Machine Learning Systems

Build efficient and scalable ML services through the vertical integration of algorithms, system software, and hardware

Acknowledgement

Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community.

Part of the course material was created by LLM itself.

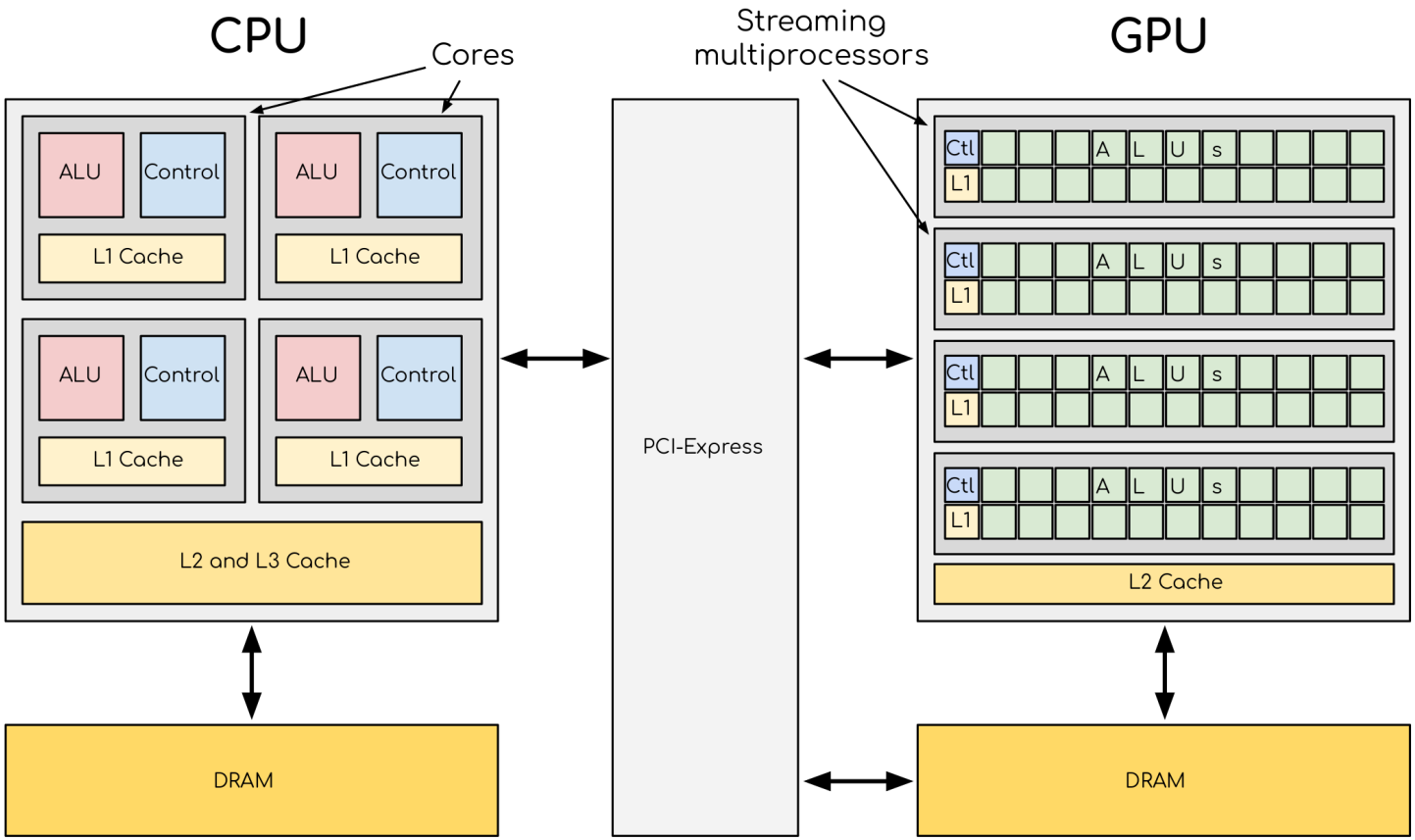
More importantly, for each and every of us, LLM shall be heavily involved in daily learning.

What kind of workload prefers GPUs?

–Tom Jerry

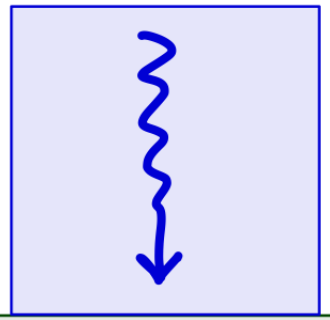
What kind of workload prefers GPUs?

Workload shall have two key characteristics: high parallelism and high arithmetic intensity



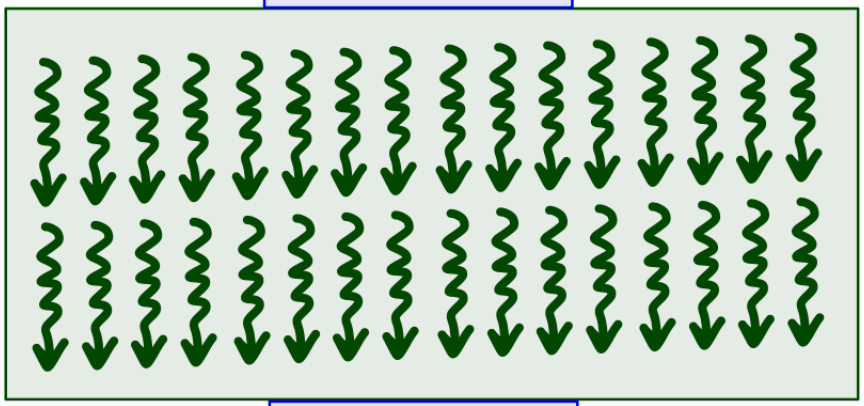
GPGPU as a really powerful accelerator

CPU thread

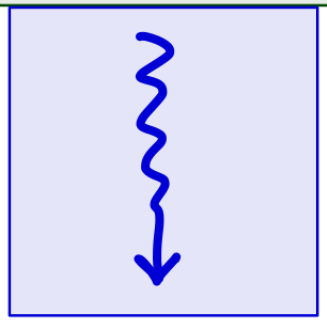


- Prepare data (matrices A, B, C).
- Copy data from RAM to VRAM.

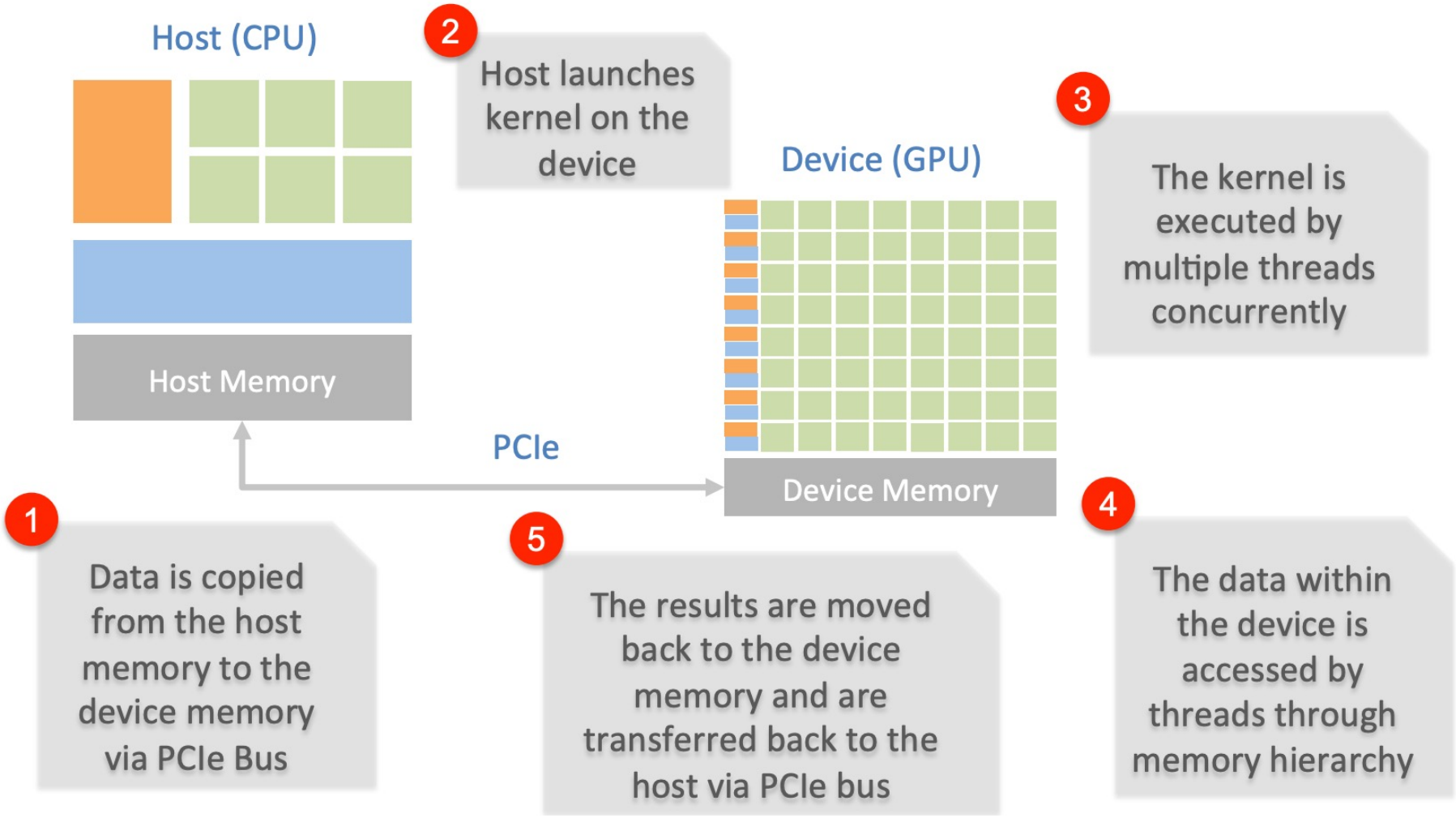
GPU threads



- Parallel Matrix Multiplication.



- Copy results from VRAM to RAM.


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

Where is the bottleneck?

–Tom Jerry

```cpp
void vectorAdd(const float* A, const float* B, float* C, int N) {
    for (int idx = 0; idx < N; idx++) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1 << 16; // 65536 elements
    size_t size = N * sizeof(float);

    // Allocate host memory using vectors (auto-managed)
    std::vector<float> A(N), B(N), C(N);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        A[i] = 1.0f;
        B[i] = 2.0f;
    }

    // Perform vector addition on CPU
    vectorAdd(A.data(), B.data(), C.data(), N);

    // Verify results (print first 5)
    for (int i = 0; i < 5; i++) {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }

    return 0;
}
```

1. Define CUDA Kernel

```cpp
__global__ void vectorAdd(const float *A, const float *B, float
*C, int N){…}
```

2. Allocate GPU memory

```cpp
cudaMalloc((void**)&d_A, size);
```

3. Copy data from CPU to GPU

```cpp
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

4. Launch CUDA kernel

```cpp
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

5. Copy data from GPU back to CPU

```cpp
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

6. Free resources

```cpp
cudaFree(d_A);
```

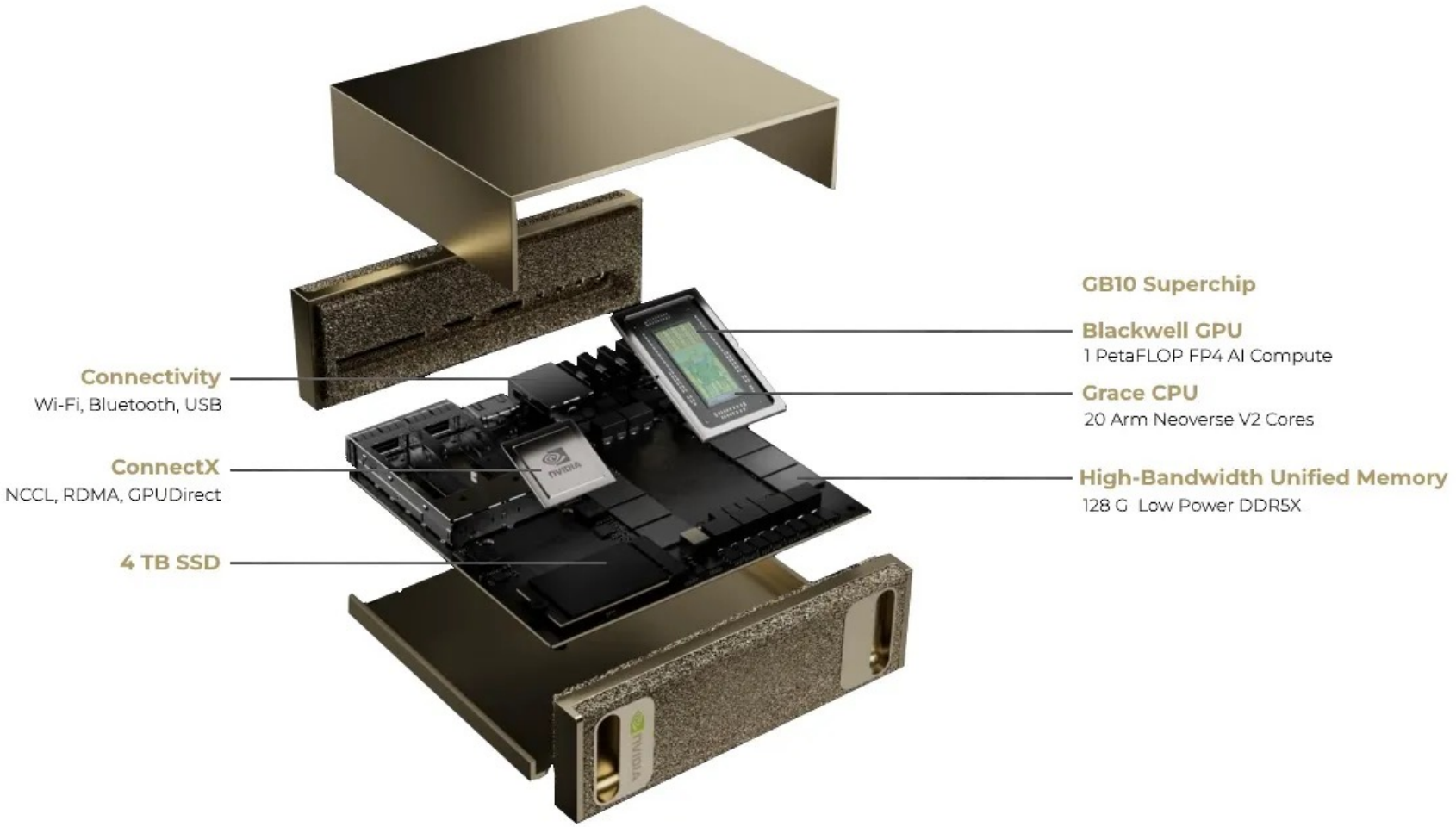
```
[ CPU Init ] → [ cudaMalloc ] → [ H2D memcpy ] → [ GPU Kernel ] → [ D2H memcpy ]
```

Size (MB)	Host Init (ms)	cudaMalloc (ms)	H2D (ms)	Kernel (ms)	D2H (ms)	Total (ms)	H2D BW (GB/s)	Kernel BW (GB/s)
256	1026.96	324.62	11.44	4.43	5.92	1373.49	46.95	181.72
4096	14118.40	770.36	191.25	58.79	90.33	15229.20	44.92	219.16

$$T_{\text{total}} = T_{\text{host}} + T_{\text{alloc}} + T_{\text{H2D}} + T_{\text{kernel}} + T_{\text{D2H}}$$

From 256 → 4096 MB (×16):

Component	Scaling
Host Init	~14×
cudaMalloc	~2.4×
H2D	~16×
Kernel	~13×
D2H	~15×


GB10 Superchip
Blackwell GPU
1 PetaFLOP FP4 AI Compute
Grace CPU
20 Arm Neoverse V2 Cores
High-Bandwidth Unified Memory
128 G Low Power DDR5X
Connectivity
Wi-Fi, Bluetooth, USB
ConnectX
NCCL, RDMA, GPUDirect
4 TB SSD

CUDA Memory Allocation: Mechanistic Model

$$T_{\text{alloc}}(S) = T_{\text{init}} + \alpha S, \quad T_{\text{free}}(S) = \beta S, \;\; \beta \ll \alpha$$

Empirical fit:

$$\alpha^{-1} \approx 30 \text{ GB/s}, \quad \beta^{-1} \approx 150 \text{ GB/s}$$

Interpretation

- Linear scaling (αS) ⇒ allocation performs **size-proportional memory-system work**
- α matches bandwidth ⇒ **physical backing + page-table setup (eager, not virtual-only)**
- $\beta \ll \alpha$ ⇒ free is **metadata-dominated (no full memory touch)**
- Large first-call latency ⇒ **CUDA context initialization fused into first allocation**

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.

A kernel is executed by many threads; how do we determine the mapping between threads and the data they operate on?

–Tom Jerry

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.

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.

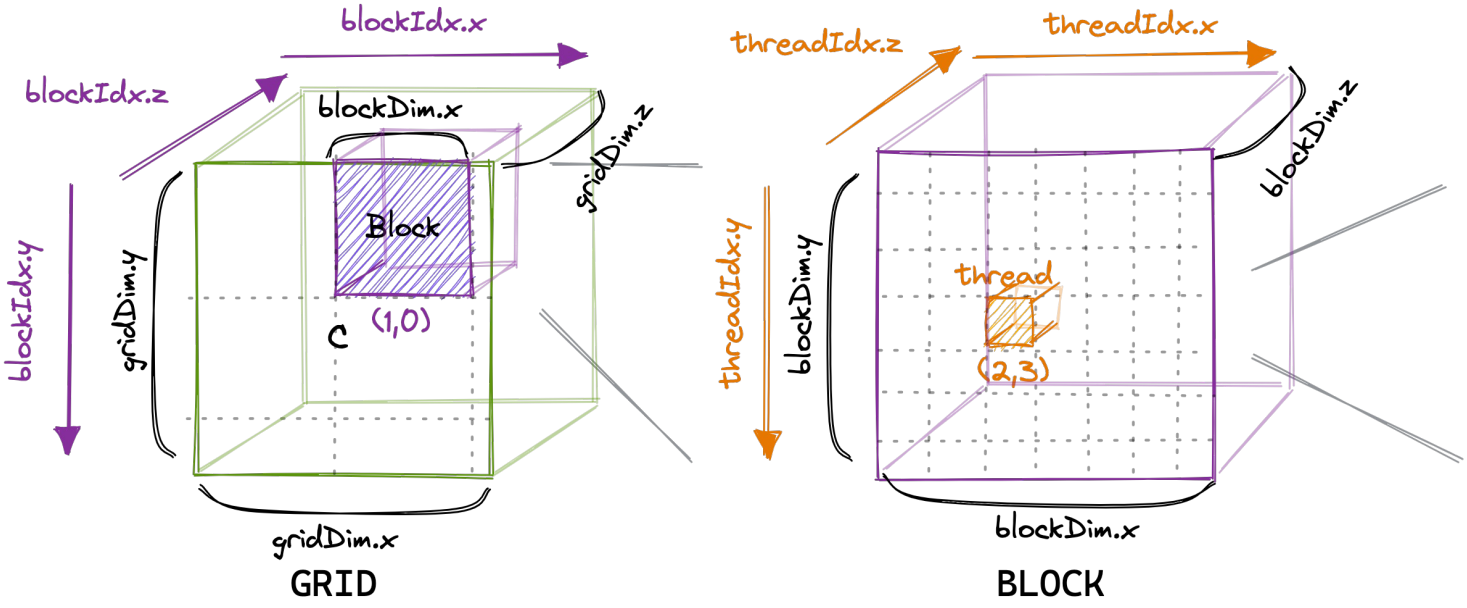
How do 2D / 3D threads get divided into warps?

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.

Thread Organization



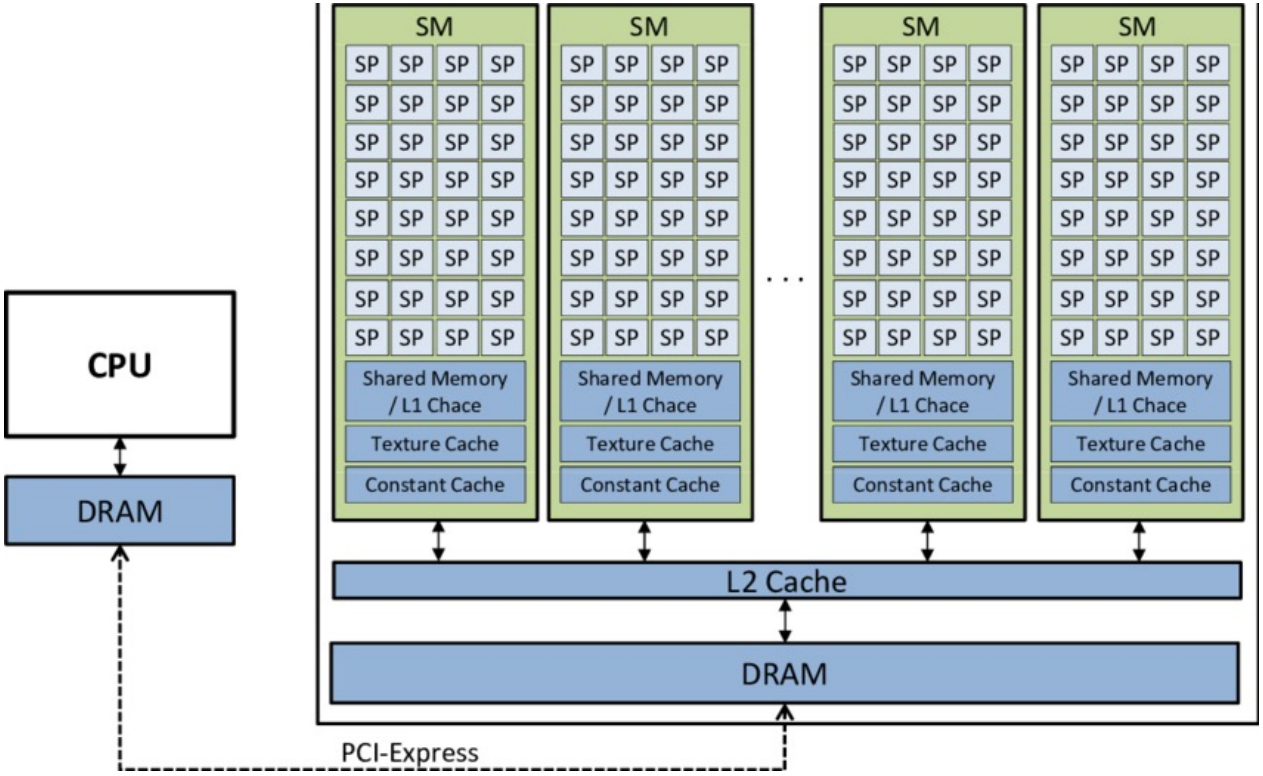
a single thread of computation, minding its own business

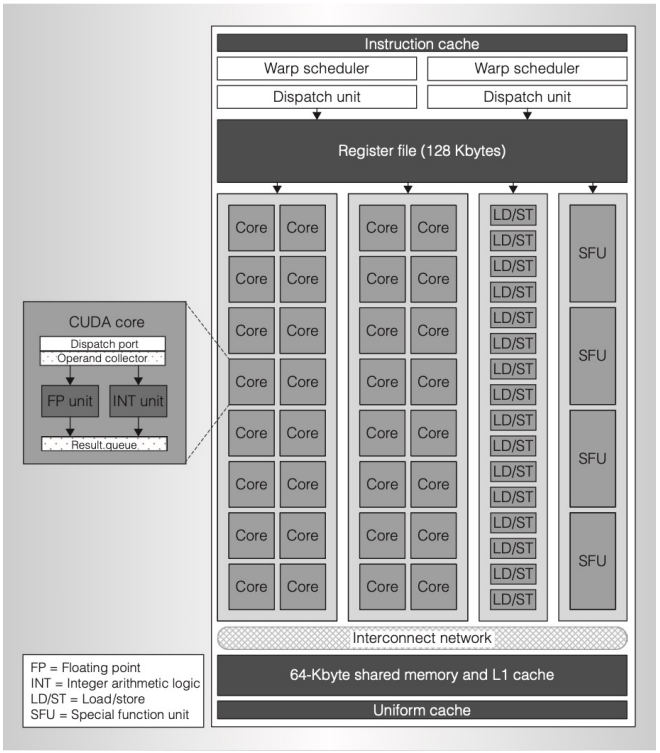
THREAD

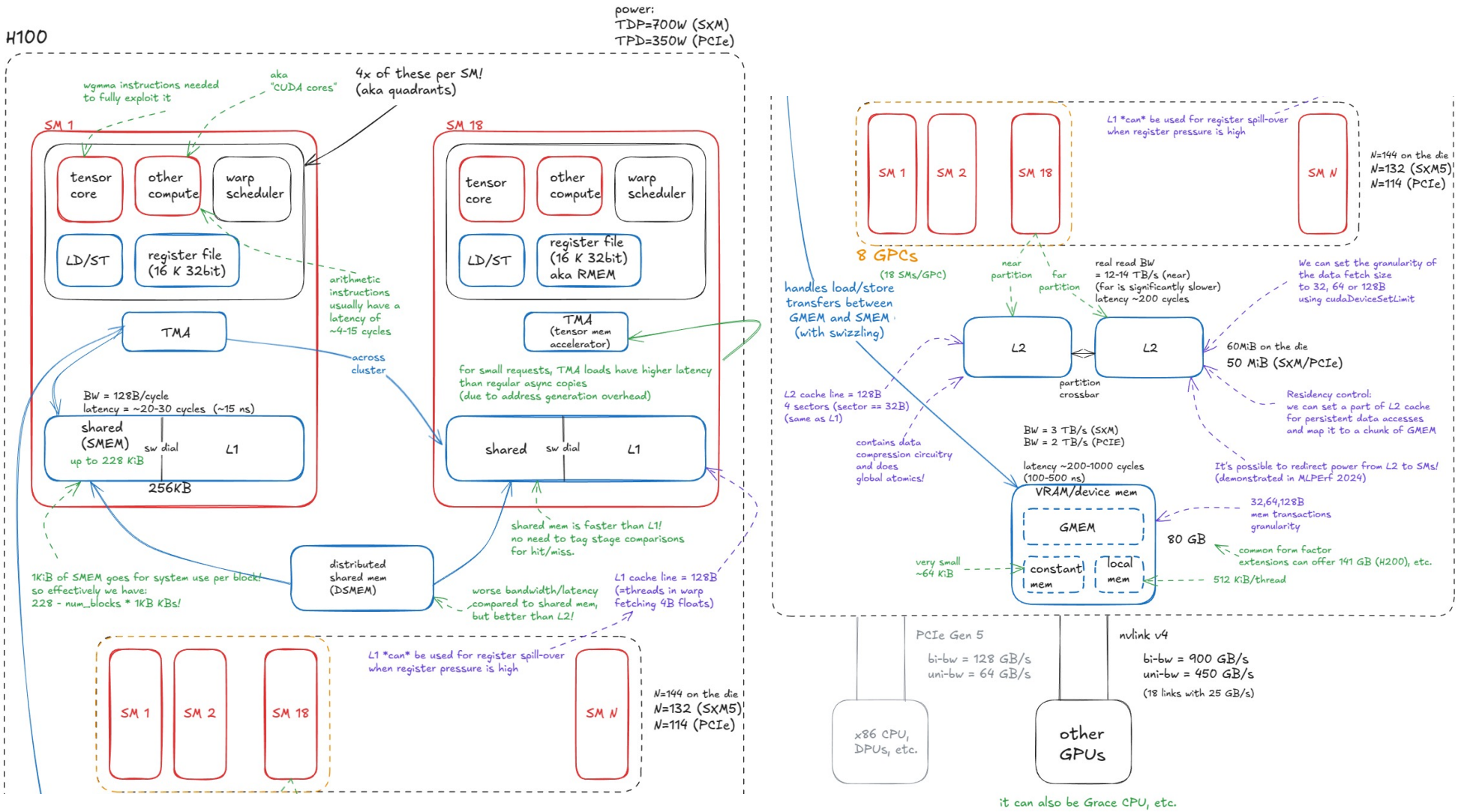
Why do we need multi-level computation granularity?

–Tom Jerry

Computation vs. Memory Hierarchy

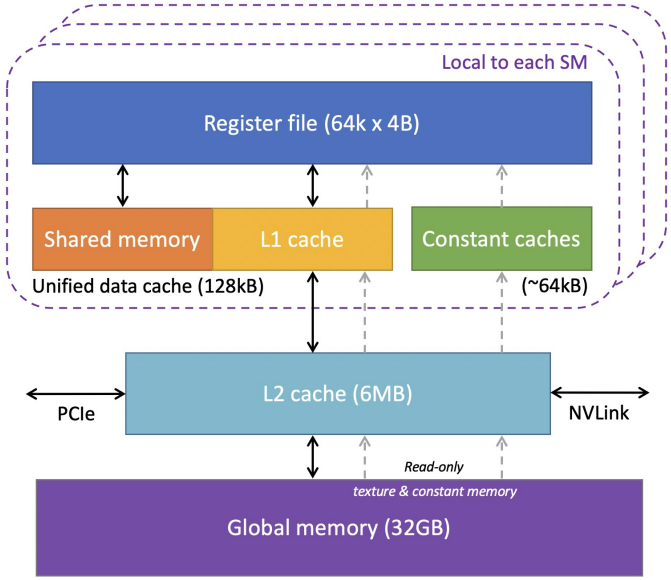




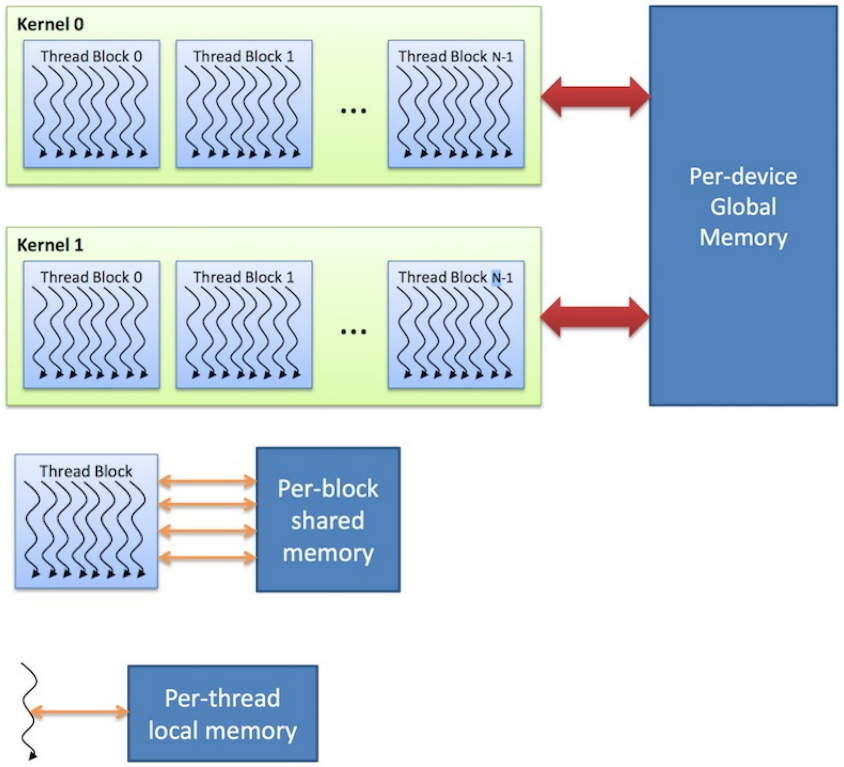

H100
power:
TDP=700W (SXM)
TPD=350W (PCIe)
wgmma instructions needed to fully exploit it
aka "CUDA cores"
4x of these per SM!
(aka quadrants)
SM 1
tensor core
other compute
warp scheduler
LD/ST
register file (16 K 32bit)
arithmetic instructions usually have a latency of ~4-15 cycles
TMA
across cluster
BW = 128B/cycle
latency = ~20-30 cycles (~15 ns)
shared (SMEM)
sw dial
L1
up to 228 KiB
256KB
1KiB of SMEM goes for system use per block!
so effectively we have:
228 - num_blocks * 1KB KBs!
distributed shared mem (DSMEM)
worse bandwidth/latency compared to shared mem, but better than L2!
SM 18
register file (16 K 32bit) aka RMEM
TMA (tensor mem accelerator)
for small requests, TMA loads have higher latency than regular async copies
(due to address generation overhead)
shared mem is faster than L1!
no need to tag stage comparisons for hit/miss.
L1 cache line = 128B
(=threads in warp fetching 4B floats)
L1 *can* be used for register spill-over when register pressure is high
SM 2
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
8 GPCs
(18 SMs/GPC)
handles load/store transfers between GMEM and SMEM (with swizzling)
near partition
far partition
real read BW
= 12-14 TB/s (near)
(far is significantly slower)
latency ~200 cycles
L2
partition crossbar
60MiB on the die
50 MiB (SXM/PCIe)
We can set the granularity of the data fetch size to 32, 64 or 128B using cudaDeviceSetLimit
Residency control:
we can set a part of L2 cache for persistent data accesses and map it to a chunk of GMEM
L2 cache line = 128B
4 sectors (sector == 32B)
(same as L1)
contains data compression circuitry and does global atomics!
BW = 3 TB/s (SXM)
BW = 2 TB/s (PCIE)
latency ~200-1000 cycles
(100-500 ns)
VRAM/device mem
GMEM
80 GB
32,64,128B mem transactions granularity
common form factor extensions can offer 141 GB (H200), etc.
It's possible to redirect power from L2 to SMs!
(demonstrated in MLPerf 2024)
very small ~64 KiB
constant mem
local mem
512 KiB/thread
PCIe Gen 5
bi-bw = 128 GB/s
uni-bw = 64 GB/s
nvlink v4
bi-bw = 900 GB/s
uni-bw = 450 GB/s
(18 links with 25 GB/s)
x86 CPU, DPUs, etc.
other GPUs
it can also be Grace CPU, etc.

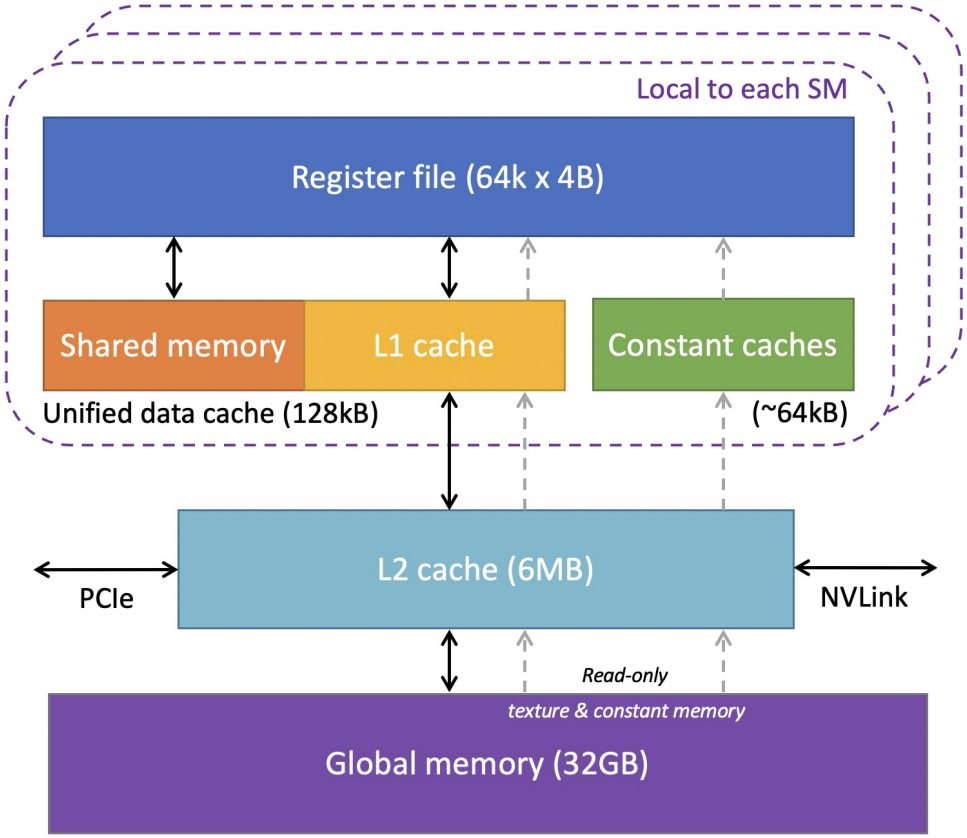
Memory Hierarchy

Level	Location	Latency	Size	Scope
Registers	SM	~1 cycle	very small	per thread
Shared Memory	SM	~20 cycles	~100 KB	per block
L2 Cache	GPU	~200 cycles	tens of MB	whole GPU
Global Memory (HBM)	GPU DRAM	~500–800 cycles	tens of GB	all threads
Host Memory	CPU	very high	large	system



Memory Hierarchy





Kernel

CUDA Concept	Computation Hardware	Memory Hierarchy
Grid	Entire GPU	Global memory (HBM / DRAM)
Block	Streaming Multiprocessor (SM)	Shared memory / L1 cache
Thread	CUDA core (warp lane)	Registers

Key intuition

- **Threads** perform computation using **registers**.
- **Threads in a block** cooperate on the same **SM** and communicate through **shared memory**.
- **Blocks in a grid** execute across the entire **GPU** and communicate via **global memory**.

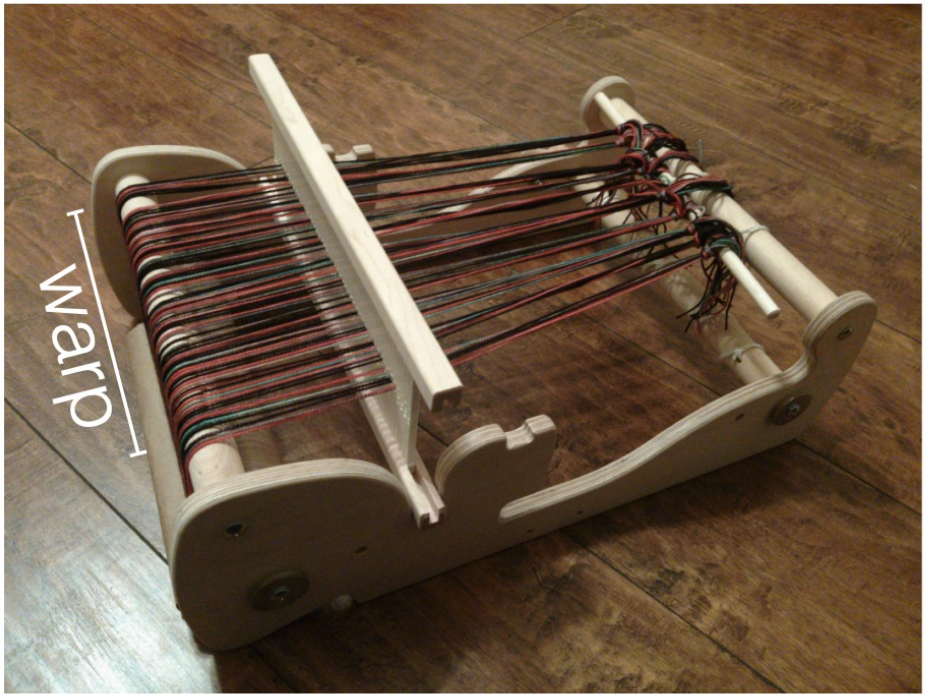
SIMD, SIMT, Warp

–Tom Jerry

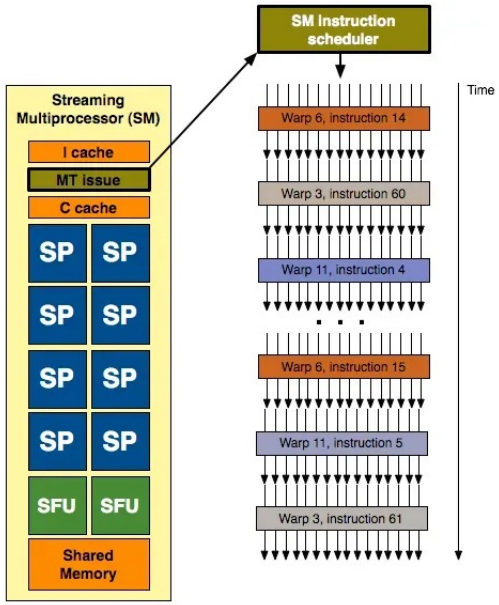
Warp

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.



Warp

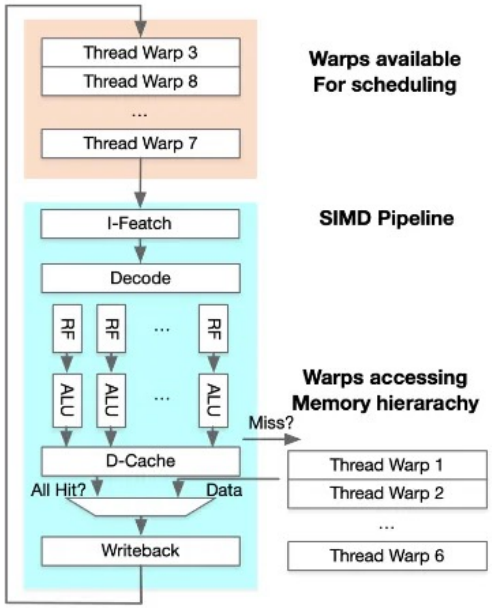


Imagine a block with 128 threads, split into 4 warps (32 threads each):

- 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.

SIMD, SIMT, Warp

Concept	What it is	Key idea
SIMD	Hardware execution style	One instruction operates on multiple data elements
SIMT	Programming model	Many threads run the same code independently
Warp	Hardware execution unit (GPU)	32 threads executed together in lockstep

---

👉 **Key takeaway:** Warp is how GPUs implement SIMT on SIMD-like hardware

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.

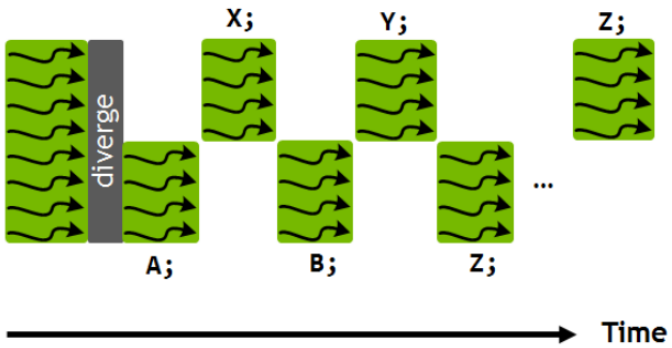
- **Memory Sharing:** Threads within a warp can efficiently share data using shared memory, which is a fast, on-chip memory accessible to all threads in the same block.
- **Warp Divergence:** When threads within the same warp follow different execution paths due to conditional statements (e.g., if-else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

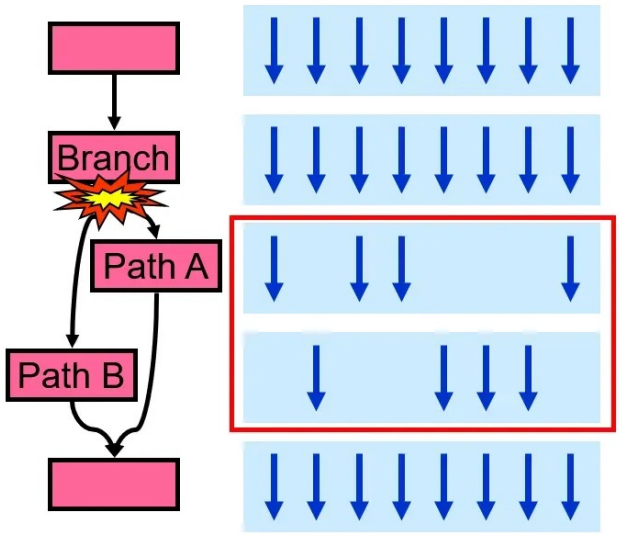
This has profound implications: memory accesses are coalesced at the warp level, divergence penalties are incurred at the warp level, and scheduling decisions operate on warps rather than individual threads. Understanding this distinction is essential, because optimizing GPU performance requires aligning program behavior with warp-level execution, not just thread-level logic.

Divergence

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

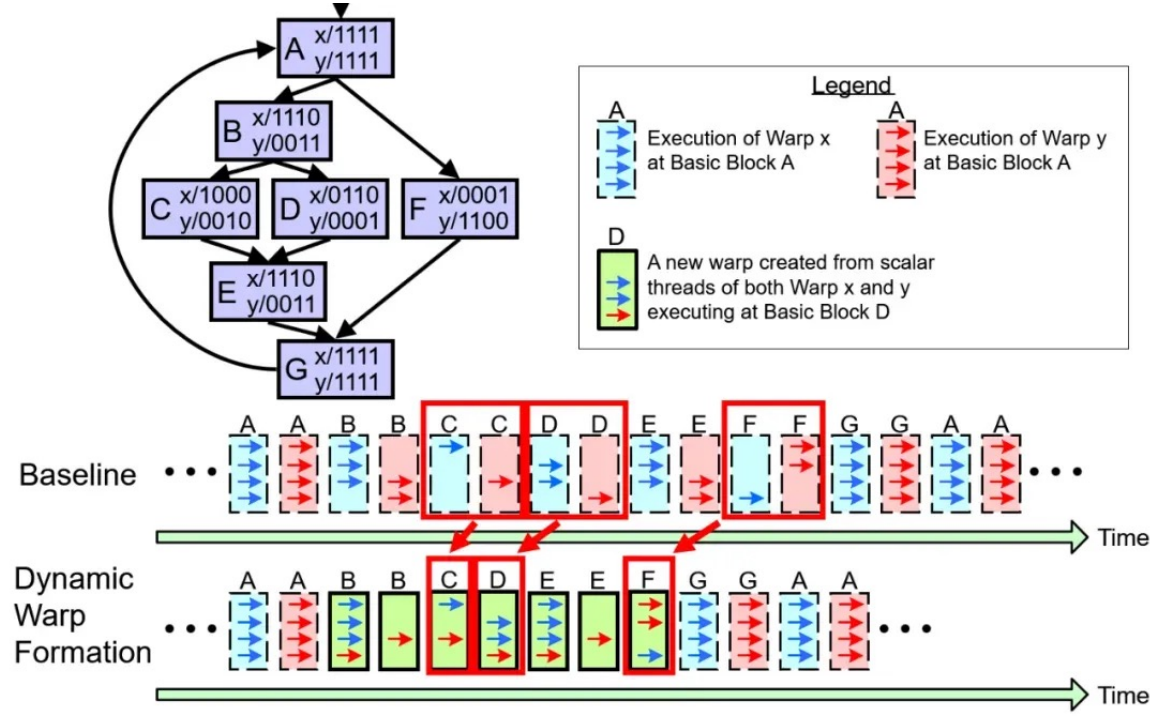
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```





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.



NVCC

The NVIDIA CUDA Compiler Driver

nvcc is the NVIDIA CUDA compiler driver, part of the CUDA toolkit. It compiles CUDA C/C++ programs containing both host (CPU) and device (GPU) code. Its job is to separate, compile, and link host and device code into one executable.

- Code Separation – Splits host and device code in each .cu file.
- Dual Compilation Paths
- Host code → compiled by gcc or cl.
- Device code → compiled by ptxas into PTX or cubin binaries.
- Linking – nvlink merges GPU objects with CPU objects into a final executable.

A host compiler (gcc / cl) for CPU code,
A device compiler (ptxas) for GPU code,
A linker (nvlink) to combine everything into one binary.

CUDA C/C++ → PTX (virtual ISA) → SASS (machine ISA) → GPU execution

PTX (Compiler planned)

PTX (Parallel Thread Execution) is NVIDIA's virtual GPU instruction set architecture (ISA) — an intermediate layer between CUDA C/C++ source code and the hardware-level SASS machine code.

It provides a human-readable, typed, and portable representation of GPU programs, defining how threads, memory spaces, and arithmetic operations behave in a device-independent way.

PTX expresses what the compiler intends the hardware to do—including data movement, parallel execution, and synchronization—while abstracting away hardware-specific details like pipelines, cache policies, and timing.

SASS (GPU reality)

SASS (Streaming Assembler) is NVIDIA's hardware-level GPU instruction set architecture, representing the actual machine code executed by the Streaming Multiprocessors (SMs).

Each SASS instruction is a 128-bit binary operation that encodes the precise arithmetic, memory, and control actions the GPU performs, including fused operations, cache policies, and register usage.

Unlike PTX, which is virtual and portable, SASS is architecture-specific (e.g., sm_80, sm_90) and reveals how the hardware executes a kernel—making it essential for low-level performance tuning, scheduling analysis, and understanding real GPU behavior.

PTX and SASS

Analogy

- **PTX ≈ Java bytecode** — portable, virtual.
- **SASS ≈ x86 assembly** — concrete, hardware-specific.

Key insight:
PTX expresses *intent*, SASS exposes *reality*. Profiling and performance analysis must look at SASS to see the true instruction mix, cache policy, and resource usage.

```
CUDA C/C++  →  PTX (virtual ISA)  →  SASS (machine ISA)  →  GPU execution
```

PTX ISA at a glance

Category	Examples	Purpose
Data Move	`ld`, `st`, `mov`, `cvt`	Move / convert data across registers & memory.
Arithmetic	`add`, `sub`, `mul`, `mad`, `fma`	Integer & floating-point math operations.
Logic / Bit	`and`, `or`, `xor`, `not`, `shl`, `shr`	Bitwise & shift operations.
Control Flow	`bra`, `@p bra`, `call`, `ret`, `exit`	Branching & predicated execution.
Comparison	`setp.eq`, `setp.lt`, `setp.ge`	Set predicate registers for conditions.
Memory Spaces	`.global`, `.shared`, `.local`, `.const`	Specify where data is loaded/stored.
Sync & Parallel	`bar.sync`, `membar`, `shfl.sync`	Coordinate threads & memory ordering.
Tensor / Async	`mma.sync`, `cp.async`	Tensor-core and async copy operations.

SASS ISA at a glance

Category	Examples	Purpose
Arithmetic	`FADD`, `FFMA`, `IMAD`, `IADD3`	Real GPU math ops — fused, hardware-optimized.
Logic / Bit	`LOP3`, `SHF`, `BFE`, `BMSK`	Bitwise logic and shift operations.
Memory Access	`LDG.E`, `STG.E`, `LDS`, `STS`, `LDC`	Load/store from global, shared, or constant memory (with cache policies).
Control Flow	`BRA`, `@P0 EXIT`, `SSY`, `SYNC`	Branching, predication, warp reconvergence.
Predicates	`ISETP`, `FSETP`, `@P`	Conditional execution on hardware predicate flags.
Special Regs	`S2R`, `S2UR`, `ULDC`	Move from thread/block registers or uniform regs.
Tensor / Async	`HMMA`, `LDGSTS`, `CPAsync`	Tensor-core matrix ops & async copy pipelines.
Sync / Barrier	`BAR.SYNC`, `MEMBAR`	Thread/block synchronization and memory fences.

Registers

Type	Purpose
General (R0–R254)	Standard per-thread registers used for integers, floats, and addresses (32-bit each; paired for 64-bit). They form the main computational workspace for every thread.
Predicate (P0–P7)	Boolean flags per thread, used for **predicated execution** (e.g., conditional instructions via `@P`, set by `ISETP`/`FSETP`).
Uniform (UR0–UR15)	Shared across a warp; store **warp-uniform values** (e.g., kernel parameters, descriptors, block indices). Save register space and broadcast same data to all threads.
Special (SR_*)	**Read-only hardware registers** containing built-in thread/block state (`SR_TID`, `SR_CTAID`, `SR_NTID`, `SR_LANEID`, `clock`).
Tensor (implicit)	Internal per-thread-group fragments used by **Tensor Core MMA instructions** (`HMMA`, `WMMA`), not directly programmer-visible.

```cpp
void vectorAdd(const float* A, const float* B, float* C, int N) {
    for (int idx = 0; idx < N; idx++) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1 << 16; // 65536 elements
    size_t size = N * sizeof(float);

    // Allocate host memory using vectors (auto-managed)
    std::vector<float> A(N), B(N), C(N);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        A[i] = 1.0f;
        B[i] = 2.0f;
    }

    // Perform vector addition on CPU
    vectorAdd(A.data(), B.data(), C.data(), N);

    // Verify results (print first 5)
    for (int i = 0; i < 5; i++) {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }

    return 0;
}
```

```
)
{
        .reg .pred      %p<2>;
        .reg .b32       %r<6>;
        .reg .b32       %f<4>;
        .reg .b64       %rd<11>;

        ld.param.u64    %rd1, [_Z15vectorAddKernelPKfS0_Pfi_param_0];
        ld.param.u64    %rd2, [_Z15vectorAddKernelPKfS0_Pfi_param_1];
        ld.param.u64    %rd3, [_Z15vectorAddKernelPKfS0_Pfi_param_2];
        ld.param.u32    %r2, [_Z15vectorAddKernelPKfS0_Pfi_param_3];
        mov.u32         %r3, %ctaid.x;
        mov.u32         %r4, %ntid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        $L__BB0_2;
        cvta.to.global.u64      %rd4, %rd1;
        cvta.to.global.u64      %rd5, %rd2;
        cvta.to.global.u64      %rd6, %rd3;
        mul.wide.s32    %rd7, %r1, 4;
        add.s64         %rd8, %rd4, %rd7;
        ld.global.f32   %f1, [%rd8];
        add.s64         %rd9, %rd5, %rd7;
        ld.global.f32   %f2, [%rd9];
        add.f32         %f3, %f1, %f2;
        add.s64         %rd10, %rd6, %rd7;
        st.global.f32   [%rd10], %f3;
$L__BB0_2:
        ret;
```

```cpp
void vectorAdd(const float* A, const float* B, float* C, int N) {
    for (int idx = 0; idx < N; idx++) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1 << 16; // 65536 elements
    size_t size = N * sizeof(float);

    // Allocate host memory using vectors (auto-managed)
    std::vector<float> A(N), B(N), C(N);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        A[i] = 1.0f;
        B[i] = 2.0f;
    }

    // Perform vector addition on CPU
    vectorAdd(A.data(), B.data(), C.data(), N);

    // Verify results (print first 5)
    for (int i = 0; i < 5; i++) {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }

    return 0;
}
```

```
//---------------------- .text._Z15vectorAddKernelPKfS0_Pfi ------------
        .section        .text._Z15vectorAddKernelPKfS0_Pfi,"ax",@progbi
        .align  128
        .global         _Z15vectorAddKernelPKfS0_Pfi
        .type           _Z15vectorAddKernelPKfS0_Pfi,@function
        .size           _Z15vectorAddKernelPKfS0_Pfi,(.L_x_1 - _Z15vecti
        .other          _Z15vectorAddKernelPKfS0_Pfi,@"STO_CUDA_ENTRY S
_Z15vectorAddKernelPKfS0_Pfi:
.text._Z15vectorAddKernelPKfS0_Pfi:
        /*0000*/                   LDC R1, c[0x0][0x28] ;
        /*0010*/                   S2R R0, SR_TID.X ;
        /*0020*/                   S2UR UR4, SR_CTAID.X ;
        /*0030*/                   LDC R9, c[0x0][RZ] ;
        /*0040*/                   IMAD R9, R9, UR4, R0 ;
        /*0050*/                   ULDC UR4, c[0x0][0x228] ;
        /*0060*/                   ISETP.GE.AND P0, PT, R9, UR4, PT ;
        /*0070*/                @P0 EXIT ;
        /*0080*/                   LDC.64 R2, c[0x0][0x210] ;
        /*0090*/                   ULDC.64 UR4, c[0x0][0x208] ;
        /*00a0*/                   LDC.64 R4, c[0x0][0x218] ;
        /*00b0*/                   LDC.64 R6, c[0x0][0x220] ;
        /*00c0*/                   IMAD.WIDE R2, R9, 0x4, R2 ;
        /*00d0*/                   LDG.E R3, desc[UR4][R2.64] ;
        /*00e0*/                   IMAD.WIDE R4, R9, 0x4, R4 ;
        /*00f0*/                   LDG.E R4, desc[UR4][R4.64] ;
        /*0100*/                   IMAD.WIDE R6, R9, 0x4, R6 ;
        /*0110*/                   FADD R9, R4, R3 ;
        /*0120*/                   STG.E desc[UR4][R6.64], R9 ;
        /*0130*/                   EXIT ;
.L_x_0:
        /*0140*/                   BRA `(.L_x_0);
```

What is the difference between CPU thread and GPU thread?

What is the difference between CPU thread and GPU thread?

CPU threads are relative heavy and managed by the operating system.

A GPU thread is a lightweight execution context that represents one instance of a data-parallel computation. GPU threads are extremely lightweight and scheduled in groups (warps) by hardware. Each thread executes the same program but operates on different data, and thousands of such threads are active concurrently within a GPU.

Key idea: CPU threads carry full software-managed state; GPU threads carry minimal hardware-managed state.

What do we mean heavy or lightweight?

What state must be maintained per thread for correct execution?

Context State	CPU Thread
Registers	Full register set saved/restored on context switch
Program Counter	Saved/restored by OS
Stack	Dedicated stack per thread
Metadata	OS thread control block (priority, state, etc.)
Context Switch	Heavy (save/restore to memory)

What state must be maintained per thread for correct execution?

Type	Name in Code	Scope	Stored Where	Purpose
General Registers (GPR)	`R0`–`Rn`	Per thread	Register file (on-chip)	Hold variables, intermediates, pointers
Predicate Registers	`P0`–`Pn`	Per thread	Register file / control logic	Store boolean conditions for branching/ masking
Uniform Registers	`URx`	Per warp	Separate uniform register file	Hold values shared across all threads in a warp
Special Registers	`SR_*` (e.g., `SR_TID`)	Per thread (read-only)	Hardware-generated	Provide thread/block IDs, lane ID, etc.
Constant Memory Registers	`c[...]`	Shared (read-only)	Constant cache / memory	Kernel parameters, constants
Program Counter (PC)	(implicit)	Per warp	Warp scheduler / control unit	Tracks current instruction location
Active Mask / SIMT Stack	(implicit)	Per warp	Warp control hardware	Tracks divergence (which threads are active)

What state must be maintained per thread for correct execution?

Each GPU thread must maintain its own execution state, including register values, program counters, and intermediate variables. This state ensures that each thread can proceed independently through the program, even though threads are executed in lockstep within a warp.

On GPU, the per-thread state is stored in SM registers, which provide the lowest-latency storage and are directly accessed by the SM, enabling fast and deterministic execution. Because GPUs rely on rapidly switching between threads to hide latency, keeping thread state in registers allows the hardware to resume execution of any thread with minimal delay.

What state must be maintained per thread for correct execution?

Context State	Where it is stored	Explanation
Registers (variables)	**Register file (per thread)**	This is the main per-thread state (all local variables, intermediates)
Program Counter (PC)	**Warp scheduler / control unit**	One PC per warp, not per thread
Thread ID (threadIdx, blockIdx)	**Special registers / hardware constants**	Exposed as registers in code, but actually generated by hardware
Predicate / active mask	**Warp control state (SIMT stack)**	Tracks which threads are active during divergence
Warp ID / lane ID	**Special registers**	Provided by hardware, not normal register allocation
Local memory (spill)	**Global memory (HBM)**	Used only when registers overflow
Shared memory state	**Shared memory (on-chip SRAM)**	Not per-thread, shared across block

How many registers are used by each thread?

PTX ISA at a glance

Category	Examples	Purpose
Data Move	`ld`, `st`, `mov`, `cvt`	Move / convert data across registers & memory.
Arithmetic	`add`, `sub`, `mul`, `mad`, `fma`	Integer & floating-point math operations.
Logic / Bit	`and`, `or`, `xor`, `not`, `shl`, `shr`	Bitwise & shift operations.
Control Flow	`bra`, `@p bra`, `call`, `ret`, `exit`	Branching & predicated execution.
Comparison	`setp.eq`, `setp.lt`, `setp.ge`	Set predicate registers for conditions.
Memory Spaces	`.global`, `.shared`, `.local`, `.const`	Specify where data is loaded/stored.
Sync & Parallel	`bar.sync`, `membar`, `shfl.sync`	Coordinate threads & memory ordering.
Tensor / Async	`mma.sync`, `cp.async`	Tensor-core and async copy operations.

```cpp
void vectorAdd(const float* A, const float* B, float* C, int N) {
    for (int idx = 0; idx < N; idx++) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1 << 16; // 65536 elements
    size_t size = N * sizeof(float);

    // Allocate host memory using vectors (auto-managed)
    std::vector<float> A(N), B(N), C(N);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        A[i] = 1.0f;
        B[i] = 2.0f;
    }

    // Perform vector addition on CPU
    vectorAdd(A.data(), B.data(), C.data(), N);

    // Verify results (print first 5)
    for (int i = 0; i < 5; i++) {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }

    return 0;
}
```

```
)
{
        .reg .pred      %p<2>;
        .reg .b32       %r<6>;
        .reg .b32       %f<4>;
        .reg .b64       %rd<11>;

        ld.param.u64    %rd1, [_Z15vectorAddKernelPKfS0_Pfi_param_0];
        ld.param.u64    %rd2, [_Z15vectorAddKernelPKfS0_Pfi_param_1];
        ld.param.u64    %rd3, [_Z15vectorAddKernelPKfS0_Pfi_param_2];
        ld.param.u32    %r2, [_Z15vectorAddKernelPKfS0_Pfi_param_3];
        mov.u32         %r3, %ctaid.x;
        mov.u32         %r4, %ntid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        $L__BB0_2;
        cvta.to.global.u64      %rd4, %rd1;
        cvta.to.global.u64      %rd5, %rd2;
        cvta.to.global.u64      %rd6, %rd3;
        mul.wide.s32    %rd7, %r1, 4;
        add.s64         %rd8, %rd4, %rd7;
        ld.global.f32   %f1, [%rd8];
        add.s64         %rd9, %rd5, %rd7;
        ld.global.f32   %f2, [%rd9];
        add.f32         %f3, %f1, %f2;
        add.s64         %rd10, %rd6, %rd7;
        st.global.f32   [%rd10], %f3;
$L__BB0_2:
        ret;
```

```cpp
void vectorAdd(const float* A, const float* B, float* C, int N) {
    for (int idx = 0; idx < N; idx++) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1 << 16; // 65536 elements
    size_t size = N * sizeof(float);

    // Allocate host memory using vectors (auto-managed)
    std::vector<float> A(N), B(N), C(N);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        A[i] = 1.0f;
        B[i] = 2.0f;
    }

    // Perform vector addition on CPU
    vectorAdd(A.data(), B.data(), C.data(), N);

    // Verify results (print first 5)
    for (int i = 0; i < 5; i++) {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }

    return 0;
}
```

```
//---------------------- .text._Z15vectorAddKernelPKfS0_Pfi ------------
        .section        .text._Z15vectorAddKernelPKfS0_Pfi,"ax",@progbi
        .align  128
        .global         _Z15vectorAddKernelPKfS0_Pfi
        .type           _Z15vectorAddKernelPKfS0_Pfi,@function
        .size           _Z15vectorAddKernelPKfS0_Pfi,(.L_x_1 - _Z15vecti
        .other          _Z15vectorAddKernelPKfS0_Pfi,@"STO_CUDA_ENTRY S
_Z15vectorAddKernelPKfS0_Pfi:
.text._Z15vectorAddKernelPKfS0_Pfi:
        /*0000*/                   LDC R1, c[0x0][0x28] ;
        /*0010*/                   S2R R0, SR_TID.X ;
        /*0020*/                   S2UR UR4, SR_CTAID.X ;
        /*0030*/                   LDC R9, c[0x0][RZ] ;
        /*0040*/                   IMAD R9, R9, UR4, R0 ;
        /*0050*/                   ULDC UR4, c[0x0][0x228] ;
        /*0060*/                   ISETP.GE.AND P0, PT, R9, UR4, PT ;
        /*0070*/                @P0 EXIT ;
        /*0080*/                   LDC.64 R2, c[0x0][0x210] ;
        /*0090*/                   ULDC.64 UR4, c[0x0][0x208] ;
        /*00a0*/                   LDC.64 R4, c[0x0][0x218] ;
        /*00b0*/                   LDC.64 R6, c[0x0][0x220] ;
        /*00c0*/                   IMAD.WIDE R2, R9, 0x4, R2 ;
        /*00d0*/                   LDG.E R3, desc[UR4][R2.64] ;
        /*00e0*/                   IMAD.WIDE R4, R9, 0x4, R4 ;
        /*00f0*/                   LDG.E R4, desc[UR4][R4.64] ;
        /*0100*/                   IMAD.WIDE R6, R9, 0x4, R6 ;
        /*0110*/                   FADD R9, R4, R3 ;
        /*0120*/                   STG.E desc[UR4][R6.64], R9 ;
        /*0130*/                   EXIT ;
.L_x_0:
        /*0140*/                   BRA `(.L_x_0);
```

How many registers are used by each thread?

The number of registers per thread is determined by the complexity of the kernel, including the number of variables, loop unrolling, and instruction-level parallelism. The compiler analyzes the program and assigns registers to hold intermediate values, attempting to balance performance and resource usage.

```
memx@spark-b6d9:~/lishang/cuda/exp1$ nvcc -Xptxas -v -arch=sm_90 vector_add.cu -o vector_add
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z15vectorAddKernelPKfS0_Pfi' for 'sm_90'
ptxas info    : Function properties for _Z15vectorAddKernelPKfS0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, used 0 barriers
ptxas info    : Compile time = 1.581 ms
```

Can threads share registers, in other words, be private or shared?

Can threads share registers, in other words, be private or shared?

Registers must be private to each thread to ensure correctness, as each thread operates on different data and must not interfere with others. They must also be deterministic because the compiler assigns specific variables to specific registers at compile time, and the hardware expects those assignments to remain fixed during execution. This guarantees predictable behavior and enables efficient instruction scheduling.

Which one has more registers, CPU or GPU?

Which one has more registers, CPU or GPU?

GPUs require many more registers because they maintain the state of thousands of concurrent threads, whereas CPUs only manage a small number of threads per core. Since each GPU thread needs its own register allocation, the total register file must scale with the number of active threads. This large register capacity is what enables GPUs to sustain massive parallelism and hide memory latency.

When is register allocation be decided, compile time or runtime?

When is register allocation be decided, compile time or runtime?

Register allocation is decided at compile time because instructions explicitly reference physical registers, and the execution model requires all threads in a warp to follow the same instruction stream. Dynamic allocation at runtime would require indirection and reconfiguration of instructions, which would greatly increase hardware complexity and reduce performance predictability.

Feature	CPU	GPU
Register allocation	compile + runtime	compile only
Register renaming	yes	no
Execution model	out-of-order	mostly in-order
Parallelism	low	massive
Latency handling	instruction-level	thread-level

What happens if registers are insufficient?

What happens if registers are insufficient?

Register spilling occurs when there are more live variables than available registers. If registers are insufficient, the compiler spills some variables to local memory, which resides in global memory. Accesses to these spilled variables are implemented as memory loads and stores, fundamentally slower than register access.

But, wait, we have on-chip cache, right?

But, wait, we have on-chip cache, right?

Cache can mitigate spill cost when the spilled variables exhibit temporal locality, meaning they are accessed repeatedly within a short time window. If the working set of spilled data fits within L1 or L2 and is not evicted, subsequent accesses can be served from cache, reducing latency significantly compared to DRAM.

Therefore, caches are still needed because not all access patterns can be predicted or explicitly managed. L1 and L2 caches provide opportunistic reuse for data that exhibits locality but is not explicitly staged in shared memory.

What if threads need to share data?

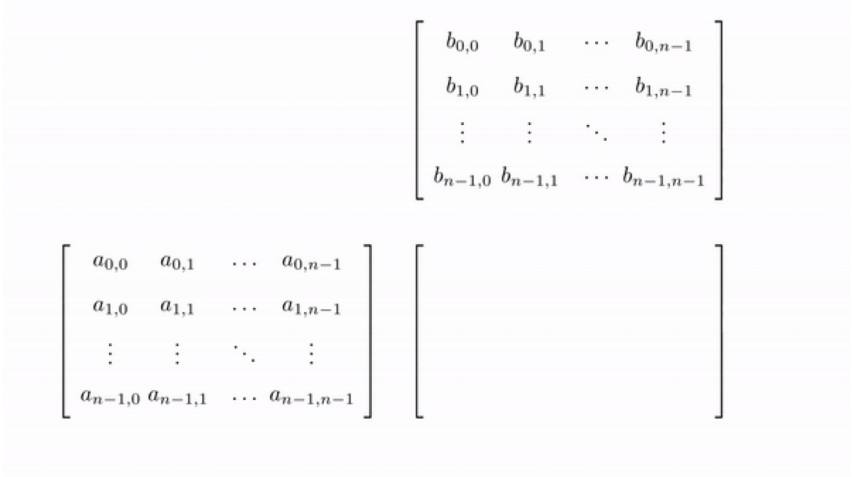
What if threads need to share data?

Shared memory exists to enable data reuse across threads within a block. While registers are private to each thread, shared memory allows multiple threads to cooperatively load and reuse data, reducing redundant global memory accesses and improving efficiency.

What kind of data reuse pattern requires shared memory?

What kind of data reuse pattern requires shared memory?

Shared memory is most useful when multiple threads access overlapping data regions, such as in matrix multiplication or attention mechanisms. In these cases, data loaded once into shared memory can be reused by many threads, amortizing the cost of global memory access.



Is shared memory managed by software or hardware?

Is shared memory managed by software or hardware?

Shared memory is explicitly managed to give programmers precise control over data movement and reuse. This allows optimized kernels to exploit known access patterns and avoid the unpredictability of hardware-managed caches.

What is the key difference between cache and shared memory?

What is the key difference between cache and shared memory?

The key difference is control: shared memory is explicitly managed by software, while caches are automatically managed by hardware. Shared memory provides guaranteed reuse when used correctly, whereas cache provides best-effort reuse based on access patterns.

What is the key difference between cache and shared memory?

Take-home message: GPUs combine explicit software-managed memory with implicit hardware-managed caching.

Shared memory is explicitly controlled by the programmer, allowing precise management of data reuse when access patterns are predictable. In contrast, caches are managed automatically by hardware, providing opportunistic reuse for irregular access patterns.

This hybrid design reflects a key architectural philosophy: give programmers control where they can exploit structure, and rely on hardware where patterns are uncertain. Effective GPU programming requires understanding when to use each mechanism.

Which one is larger, GPU shared memory or CPU cache?

Which one is larger, GPU shared memory or CPU cache?

Shared memory is smaller because it is designed for controlled, high-efficiency reuse rather than general-purpose caching. Its size is a tradeoff between capacity and speed, and it is optimized for predictable access patterns rather than arbitrary workloads.

Which one is larger, GPU shared memory or CPU cache?

Metric	Modern CPU (Intel)	Modern GPU (H100 / Blackwell)	Key Insight
Per-core memory (closest fast local storage)	**~1–2 MB L2 per core**	**~256 KB (L1 + shared memory per SM)**	CPU has much larger **private fast memory per core**
Total shared cache	**~30–100 MB (L3)**	**~50–100+ MB (L2)**	Comparable total capacity
Cache ownership	Private (L2) + shared (L3)	Fully shared (L2) + configurable L1/shared	GPU has weaker locality guarantees
Effective cache per compute unit	**~1–2 MB**	**~0.2–0.5 MB (shared L2 fraction)**	GPU per-unit cache is much smaller
Control model	Hardware-managed cache hierarchy	Software-managed (shared) + hardware cache	GPU exposes more control but less guarantee
Design goal	Low latency, strong locality	High bandwidth, throughput	Different optimization philosophy

How large should the working set be to benefit from L2 cache?

How large should the working set be to benefit from L2 cache?

A working set benefits from L2 cache not simply by being smaller than the physical L2 size, but by being small enough and reused quickly enough to avoid eviction under concurrent access. In practice, although modern GPUs have tens of megabytes of L2, the effective usable portion per kernel is often only a few megabytes due to sharing across many SMs and warps. Therefore, L2 is most effective for small, frequently reused data with short reuse distance, while large or streaming working sets quickly lose locality and fall back to global memory.

DeepSeek’s lession

Aspect	Summary
Goal	Push GPU efficiency beyond CUDA’s default compiler by writing or modifying **PTX (Parallel Thread Execution)** code directly.
Why	CUDA and `ptxas` hide low-level control (cache behavior, scheduling, memory hints). DeepSeek needed **fine-grained tuning** for bandwidth-critical LLM kernels on H100/H800 GPUs.
Approach	Selectively replaced compiler-generated PTX in **hot kernels** with custom instructions controlling **cache policy** and **memory streaming behavior**.
Example	Used a custom PTX load instruction:`ptx ld.global.nc.L1::no_allocate.L2::256B %r, [%rd]; `→ Non-coherent global load, **bypasses L1 cache**, **fetches 256B via L2**.Reduces L1 pollution and improves large-tensor streaming performance.
Result	Higher effective memory bandwidth and improved throughput (reported ~5–10% gain in key kernels).
Takeaway	PTX gives access to **hardware-level cache & memory control** normally hidden by CUDA — useful for expert optimization, but fragile and hardware-specific.

How do registers, shared memory, L1, L2, and global memory interact?

How do registers, shared memory, L1, L2, and global memory interact?

These components form a hierarchy:

- Registers store immediate, per-thread data
- Shared memory stores per-block cross-thread cooperative data
- Caches provide opportunistic, irregular or unpredictable reuse
- Global memory serves as the backing store for all data regardless of reuse

Data flows between these levels depending on access patterns and locality.

How do registers, shared memory, L1, L2, and global memory interact?

Take-home message: Performance is dominated by how well data movement is managed across the memory hierarchy.

The GPU memory system spans multiple levels, from fast but limited registers and shared memory to slower but large global memory. Each level offers different tradeoffs between latency, bandwidth, and capacity. Efficient programs minimize expensive global memory accesses by exploiting locality and reuse through registers and shared memory. As a result, optimizing GPU programs is less about reducing arithmetic operations and more about structuring data access patterns to align with the hierarchy.

What decisions are made at runtime?

Concept	Definition
Register allocation	Assigning variables to registers for each thread
Spilling	Moving excess variables to local/global memory when registers are insufficient
Instruction scheduling	Ordering instructions to maximize pipeline utilization and avoid stalls
Warp scheduling	Selecting which warp issues an instruction each cycle
Block scheduling	Assigning thread blocks to SMs for execution
Resource allocation across SMs	Distributing blocks and resources (registers, shared memory) across SMs

What decisions are made at runtime?

Concept	Definition	When decided	Where executed	Scope
Register allocation	Assigning variables to registers for each thread	Compile time (ptxas)	Compiler	Per thread
Spilling	Moving excess variables to local/global memory when registers are insufficient	Compile time	Compiler + runtime memory access	Per thread
Instruction scheduling	Ordering instructions to maximize pipeline utilization and avoid stalls	Compile time (mostly)	Compiler (ptxas)	Per thread / warp
Warp scheduling	Selecting which warp issues an instruction each cycle	Runtime	Hardware (SM warp schedulers)	Per warp
Block scheduling	Assigning thread blocks to SMs for execution	Runtime (kernel launch)	Hardware scheduler	Per block
Resource allocation across SMs	Distributing blocks and resources (registers, shared memory) across SMs	Runtime	Hardware + launch configuration	Per SM

What decisions are made at runtime?

Take-home message: GPU execution is the result of a collaboration between compile-time resource allocation and runtime scheduling.

Some decisions, such as register allocation and instruction layout, are fixed at compile time to ensure efficient execution. Others, such as warp scheduling and block distribution, are handled dynamically by hardware at runtime. This division creates a two-level control system where performance depends on both static decisions made by the compiler and dynamic decisions made by the scheduler. Neither alone is sufficient; efficiency emerges from their interaction.

Computation and storage resource trade-offs?

SM occupancy reflects a tradeoff between per-thread resource usage, i.e., registers/shared memory for computation efficiency, and the number of concurrent threads for latency hiding and throughput.

Occupancy optimization: computation and storage resource trade-offs?

- More shared memory can improve reuse but reduces the number of blocks that can run concurrently, affecting occupancy.

- More registers reduce spilling and improve instruction-level parallelism, but excessive usage reduces occupancy. The goal is to balance these effects for maximum throughput.

- Higher occupancy is beneficial for latency-bound kernels, but for compute-bound kernels with high instruction-level parallelism, lower occupancy can still achieve high performance.

How do these mechanisms affect kernel design?

Parallelism is limited not by thread count but by available on-chip resources such as registers and shared memory.

- They require careful management of resources, balancing register usage, shared memory, and occupancy to achieve optimal performance.
- Because predictable reuse patterns can be exploited more efficiently by software than by hardware.
- They lead to designs that maximize locality, minimize memory traffic, and carefully balance resource usage.

How do these mechanisms affect kernel design?

- Use multiples of 32 threads per block (match the warp size).
- Typical block size: 128–256 threads for good SM utilization.
- Launch many more blocks than SMs to ensure load balancing.
- Match block dimensions to data layout (e.g., 16×16 for matrices).
- Be mindful of shared memory usage, which limits blocks per SM.
- Watch register usage, since high register pressure reduces occupancy.
- Ensure the grid covers the entire dataset (gridDim ≈ ceil(N / blockDim)).
- Expose enough parallelism so the GPU always has work to schedule.

Final Synthesis

Take-home message: GPU computing is about orchestrating computation, memory, and scheduling within a constrained and hierarchical system.

Taken together, these ideas form a unified perspective: a GPU is a resource-constrained, latency-hiding machine where performance depends on how well computation, storage, and concurrency are balanced. The programming model exposes this structure rather than hiding it, requiring developers to reason explicitly about hardware behavior. Mastery of GPU systems therefore comes not from writing parallel code alone, but from understanding how that code interacts with the underlying architecture.

TMEM in Blackwell — A New Data Path for Tensor Compute

Take-home message: TMEM inserts a dedicated on-chip tensor buffer between HBM and Tensor Cores, reducing register pressure and enabling higher utilization.

In Hopper (H100), tensor operands must ultimately reside in registers, which are limited (~256K 32-bit registers per SM) and shared with thread state, creating a hard tradeoff between tile size and occupancy. Blackwell introduces TMEM (Tensor Memory) as a new on-chip storage level designed specifically for tensor workloads. Instead of staging large matrix tiles entirely in registers, data can now flow through a more structured pipeline:

$$\text{HBM} \rightarrow \text{L2} \rightarrow \text{Shared Memory / TMA} \rightarrow \text{TMEM} \rightarrow \text{Tensor Core}$$

TMEM is optimized for high-bandwidth, warp-group-level tensor access, allowing larger tiles to be kept close to compute without consuming excessive registers. This enables: (1) higher effective occupancy by reducing per-thread register demand, (2) better data reuse across tensor operations, and (3) improved pipeline overlap between data movement and compute. Practically, this architectural shift helps explain why Blackwell can achieve large performance gains (e.g., ~50% FP4 throughput increase with only modest core count growth), as efficiency improvements come from dataflow and storage redesign, not just additional compute units.

From GB200 to GB300, why does FP4 performance jumps by 50% (from 10 to 15 PFLOPS) while the physical core count only grows by about 8%?

From GB200 to GB300, why does FP4 performance jumps by 50% while the physical core count only grows by about 8%?

Key Factor	GB200 (Standard)	GB300 (Ultra)	Impact on FP4 Performance
Active SMs	~148	**160** (Full Die)	~8% more raw hardware "engines."
SFU Throughput	16 / SM	**32 / SM**	**Doubles Softmax speed.** This "unblocks" Tensor Cores.
L2 Cache	128 MB	**192 MB**	1.5x larger "on-chip" buffer to feed the cores directly.
HBM Capacity	192 GB	**288 GB**	1.5x more space for weights (reduces off-loading).
HBM Bandwidth	**~8 TB/s**	**~8 TB/s**	**Stayed Stable.** This makes the **L2 Cache** even more critical.
Power (TDP)	1,200W	**1,400W**	Higher power = higher sustained clock frequencies.
Peak FP4 (Dense)	10 PFLOPS	**15 PFLOPS**	**Total Result: 1.5x throughput jump.**

Hierarchy Level	Resource / Unit	GB200 (Blackwell)	GB300 (Blackwell Ultra)	Change / Impact
Per Sub-partition	**CUDA Cores**	32	32	No change (Unified FP32/INT32)
(1 Warp Scheduler)	**LD/ST Units**	8	8	No change (Private resource)
	SFU Units	4	**8**	**Double throughput** for Softmax/Attention.
	Tensor Core	1 (5th Gen)	1 (5th Gen)	Enhanced for native **FP4** precision.
---	---	---	---	---
Per SM (Total)	**CUDA Cores**	128	128	Standard Blackwell density.
(4 Sub-partitions)	**SFU Units**	16	**32**	Fixes the "Attention Bottleneck."
	Tensor Cores	4	4	5th Gen architecture.
	L1/Shared Mem	228 KB	228 KB	High-speed on-chip SRAM.
	TMEM (Tensor)	256 KB	256 KB	Dedicated buffer for Tensor math.
---	---	---	---	---
Full GPU Die	**Active SMs**	~148	**160**	Full die utilization (+8% cores).
	HBM3e Capacity	192 GB	**288 GB**	**1.5x capacity** (12-high stacks).
	L2 Cache	128 MB	**192 MB**	Larger buffer to feed FP4 pipelines.
	Max Power (TDP)	1,200W	**1,400W**	Higher clocks + sustained throughput.
	Peak FP4 (Dense)	**10 PFLOPS**	**15 PFLOPS**	**1.5x total AI performance gain.**