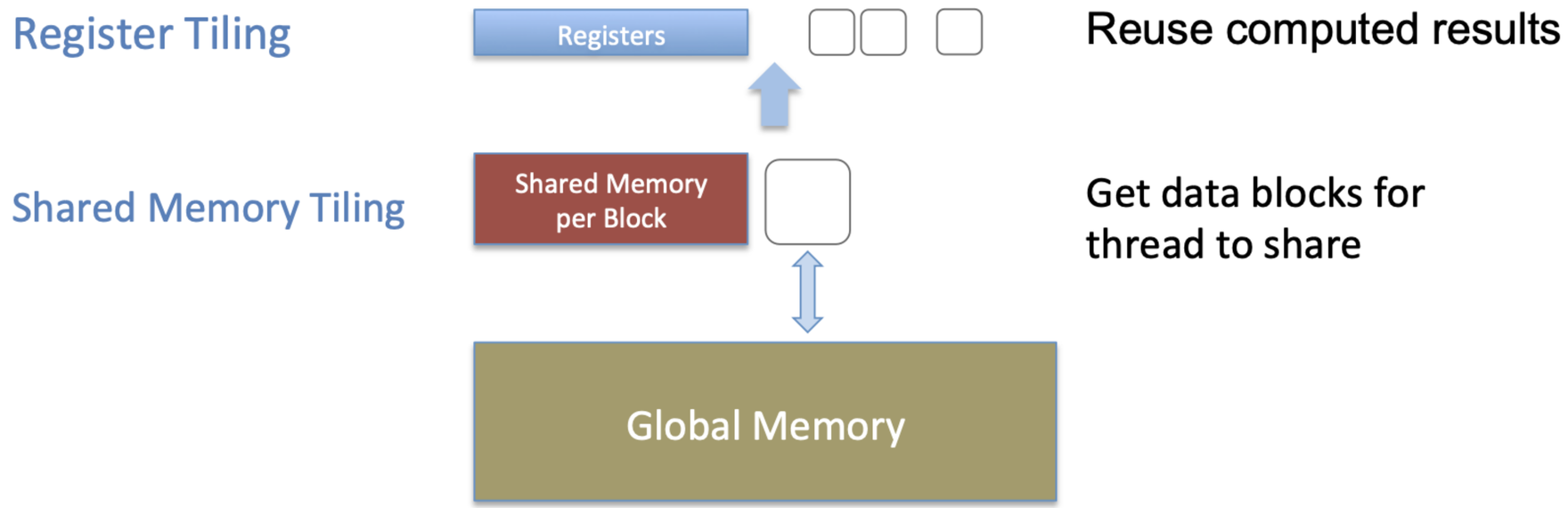
```
// Details regarding this thread
const int by = blockIdx.y;
const int bx = blockIdx.x;

const int ty = threadIdx.y;
const int tx = threadIdx.x;

// Working on C[row,col]
const int row = TILE_WIDTH*by + ty;
const int col = TILE_WIDTH*bx + tx;

// Allocating shared memory
__shared__ float sh_A[TILE_WIDTH][TILE_WIDTH];
__shared__ float sh_B[TILE_WIDTH][TILE_WIDTH];
```

```
// Parallel mat mul
float value = 0;
for (int phase = 0; phase < phases; phase++)
{
    // Load Tiles into shared memory
    if ((row < C_n_rows) && ((phase*TILE_WIDTH+tx) < A_n_cols))
      sh_A[ty][tx] = d_A_ptr[(row)*A_n_cols + (phase*TILE_WIDTH+tx)];
    else
      sh_A[ty][tx] = 0.0f;

    if (((phase*TILE_WIDTH + ty) < A_n_cols) && (col < C_n_cols))
      sh_B[ty][tx] = d_B_ptr[(phase*TILE_WIDTH + ty)*C_n_cols + (col)];
    else
      sh_B[ty][tx] = 0.0f;

    // Wait for all threads to load elements
    __syncthreads();

    // .
    // .
    // .
}
```

3. Shared Memory Through Tiling

Also known as Tiling.

Basic idea is to move blocks/tiles of commonly useable data from global memory into shared memory or registers memory.



3. Shared Memory Through Tiling

Focused Access pattern

Identify block/tile of global memory data to be accessed by threads.

Load the data into the fast memory (Shared, register)

Get the multithreads to use the data

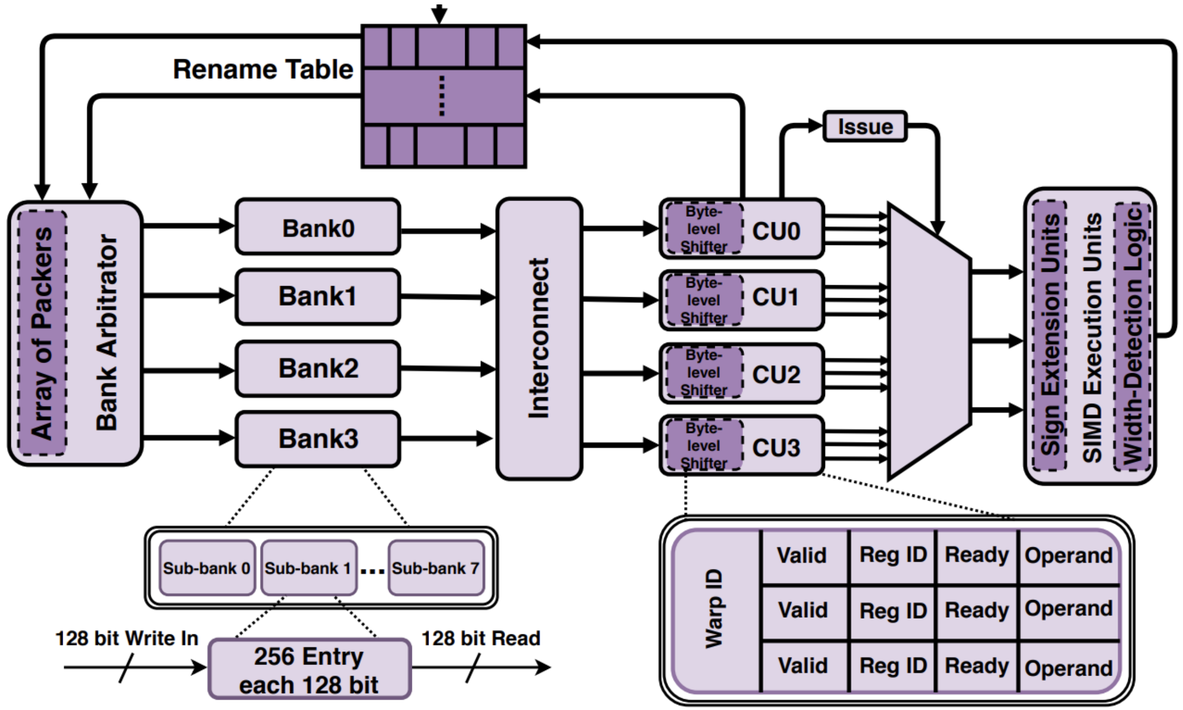
Assure barrier synchronization

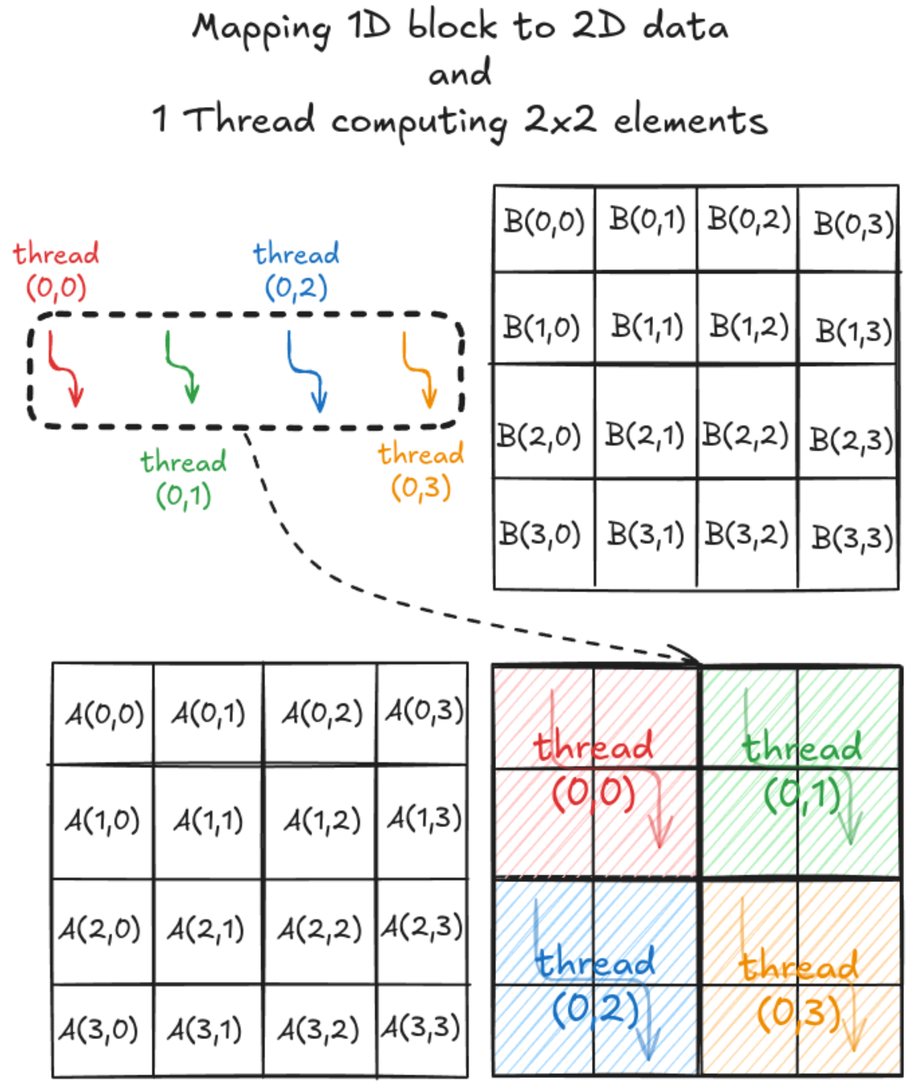
Repeat (move to next block, next iterations etc.)

Make the most of one load of data into fast memory

4. Use Registers Through Thread Coarsening

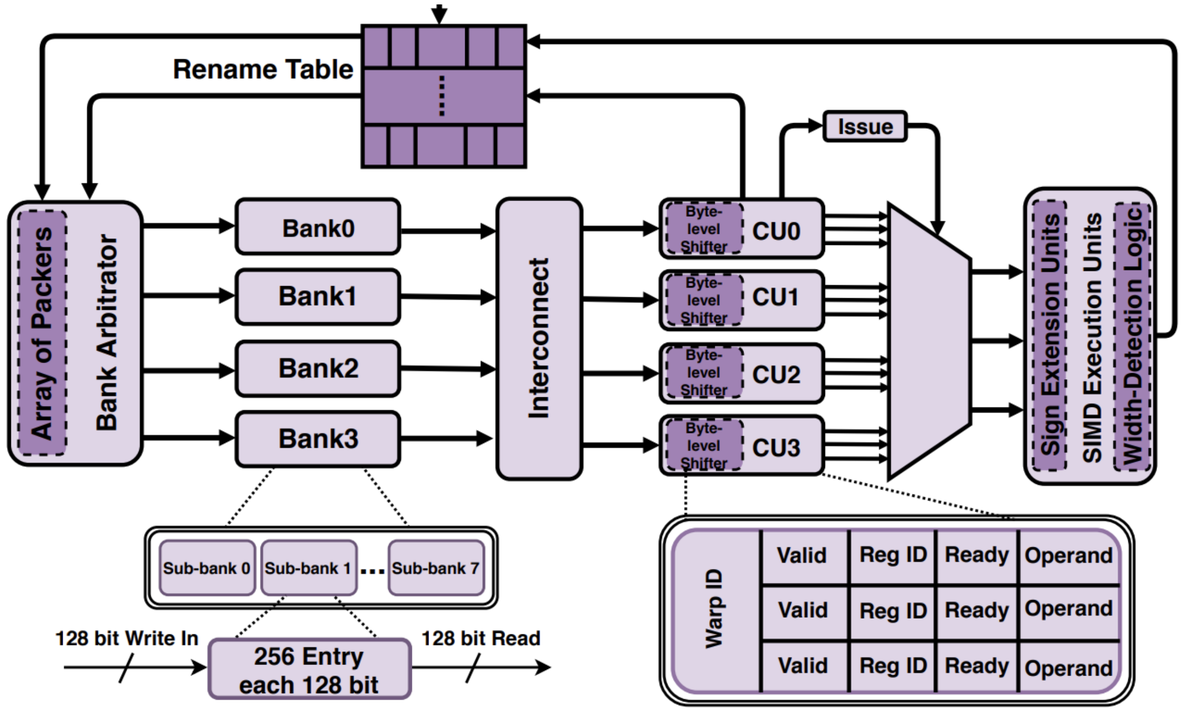
Registers: Data cannot be shared between threads.

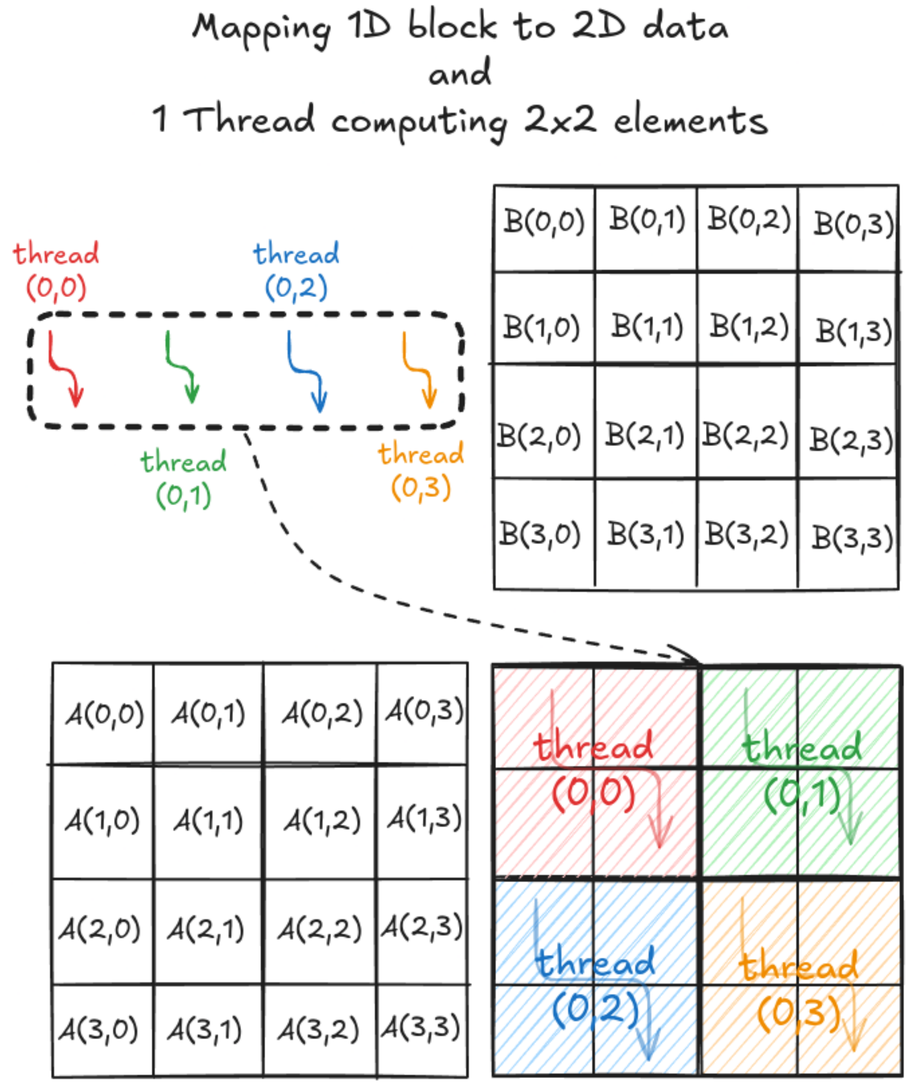




4. Use Registers Through Thread Coarsening

Registers: Data cannot be shared between threads.





4. Use Registers Through Thread Coarsening

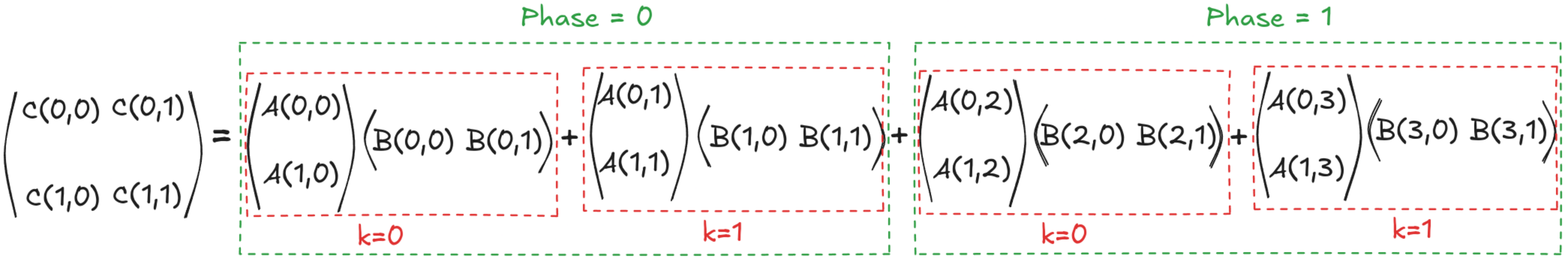
$$C[0,0] = \overbrace{\overbrace{A[0,0] \cdot B[0,0]}^{k_{phase}=0} + \overbrace{A[0,1] \cdot B[1,0]}^{k_{phase}=1}}^{phase=0} + \overbrace{\overbrace{A[0,2] \cdot B[2,0]}^{k_{phase}=0} + \overbrace{A[0,3] \cdot B[3,0]}^{k_{phase}=1}}^{phase=1}$$

$$C[0,1] = \underbrace{\underbrace{A[0,0] \cdot B[0,1]}_{k_{phase}=0} + \underbrace{A[0,1] \cdot B[1,1]}_{k_{phase}=1}}_{phase=0} + \underbrace{\underbrace{A[0,2] \cdot B[2,1]}_{k_{phase}=0} + \underbrace{A[0,3] \cdot B[3,1]}_{k_{phase}=1}}_{phase=1}$$

$$C[1,0] = \underbrace{\underbrace{A[1,0] \cdot B[0,0]}_{k_{phase}=0} + \underbrace{A[1,1] \cdot B[1,0]}_{k_{phase}=1}}_{phase=0} + \underbrace{\underbrace{A[1,2] \cdot B[2,0]}_{k_{phase}=0} + \underbrace{A[1,3] \cdot B[3,0]}_{k_{phase}=1}}_{phase=1}$$

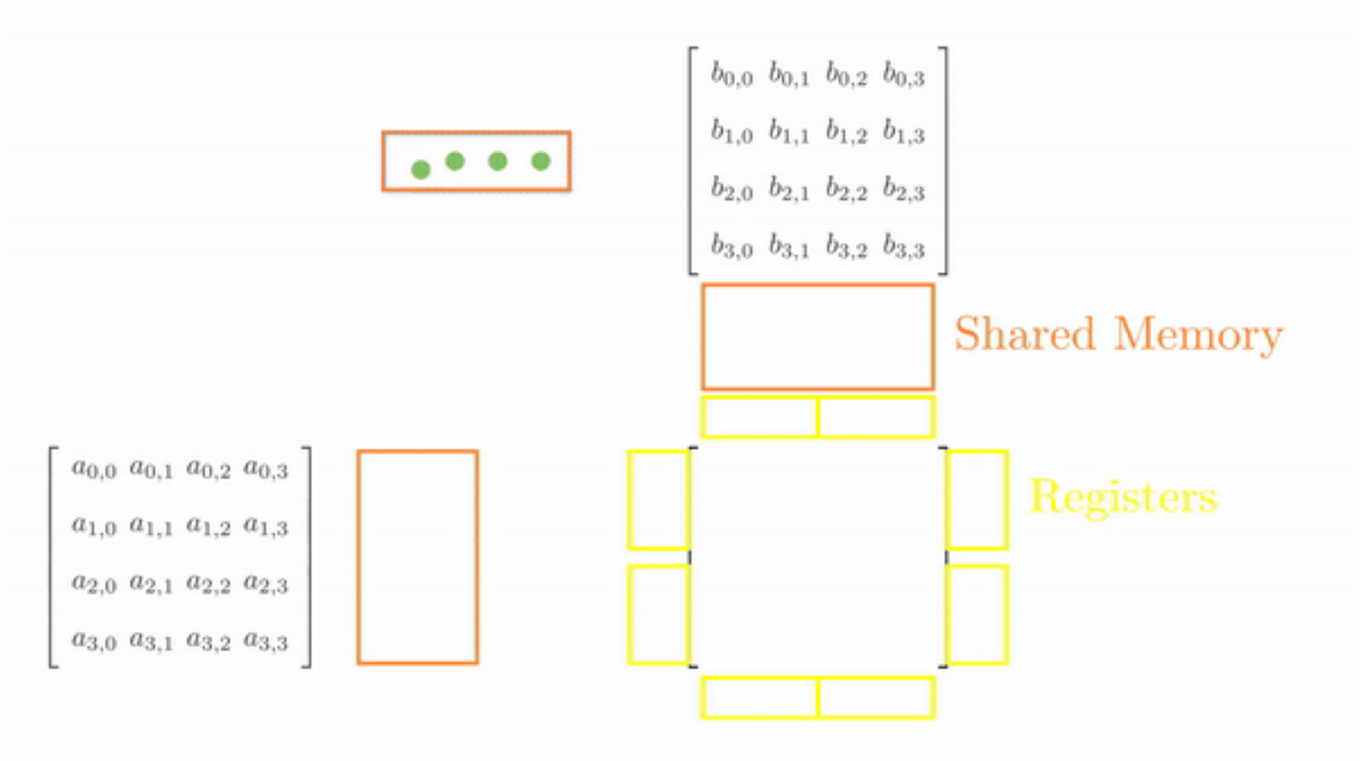
$$C[1,1] = \underbrace{\underbrace{A[1,0] \cdot B[0,1]}_{k_{phase}=0} + \underbrace{A[1,1] \cdot B[1,1]}_{k_{phase}=1}}_{phase=0} + \underbrace{\underbrace{A[1,2] \cdot B[2,1]}_{k_{phase}=0} + \underbrace{A[1,3] \cdot B[3,1]}_{k_{phase}=1}}_{phase=1}$$

4. Use Registers Through Thread Coarsening



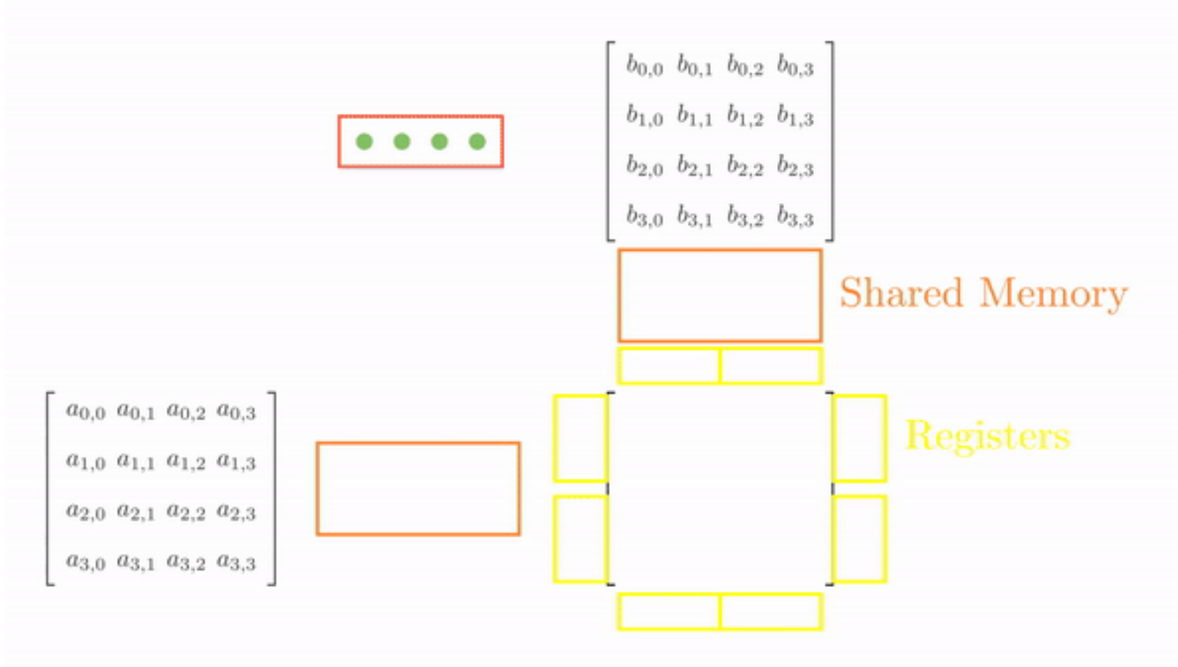
Elements of A and B are stored in Registers

4. Use Registers Through Thread Coarsening



5. Vectorized Memory Access

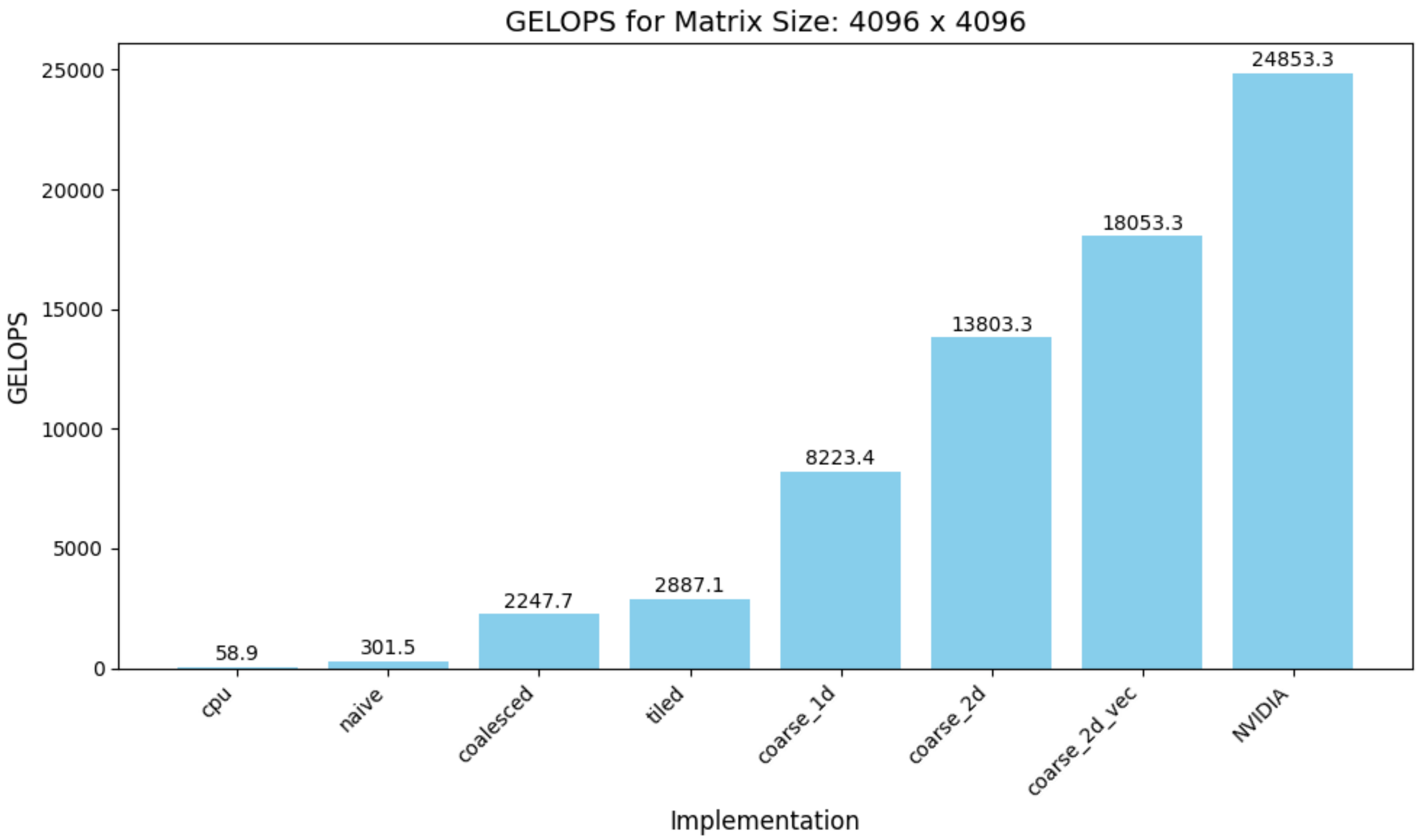
In the context of GPU-accelerated matrix multiplication, **vectorized memory access** refers to the technique of transferring multiple contiguous data elements simultaneously using a single instruction. This approach leverages the hardware's capability to handle data in chunks, thereby reducing the number of memory transactions and improving overall performance.



Implementation Details:

1. **Data Contiguity:** For vectorized access to be effective, the data elements must be stored contiguously in memory. In matrix operations, this often involves ensuring that rows or columns of the matrix are laid out sequentially in memory.
2. **Data Transfer Using Vector Types:** In CUDA, data can be loaded and stored using vector types such as `float4`, which represents four `float` values packed together. By casting pointers to these vector types, a single instruction can move multiple data elements.

```
// Assuming d_A is a pointer to the global memory array
float4* vec_ptr = reinterpret_cast<float4*>(d_A + index);
float4 data = vec_ptr[0];
```


GELOPS for Matrix Size: 4096 x 4096
GELOPS
Implementation
25000
20000
15000
10000
5000
0
58.9
301.5
2247.7
2887.1
8223.4
13803.3
18053.3
24853.3
cpu
naive
coalesced
tiled
coarse_1d
coarse_2d
coarse_2d_vec
NVIDIA

1. Naïve kernel (one thread = one element of C)

- Each thread computes one element $C[m, n]$.
- That means:

$$C[m, n] = \sum_{k=0}^{K-1} A[m, k] \cdot B[k, n]$$

- Spawns $M \times N$ threads.
- Each thread directly loads row of A and column of B from global memory.
- Problem: **very poor memory efficiency** (lots of redundant loads, strided column accesses).

2. Warp/thread organization

- Threads are grouped into **warps** (32 threads) and blocks.
- If warps are mapped “column-wise,” memory access is strided → bad.
- Mapping threads “row-wise” improves **coalescing** and reduces memory misses.
- So, the first important fix is **thread indexing layout** for better memory access.

3. Tiling with Shared Memory

- Avoid redundant global memory loads by **tiling** matrices A and B.
- Divide data into blocks that fit in **shared memory** (on-chip, low latency).
- Threads in a block **cooperatively load tiles** in a coalesced manner.
- Shared data is **reused across multiple threads**, greatly cutting global memory traffic.

4. Thread Coarsening & Register Blocking

- Boost **arithmetic intensity**: let each thread compute a **sub-block of** C (1D or 2D).
- Threads load small chunks of A and B into **registers** for repeated use.
- Benefits:
  - Lower per-thread overhead.
  - Higher compute density (**more FLOPs per memory access**).
  - Registers = **fastest memory** on the GPU.

5. Vectorized memory access

- Finally, optimize the **global ↔ shared memory transfers**:
  - Use `float4` (or similar) vectorized loads/stores → 4 contiguous floats in one instruction.
  - Ensure **alignment** and enforce tile sizes as multiples of 4.
  - Load row-wise (contiguous in memory), then **transpose into shared memory** so column accesses later are efficient.
- This maximizes global memory bandwidth utilization.