Programming AI hardware

Systems for Machine Learning

Chapter 3

https://www.cse.iitb.ac.in/~mythili/sysml/

Why GPUs?

Why Specialized Hardware?

- ML workloads have high compute, memory demands
  - Trillions of FLOP/s, billions of parameters in memory
- Most ML operations are highly parallelizable
  - Matrix multiplication can be done in parallel
- CPUs not very efficient to run ML workloads
  - Not designed for parallel, memory-intensive workloads
  - Even 1024x1024 matmul takes a long time on CPU!
- Modern ML workloads accelerated with GPUs and other hardware

CPU vs. GPU

Characteristic	CPU	GPU
Core Design	Few complex cores	Many simple cores
Control Hardware	Extensive	Minimal
Parallelism	Limited, higher overhead	Massive, lightweight
Memory bandwidth	Low	High
Optimized for	Irregular memory access, complex control flow	Structured memory access, simpler control flow
Useful for	Sequential, low latency general purpose workloads	Parallel, high throughput numerical computation

CPU vs. GPU

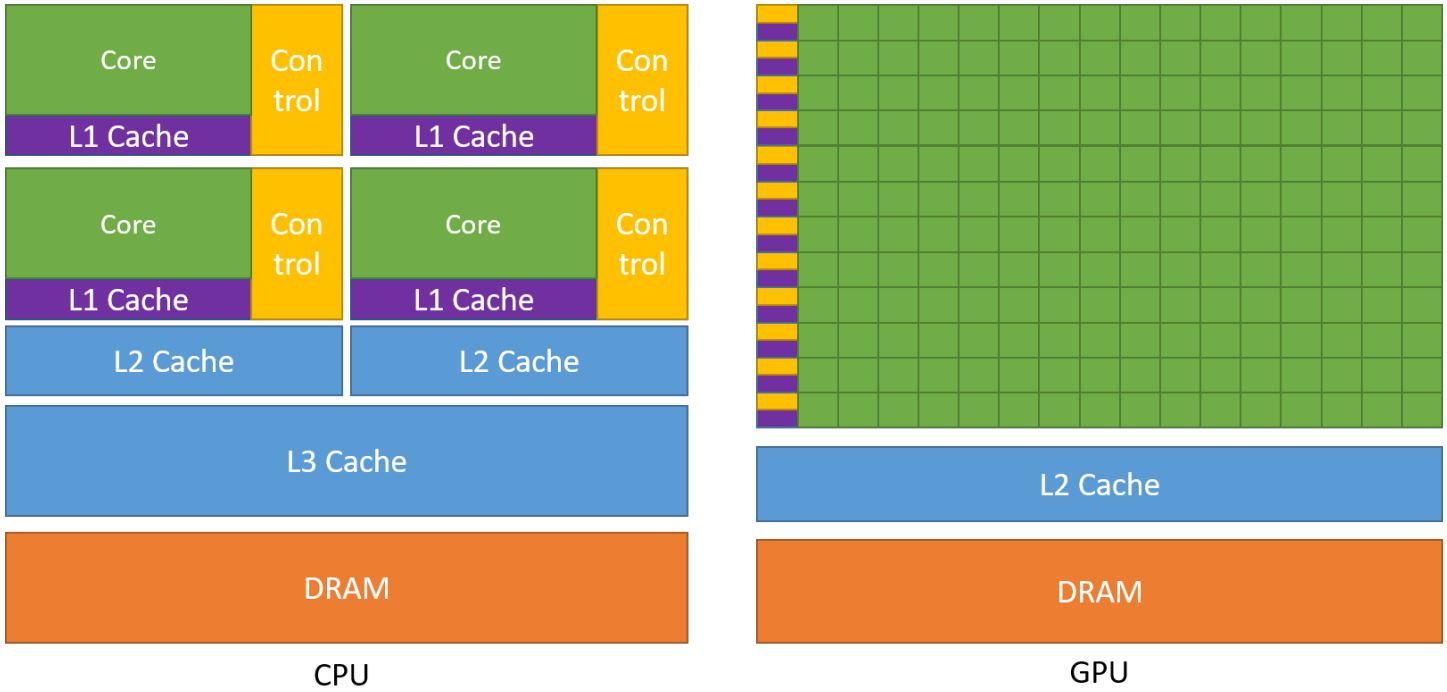


Image Credit: `https://docs.nvidia.com/cuda/cuda-programming-guide/`

Evolution of AI Accelerators

- GPUs (Graphical Processing Units) initially developed for rendering graphics
  - Massive parallel processing across pixels and vertices
- **General purpose GPU (GPGPU)** computing started with NVIDIA's Tesla architecture circa 2008
  - GPUs used for any parallel computation, not just graphics
  - Hundreds of streaming multiprocessors, thousands of lightweight threads
  - C++ based CUDA framework for easy programming
- More optimizations for ML in recent hardware
  - Tensor cores for matrix multiplication

Other AI Accelerators

- Beyond GPGPUs to domain-specific AI accelerators
- Google's **Tensor Processing Units (TPUs)** optimized for matrix multiplication
- Neural Processing Units (NPUs) embedded in mobile devices and laptops for speech recognition and image classification (e.g., Apple's neural engine)
- Other domain-specific accelerators for language processing, graph processing, etc.

Computation Patterns in ML Workloads

- **Vector operations** – same arithmetic operation applied independently across a vector
  - Example: element-wise addition
  - Accelerated via SIMD or SIMT programming
- **Matrix multiplication**
  - Example: activation and weight multiplication in NNs
  - Accelerated via SIMT, tensor cores in GPUs, TPUs
- **Non-linear operations**
  - Example: non-linear activation functions like GeLU
  - Accelerated via Special Function Units (SFUs) in GPUs

SIMD (Single Instruction Multiple Data)

- Single instruction operates on multiple data items at a time in parallel
- Example: load sixteen 32-bit floats into a 512-bit register, perform arithmetic operations on all of them at once using a vector instruction
- Example: Intel's SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions), ARM's SVE (Scalable Vector Extension) instructions
- Highly effective for ML workloads

Code example: SIMD

Code 1: Vector Addition Using 256-Bit Registers

```
1  void vadd(float *a, float *b, float *c,
       int n) {
2       for (int i = 0; i < n; i += 8)
3           _mm256_storeu_ps(&c[i],
                _mm256_add_ps(
4                _mm256_loadu_ps(&a[i]),
5                _mm256_loadu_ps(&b[i])));
6  }
```

stride of 8

eight additions in a single instruction

eight single-precision floats loaded / stored into 256-bit registers

SIMT (Single Instruction Multiple Threads)

- Threads execute the same instruction, but on different data elements
  - Easier than manipulating vector registers
- GPU executes parallel threads in SIMT model
- Well suited for highly parallel ML computations
- Model used by NVIDIA's CUDA
- Programmer specifies the behaviour of a single thread and partitioning of data across different threads

Code example: SIMT

Code 2: Vector Addition Using a Single CUDA Block

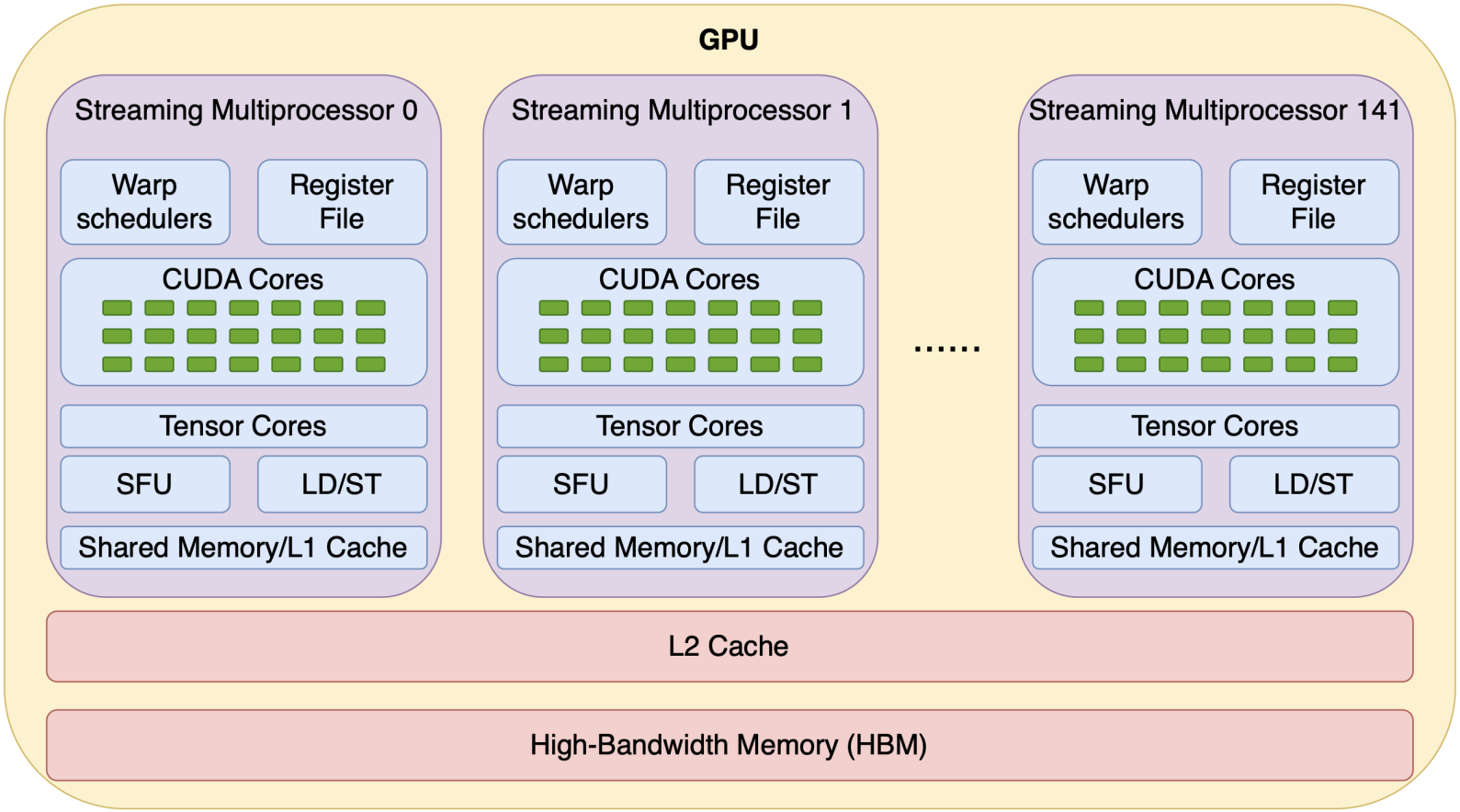
```
1  __global__ void vadd(float *a, float *b,
       float *c) {
2       int i = threadIdx.x;
3       c[i] = a[i] + b[i];
4  }
```

unique thread index identifies data each thread should operate on

same instruction executed by all threads

GPU Architecture

GPU Architecture



GPU Compute

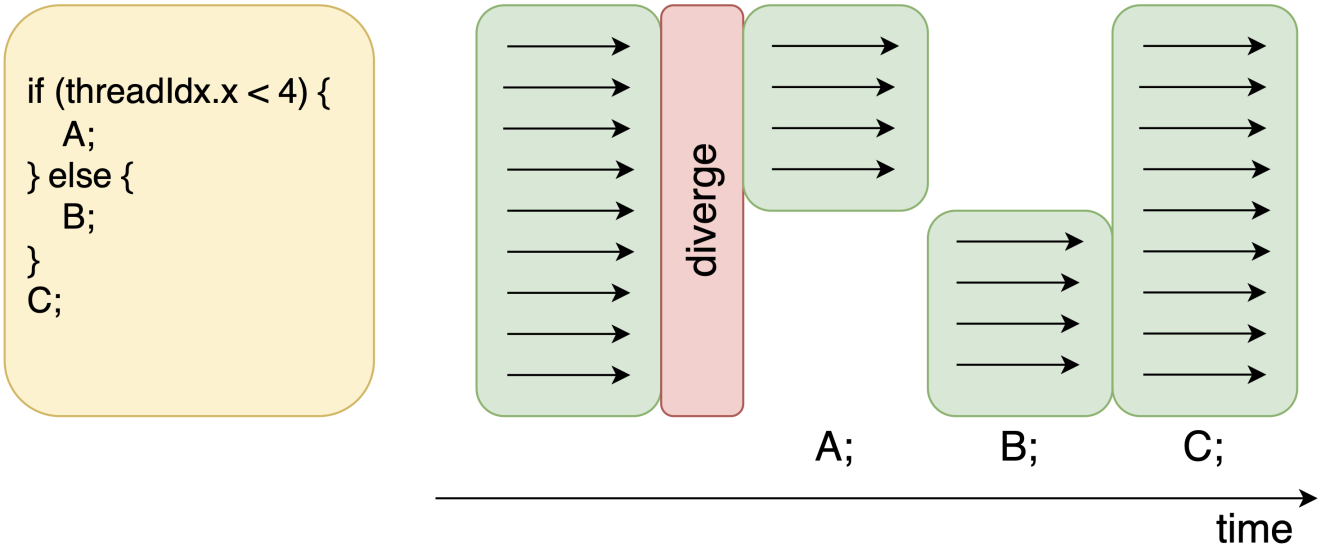
- Hundreds of **Streaming Multiprocessors (SMs)**, each can execute thousands of lightweight **threads**
  - NVIDIA H100 has 132 SMs, each with 2048 threads
- Unlike CPU, SM holds register context of all threads, no switching overheads
- A group of 32 threads is a **warp**, realizes SIMT model
  - Executes same instruction in lockstep on different data
- Warp is the unit of scheduling in the GPU
  - GPU's **warp scheduler** picks ready warps to run from a larger pool of warps, hides memory access latency

CUDA Cores vs. Tensor Cores

- **CUDA cores** - floating-point and integer execution units
  - Executes one int/float operation per clock cycle
- **Tensor cores** - specialized execution units for matrix multiplication
  - Performs a matrix multiply-accumulate on tiles (say, 8x8) in one clock cycle
  - Significantly accelerates large matmuls in deep learning
- Example: NVIDIA H100 GPU has 132 SMs, each with:
  - 128 CUDA cores (can run 4 warps of 32 threads)
  - 4 tensor cores (each can handle compute of full warp)

Warp Divergence

- If threads within warp diverge due to a branch, code paths execute sequentially
  - Some threads masked off each time
- Reduces effective throughput, to be avoided



Compute Throughput of GPU

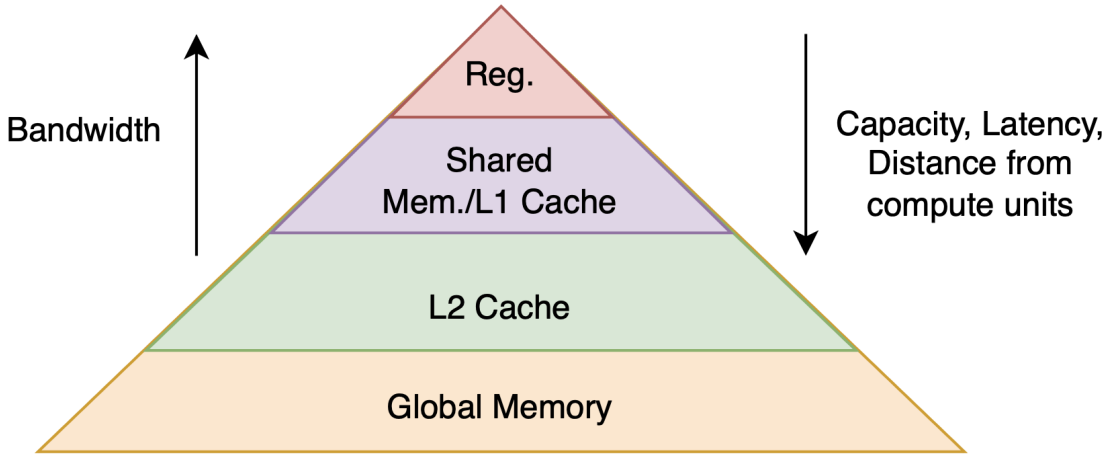
- Performance expressed in **FLOP/s** (floating-point operations per second)
- Consider the NVIDIA's H100 GPU
  - 132 SMs, each with 128 FP32 CUDA cores
  - Clock frequency of 1.98 GHz
  - Peak FP32 FLOP/s from CUDA cores: $132 \times 128 \times 2 \times 1.98 \times 109 = 67$ TFLOP/s
  - Factor of 2 to count for multiply-add as separate operations

Compute Throughput of GPU

- Tensor cores also contribute to GPU's reported FLOP/s
- Consider the NVIDIA's A100 GPU
  - 108 SMs, each can do 512 multiply-add ops per cycle
  - Clock frequency of 1.41 GHz
  - Peak tensor core FLOP/s: $108 \times 512 \times 2 \times 1.41 \times 109 = 156$ TFLOP/s
- Ratio of achieved FLOP/s to theoretical peak FLOP/s is **compute utilization**

Memory Wall

- **Memory wall** – imbalance between growing compute throughput and memory bandwidth, starves arithmetic units of data
- Solution? Small caches close to the compute units to provide faster access



GPU Memory Hierarchy

- **Registers** – private register state per thread inside SM, holds local variables and intermediate values
- **Shared memory** – fast programmer controlled cache, private to each SM
- **L1 cache** – automatically caches frequently accessed data, private to each SM
- **L2 cache** – larger cache shared by all SMs in GPU
- **Global memory** – off-chip, larger capacity, slower access

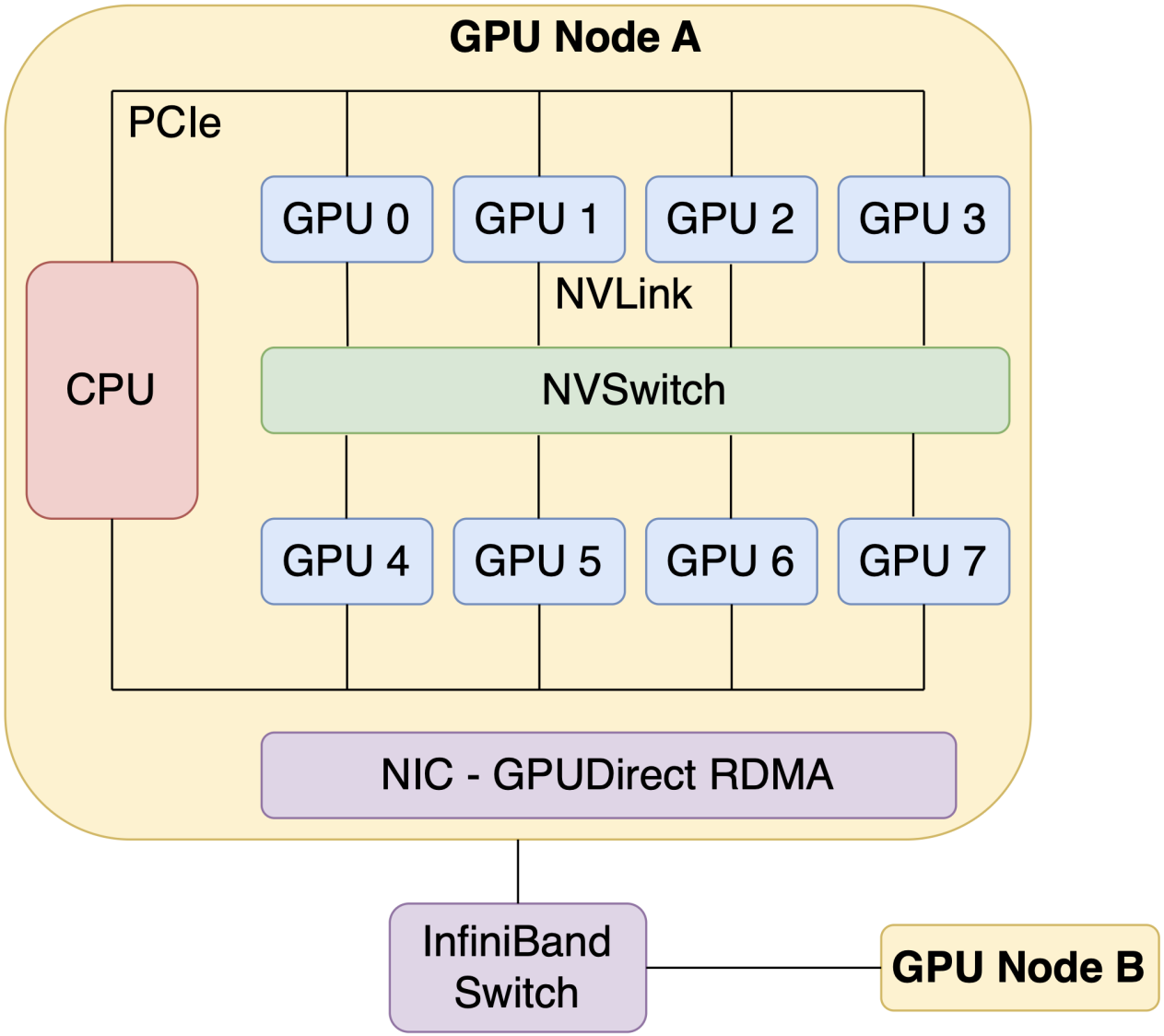
GPU Memory Hierarchy

Memory Tier	Location	Typical Capacity	Approximate Bandwidth	Latency
Registers	On-chip	256 KB per SM	20-70 TB/s	1 clock cycle
Shared Memory/L1	On-chip	64-128 KB per SM	10 TB/s	20-40 clock cycles
L2 cache	On-chip	50-192 MB	3-6 TB/s	100-200 clock cycles
Global Memory (HBM)	Off-chip	16-96+ GB	0.5-3 TB/s	300-500+ clock cycles
System RAM	Off-device	Hundreds of GBs	10-50 GB/s	10,000 cycles

GPU Communication

- **Host-to-device** – GPU (device) to CPU (host) communication via PCI Express (PCIe)
- **Intra-node** – Multiple GPUs in one GPU "node" communicate via NVLink, NVSwitch
- **Inter-node** – Remote Direct Memory Access (RDMA) over Infiniband/Ethernet between GPUs across nodes
  - RDMA avoids CPU intervention in data transfer

GPU Communication



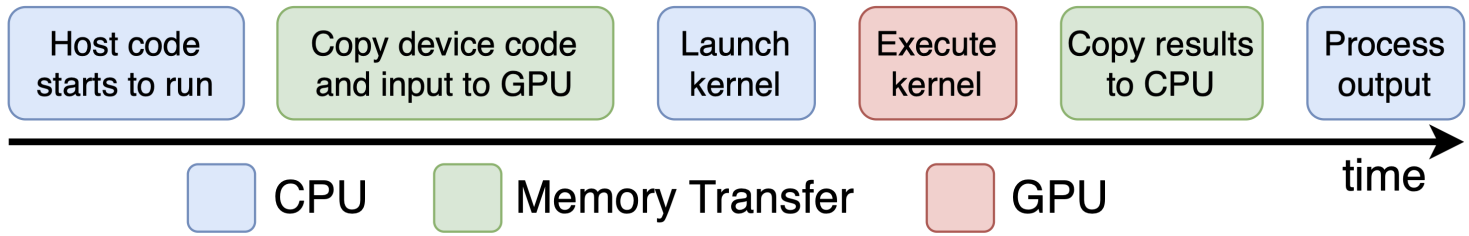
NVIDIA GPU Generations

Generation	SMs	CUDA Cores	Tensor Cores	HBM Capacity	Memory Bandwidth	NVLink Bandwidth
Volta	80	5,120	640	16/32 GB	900 GB/s	300 GB/s
Ampere	108	6,912	432	40/80 GB	1.6–2.0 TB/s	600 GB/s
Hopper	132	16,896	528	80 GB	3.0–3.35 TB/s	900 GB/s
Blackwell	192	24,576	768	192 GB	8 TB/s	1.8 TB/s

CUDA Programming Model

CUDA Programming Model

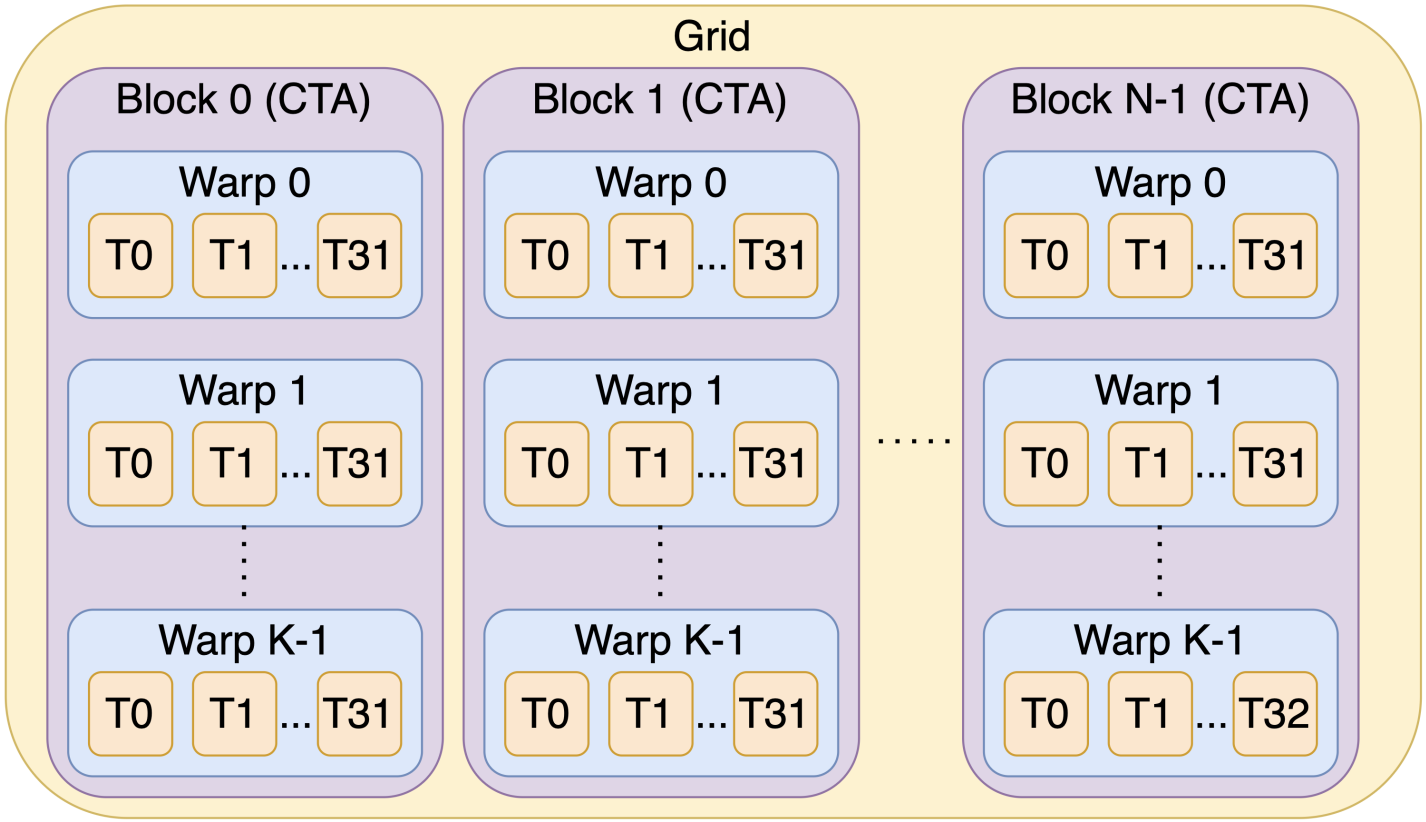
- **CUDA** (Compute Unified Device Architecture) is NVIDIA's programming model for GPU computing
- CUDA **kernel** – C++ function that runs on GPU
- CUDA runtime API handles memory allocation and kernel launches on GPU
- CUDA driver interfaces with GPU hardware



Threads, Blocks, and Grids

- **Thread** – smallest execution unit on GPU
- **Thread blocks** – collection of threads that run on the same SM
  - Generally, up to 1024 threads per block
  - Also known as CTA (Cooperative Thread Array)
- **Grid** – collection of blocks launched for a GPU kernel invocation
- Software hierarchy maps to GPU compute units

GPU Software Hierarchy



The Concept of a Kernel

Code 3: Matrix Addition Using a Single CUDA Block

```
1  // Kernel definition
2  __global__ void MatAdd(float* A, float*
       B, float* C, int N) {
3       int row = threadIdx.y;
4       int col = threadIdx.x;
5       int index = row * N + col;
6       C[index] = A[index] + B[index];
7  }
```

GPU kernel callable from the CPU

mapping between data elements and threads

The Concept of a Kernel

Code 4: Matrix Addition Using a Single CUDA Block

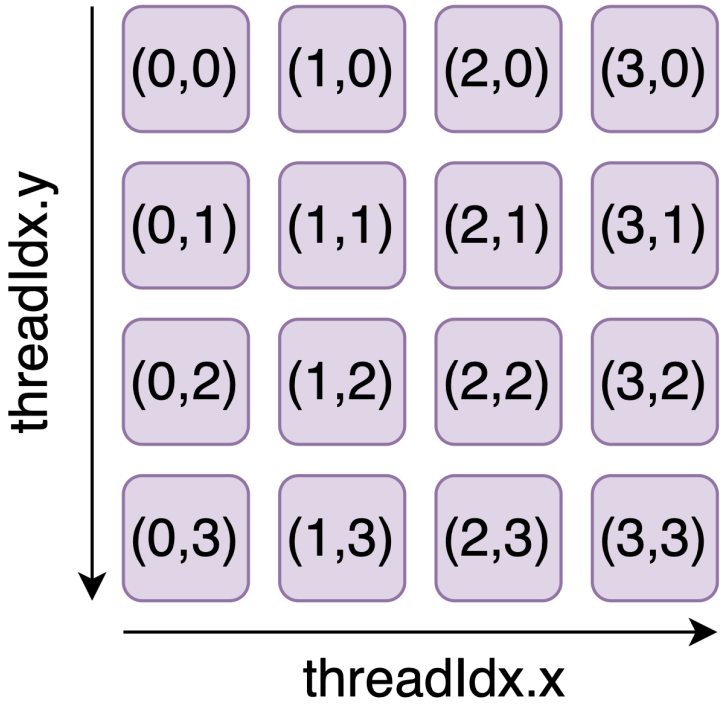
```
1  int main() {
2      // Kernel invocation
3      int numBlocks = 1;
4      int N = 16;
5      dim3 threadsPerBlock(N, N);
6      MatAdd<<<numBlocks,
           threadsPerBlock>>>(A, B, C, N);
7  }
```

number of thread blocks in the grid

number of threads in a block

kernel launch syntax

Mapping Threads to Data Elements



(a) Two-dimensional thread block

C[0,0] t(0,0)	C[0,1] t(1,0)	C[0,2] t(2,0)	C[0,3] t(3,0)
C[1,0] t(0,1)	C[1,1] t(1,1)	C[1,2] t(2,1)	C[1,3] t(3,1)
C[2,0] t(0,2)	C[2,1] t(1,2)	C[2,2] t(2,2)	C[2,3] t(3,2)
C[3,0] t(0,3)	C[3,1] t(1,3)	C[3,2] t(2,3)	C[3,3] t(3,3)

(b) Mapping between threads and elements of matrix C

Built-in Variables

- **`threadIdx`** – unique index of a thread within its block
- **`blockDim`** – dimensions of a thread block
- **`blockIdx`** – unique index of a thread block within the grid
- **`gridDim`** – dimensions of the grid
- Each of these have x, y, z components

Vector Addition Using Multiple Blocks

Code 5: Vector Addition Using Multiple Blocks

```
1  __global__ void vecAdd(float* A, float*
       B, float* C, int N) {
2      int globalIndex = threadIdx.x +
           blockDim.x * blockIdx.x;
3      if (globalIndex < N)
4          C[globalIndex] = A[globalIndex]
               + B[globalIndex];
5  }
```

maps thread and block indices to a global index of the vector

Vector Addition Using Multiple Blocks

Code 6: Vector Addition Using Multiple Blocks

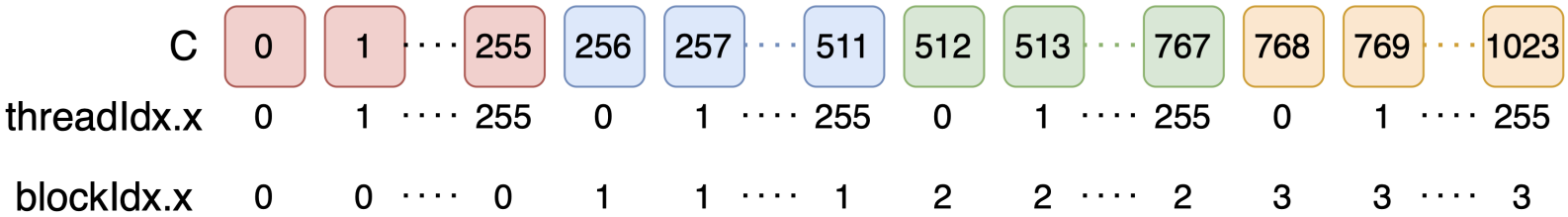
```
1  int main() {
2      int N = 1024;
3      int threadsPerBlock = 256;
4      int numBlocks =
           (N+threadsPerBlock-1) /
           threadsPerBlock;
5      vecAdd<<<numBlocks,
           threadsPerBlock>>>(A, B, C, N);
6  }
```

ceiling division for cases when N is not divisible by number of threads

one-dimensional thread blocks

Vector Addition Using Multiple Blocks

- Code shown earlier creates 4 blocks in the grid, each with 256 threads to process 1024-element array
- Shown below is the thread/block identifier of a thread that processes each vector element



Matrix Addition Using Multiple Blocks

Code 7: Matrix Addition Using Multiple Blocks

```
1  __global__ void matAdd(float* A, float*
       B, float* C, int N) {
2       int row = blockIdx.y * blockDim.y +
            threadIdx.y;
3       int col = blockIdx.x * blockDim.x +
            threadIdx.x;
4       if (row < N && col < N) {
5           int index = row * N + col;
6           C[index] = A[index] + B[index];
7       }
8  }
```

Thread maps to row and col based on its identifiers

Matrix Addition Using Multiple Blocks

Code 8: Matrix Addition Using Multiple Blocks

```
1  int main() {
2      dim3 threadsPerBlock(16, 16);
3      dim3 numBlocks((N+15)/16, (N+15)/16);
4      matAdd<<<numBlocks,
           threadsPerBlock>>>(A, B, C, N);
5  }
```

two-dimensional thread blocks

Matrix Multiplication

Code 9: Matrix Multiplication Using Multiple Blocks

```
1  __global__ void matMul(float* A, float*
       B, float* C, int N) {
2       int row = blockIdx.y * blockDim.y +
            threadIdx.y;
3       int col = blockIdx.x * blockDim.x +
            threadIdx.x;
4       if (row < N && col < N) {
5           float sum = 0.0f;
6           for (int k = 0; k < N; k++)
7               sum += A[row*N+k]*B[k*N+col];
8           C[row * N + col] = sum;}}
```

Memory Coalescing

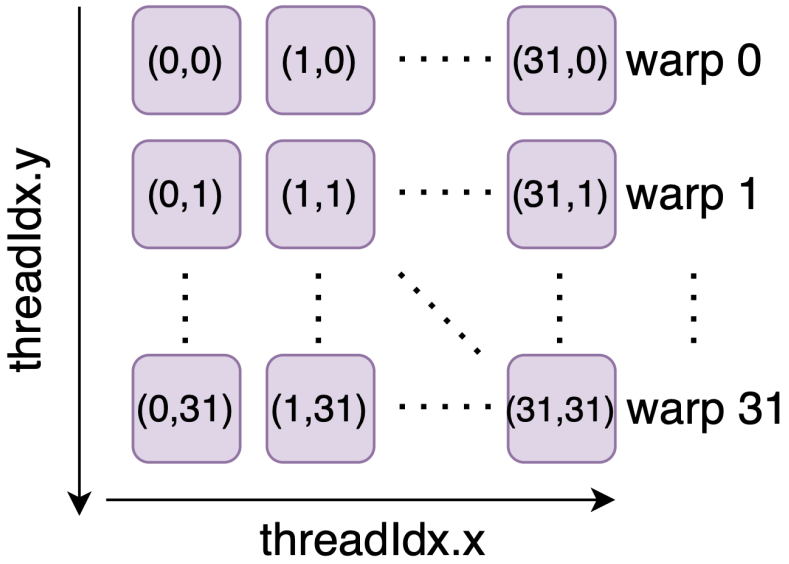
- Global memory has high bandwidth, high latency
  - Separate requests lead to under-utilization of bandwidth
- Hardware fetches multiple consecutive addresses together on every access, in a **DRAM burst**
- **Coalesced** memory access pattern when threads in a warp access consecutive addresses
- Serviced by single DRAM burst, better performance

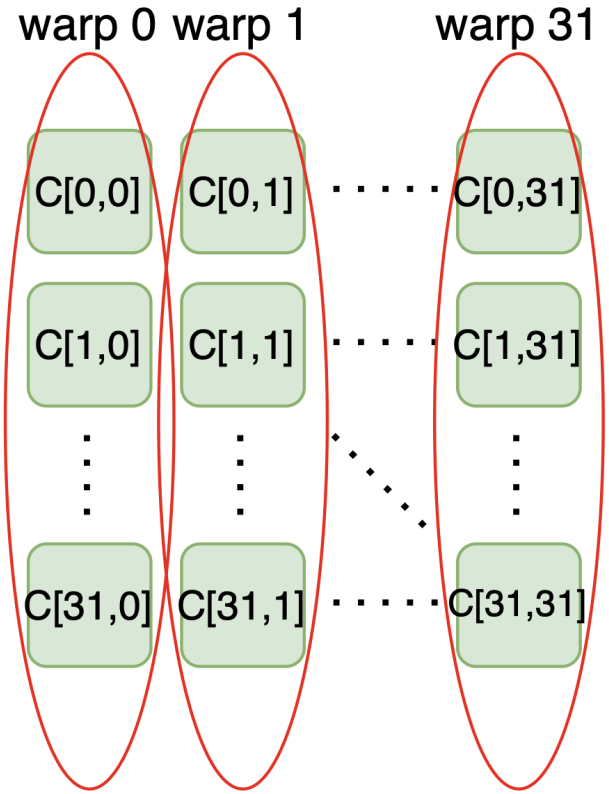
Uncoalesced Matrix Addition

Code 10: Uncoalesced Matrix Addition

```
1  __global__ void uncoalesced_matAdd
       (float* A,float* B,float* C,int N) {
2       int col = blockIdx.y * blockDim.y +
            threadIdx.y;
3       int row = blockIdx.x * blockDim.x +
            threadIdx.x;
4       if (row < N && col < N) {
5           int index = row * N + col;
6           C[index] = A[index] + B[index];
7       }
8  }
```

Uncoalesced Matrix Addition



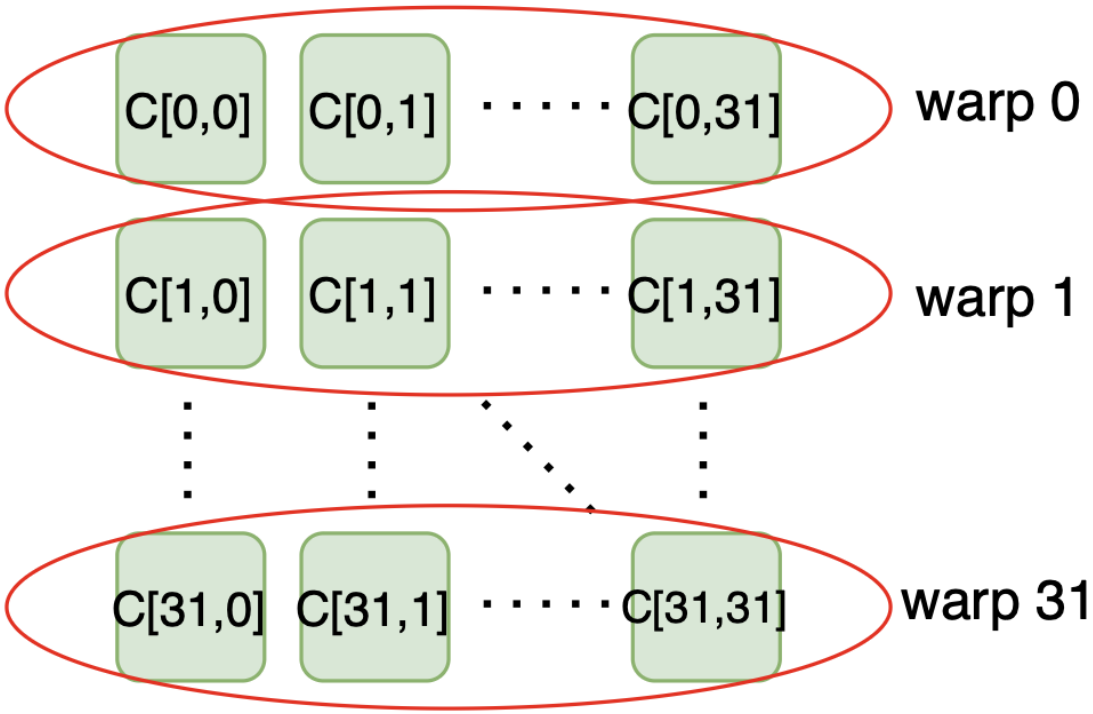


Coalesced Matrix Addition

Code 11: Coalesced Matrix Addition

```
1  __global__ void coalesced_matAdd(float*
       A, float* B, float* C, int N) {
2      int row = blockIdx.y * blockDim.y +
           threadIdx.y;
3      int col = blockIdx.x * blockDim.x +
           threadIdx.x;
4      if (row < N && col < N) {
5          int index = row * N + col;
6          C[index] = A[index] + B[index];
7      }
8  }
```

Coalesced Matrix Addition



CUDA APIs

CUDA Memory APIs

- **cudaMallocHost** - page-locked (pinned) memory allocated on host, freed by **cudaFreeHost**
- **cudaMalloc** - memory allocated on GPU, freed by **cudaFree**

Code 12: Allocating Device Memory

```
1  float* d_A;
2  int N = 1024;
3  cudaMalloc((void**)&d_A,
       N*sizeof(float));
```

after allocation d_A points to device memory

CUDA Memory APIs

- **`cudaMemset`** – sets device memory bytes to a specific value
- **`cudaMemcpy`** – copies data between host and device memory
- **`cudaMemcpyAsync`** – asynchronous transfer of data between page-locked host memory and device memory
  - Enables concurrent kernel execution and data transfer

Unified Virtual Memory (UVM)

- APIs expose CPU memory and GPU memory as a single virtual address space
- Memory pages automatically transferred between CPU and GPU memory upon access

Code 13: Usage of Unified Memory

```
1  float* A;
2  cudaMallocManaged(&A, N * sizeof(float));
3  for (int i = 0; i < N; i++) A[i] = i;
4  kernel<<<numBlocks,
       threadsPerBlock>>>(A);
5  cudaDeviceSynchronize();
6  printf("%f\n", A[0]);
7  cudaFree(A);
```

same pointer accessible from both CPU and GPU

Shared Memory

- Programmer-managed fast on-chip cache
- Shared amongst threads of the same block

Code 14: Static Shared Memory

```
1  __global__ void kernel(float* A) {
2      __shared__ float cache[256];
3      int idx = threadIdx.x;
4      cache[idx] = A[idx];
5      A[idx] = cache[idx] * 2.0f;
6  }
```

shared across a thread block

CUDA Function Specifiers

Specifier	Executes on	Called from	Return type
`__global__`	GPU	CPU	`void` only
`__device__`	GPU	GPU only	Any
`__host__`	CPU	CPU only	Any
`__host__ __device__`	CPU and GPU	CPU and GPU	Any

Thread Synchronization

Code 15: Thread Synchronization

```
1  __global__ void kernel(int* input_data,
       int* output_data) {
2      __shared__ int shared_data[128];
3      shared_data[threadIdx.x] =
           input_data[threadIdx.x];
4      __syncthreads();
5      if (threadIdx.x == 0) {
6          int sum = 0;
7          for (int i=0;i<blockDim.x;++i)
8              sum += shared_data[i];
9          output_data[blockIdx.x] = sum;
10     }
11 }
```

blocks execution until all threads in a block reach this point

thread-0 performs summation after all threads are done

CUDA Events

- For accurate timing measurements

Code 16: CUDA Events

```
1  cudaEvent_t start, stop;
2  cudaEventCreate(&start);
3  cudaEventCreate(&stop);
4  cudaEventRecord(start);
5  kernel<<<blocks, threads>>>();
6  cudaEvenRecord(stop);
7  cudaEventSynchronize(stop);
8  float ms;
9  cudaEventElapsedTime(&ms, start, stop);
```

Error Detection

Code 17: Basic CUDA Error Checking

```
1  cudaError_t err;
2  err = cudaMalloc((void**)&d_A, N *
       sizeof(float));
3  if (err != cudaSuccess) {
4      printf("CUDA Error: %s\n",
5      cudaGetErrorString(err));
6  }
```

indicates success/failure

converts error codes to strings

Error Detection

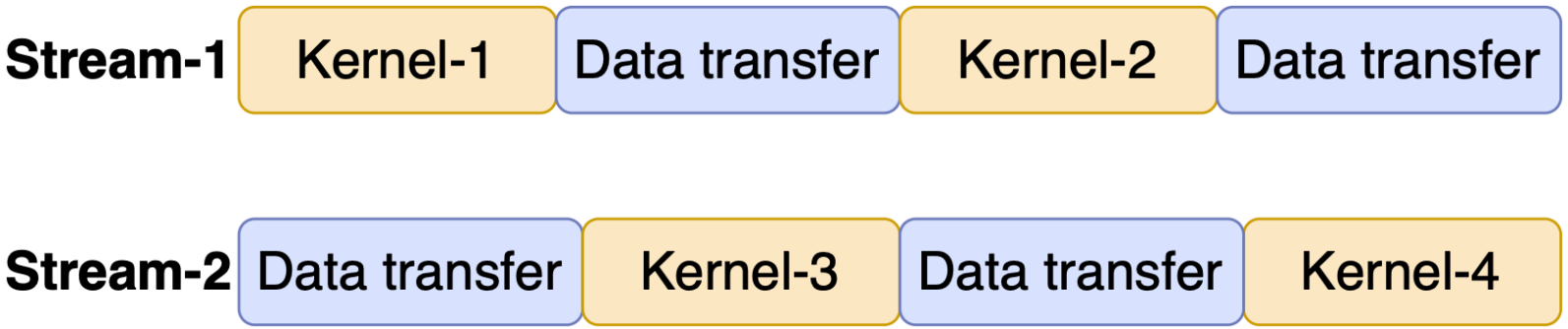
Code 18: Kernel Error Detection

```
1  kernel<<<blocks, threads>>>(A);
2  cudaError_t err = cudaGetLastError();
3  if (err != cudaSuccess) {
4      printf("Launch Error: %s\n",
5      cudaGetErrorString(err));
6  }
```

to capture error in async kernel launch

CUDA Streams

- Kernel launches, memory copies issued in a default execution **stream**
- Operations in one stream execute one after the other
- Operations in different streams may execute concurrently
- Decomposes tasks into multiple parallel execution pipelines



CUDA Streams

Code 19: Creating CUDA Streams

```
1 cudaStream_t stream1, stream2;
2 cudaStreamCreate(&stream1);
3 cudaStreamCreate(&stream2);
```

Code 20: Concurrent Stream Execution

```
1 cudaMemcpyAsync(..., stream1);
2 kernelA<<<..., stream1>>>();
3
4 cudaMemcpyAsync(..., stream2);
5 kernelB<<<..., stream2>>>();
```

Atomics and Synchronization

- `cudaDeviceSynchronize` – blocks the CPU until all previously launched GPU work completes
- Atomic operations allow multiple threads to update shared variables without race conditions
  - Example: `atomicAdd`, `atomicSub`, `atomicCAS`, `atomicMin`, `atomicMax`

CUDA Compilation Workflow

CUDA Compilation

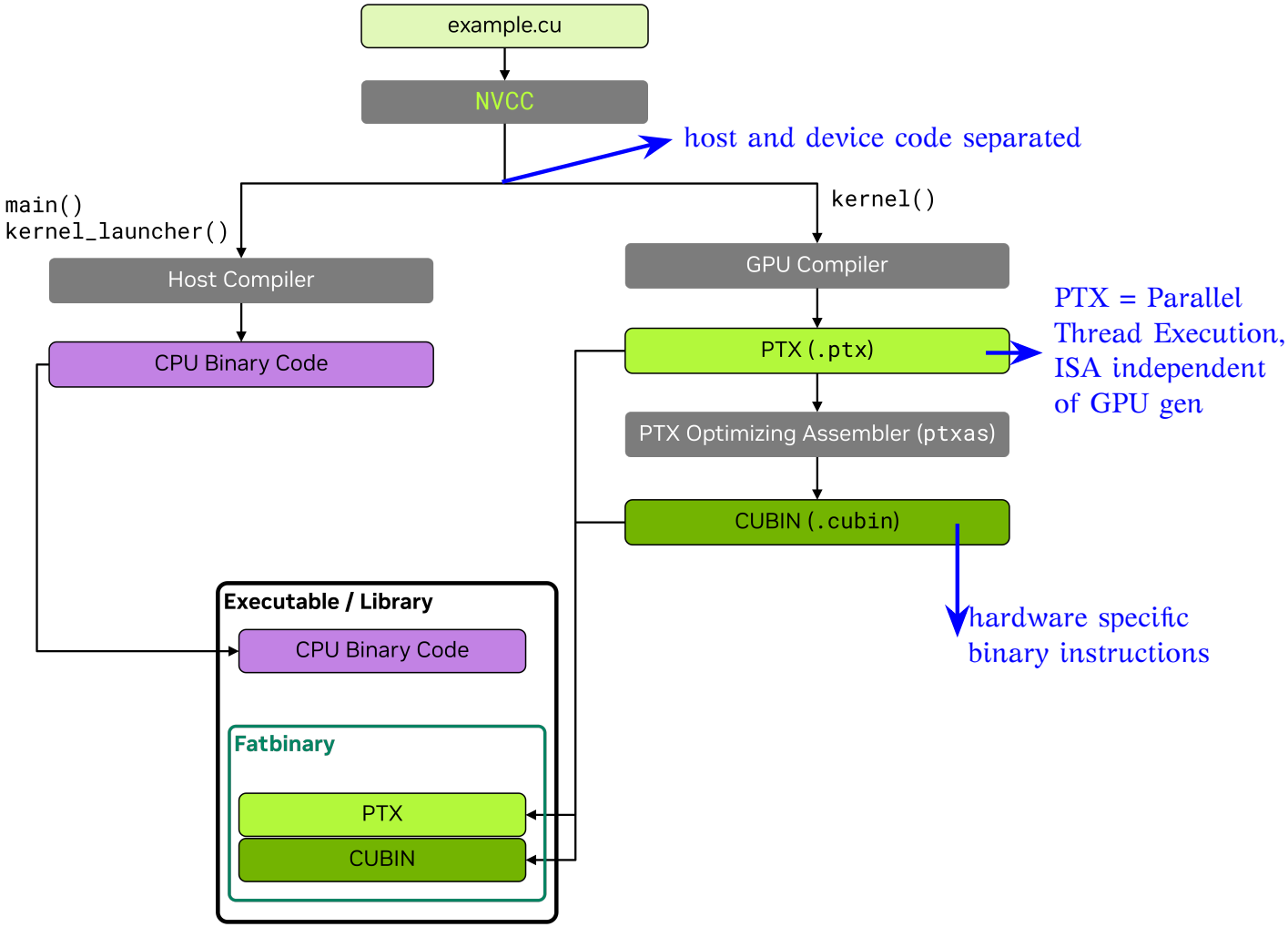


Image Credit: `https://docs.nvidia.com/cuda/cuda-programming-guide/`

Mapping Software Abstractions to Hardware

Software Abstractions to Hardware

Software	Hardware
Grid (all blocks from one kernel launch)	Entire GPU (distributed across all SMs)
Thread block (CTA)	Streaming multiprocessor (always on one SM, never splits)
Warp (32 threads)	Warp scheduler (issues one instruction for all 32 threads)
Thread (smallest execution unit)	Instructions issued to CUDA/tensor cores

Thread Block Assignment to SMs

- Once SM has enough free resources, scheduler dispatches new block from the grid to execute
- **Most-room policy** – pick SM with largest amount of free resources
- **Least-room policy** – pack into already used SMs whenever possible

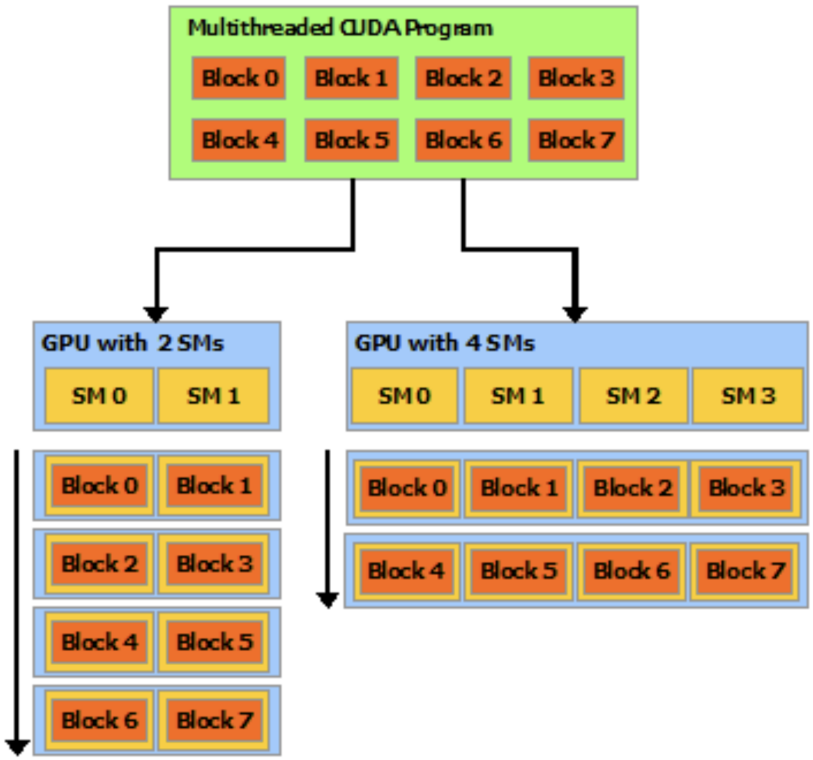
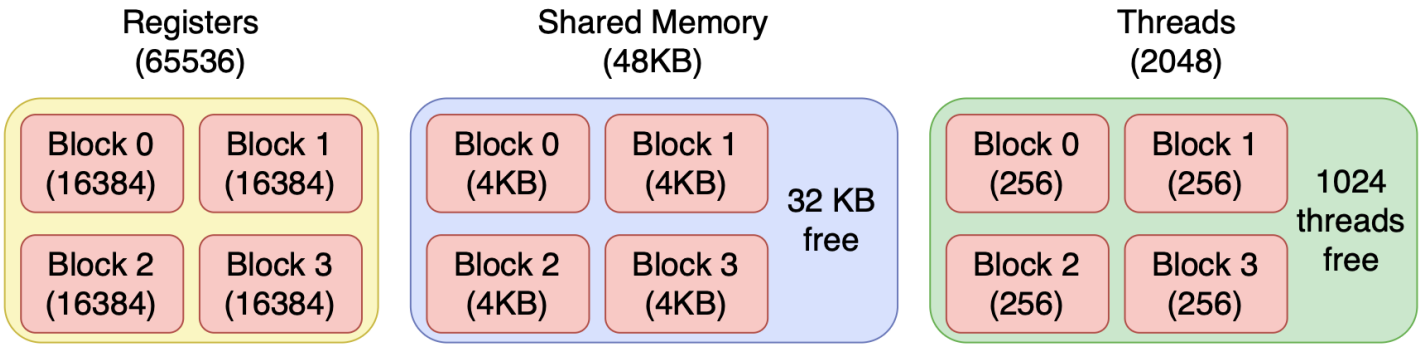


Image Credit: `https://docs.nvidia.com/cuda/cuda-programming-guide/`

SM Occupancy

- **SM Occupancy** – fraction of maximum possible warps that are resident in the SM
- Constrained by whichever hardware resource (threads, registers, shared memory capacity) is exhausted first

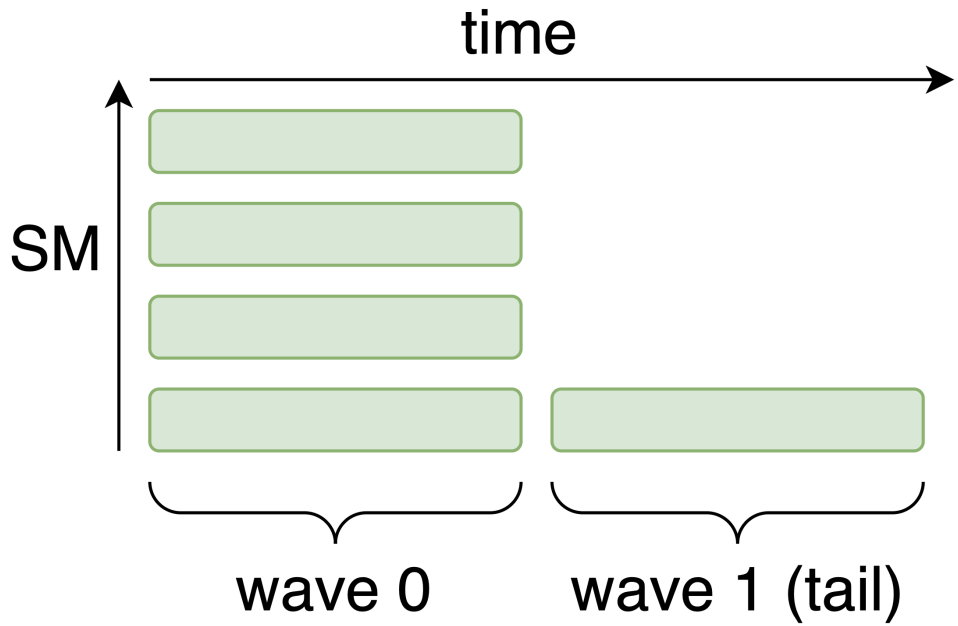


Exercise: SM Occupancy

Find the SM occupancy for the above example.

Wave Quantization

- **Wave quantization** – when number of blocks not divisible by number of SMs, a few SMs do not reach peak capacity in the last waves
- Tail effects also caused by differences in execution times across blocks



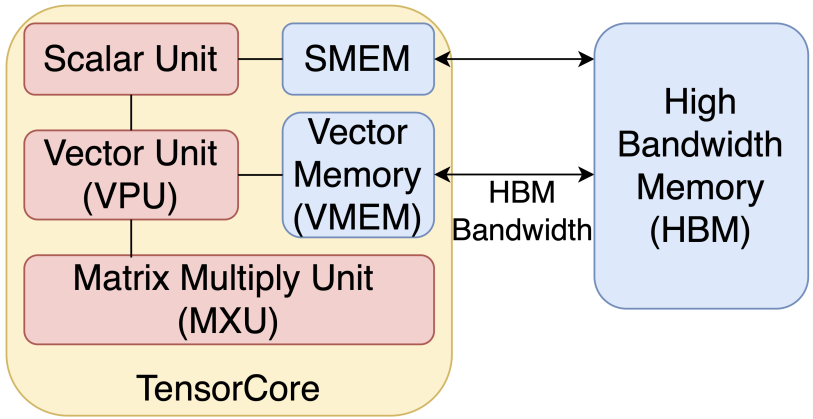
Warp Scheduling

- Warps **ready** when operands fetched from memory, stalled otherwise
- **Warp schedulers** select subset of ready warps for instruction issue
- Register context remains resident on-chip even when warp is not actively running
- Large number of warps hide memory access latencies
- Different warp scheduling policies
  - Example: round robin, least recently fetched, fair, etc.

Tensor Processing Units (TPUs)

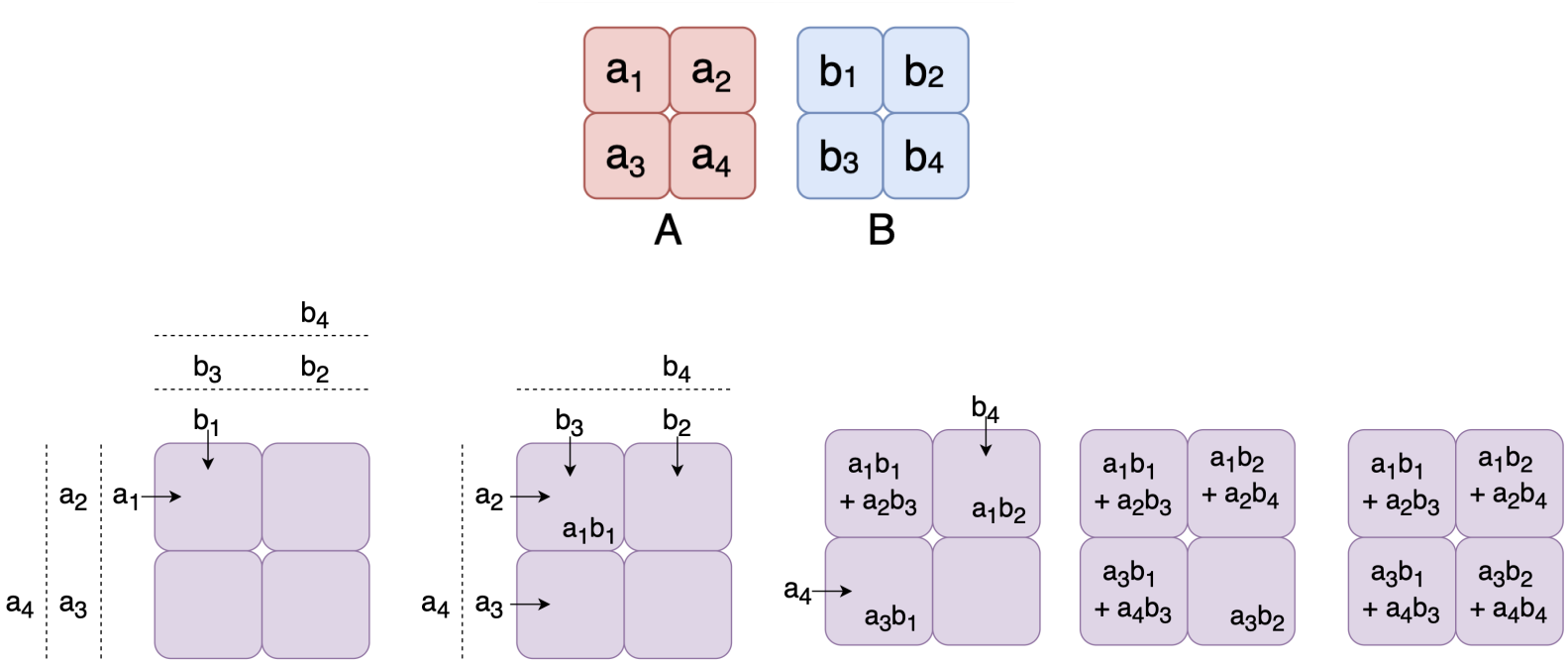
Tensor Processing Units (TPUs)

- Optimized for tensor algebra execution
- Components: **TensorCore** and **HBM**
- MXU optimized for matrix multiplications using systolic arrays
- Data movement to compute units via caches (e.g., VMEM)



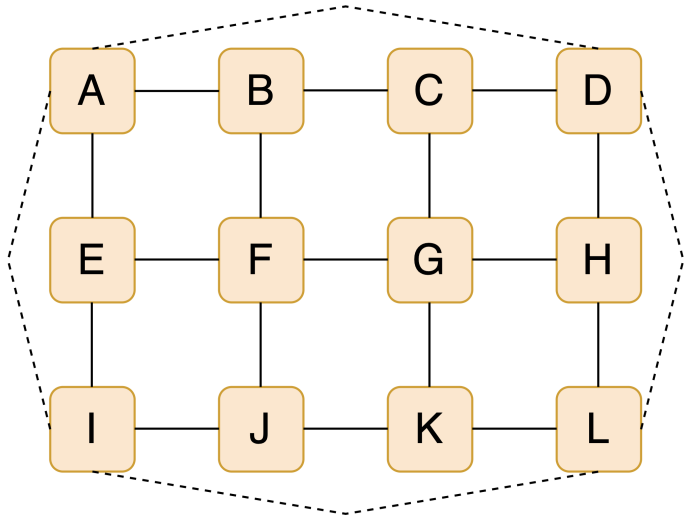
Systolic Array

- Performs matmul efficiently every few cycles
- Matrices flow in to processing elements, which accumulate partial sum
- More efficient than reading, writing matrices



TPU Networking

- Unlike GPUs, TPUs rely on nearest-neighbour communication
- Inter-Chip Interconnect (ICI) – TPUs inside a pod connected via 2D/3D torus
- Datacenter Network (DCN) – Communication between TPU pods via PCIe and CPU



CPU vs. GPU vs. TPU

Category	CPU	GPU	TPU
Core count	8–128 powerful cores	Thousands of CUDA and Tensor cores	Systolic array (128 × 128 MXU)
Execution model	Complex components (OOO execution)	SIMT; warps hide stalls	Dataflow; operands pass between units
Memory bandwidth	50–100 GB/s	1–3.5 TB/s (HBM2/3)	2–3 TB/s
Compiler burden	Low	High	Very high
Peak power	5–30 W	400–700 W	200–450 W
Irregular ops	Handles easily	Handles well	Poor; systolic array stalls
Ecosystem	Universal; any language or framework	CUDA, cuDNN, Triton	JAX, TensorFlow / XLA; limited low-level access

Advanced Hardware Features

Advanced Hardware Features

- **Tensor cores** – for matrix multiplication operations
  - Example: $D = A \times B + C$
- **Tensor Memory Accelerator (TMA)** – asynchronously fetches multi-dimensional arrays from global memory to shared memory
- **Warpgroup Matrix-Multiply-Accumulate (WGMMA)** – allows a group of warps to issue asynchronous instructions to tensor cores
- **Warp specialization** – Warps specialized to coordinate asynchronous compute and memory operations in a producer-consumer pattern