

LLM Sys

GPU Programming

Lei Li



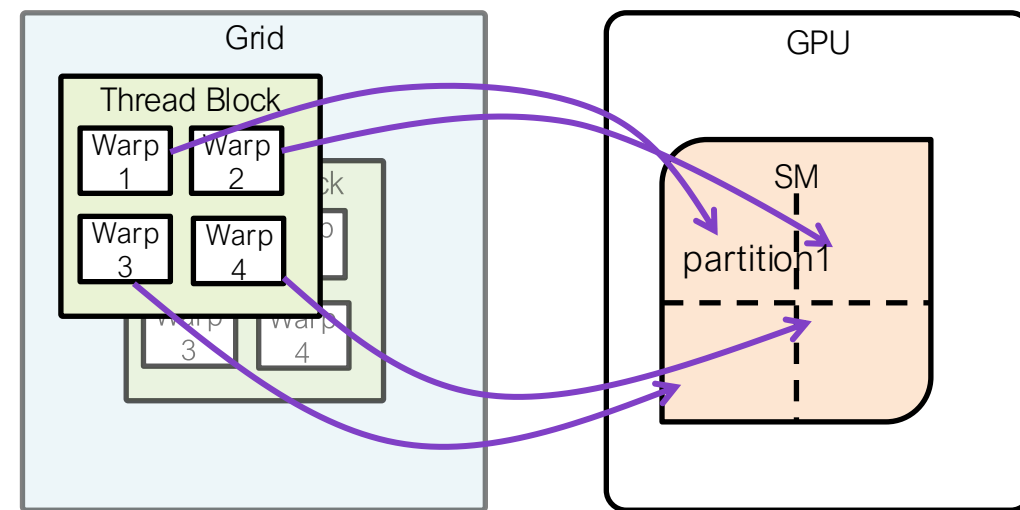
Language
Technologies
Institute

Carnegie Mellon University

School of Computer Science

Recap

- GPU is composed of
 - streaming processing units (SMs)
 - each with four partitions of 32 cores
 - shared L1 cache
 - memory
 - L2 cache: share with all SMs
- Threads organized in
 - grid of thread blocks
 - each block is divided into warps running on one SM.



H100 Architecture

132/114 SMs, 80G HBM3



H100 SM

4 partitions per SM, each with 32 cores → 32 threads each

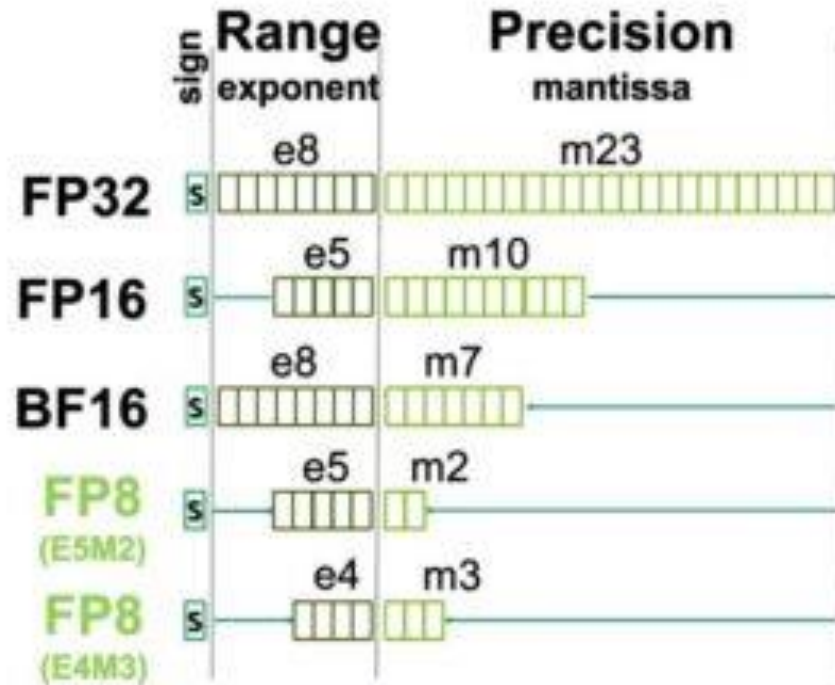
128 cores per SM

64KB register per partition (fastest to access)

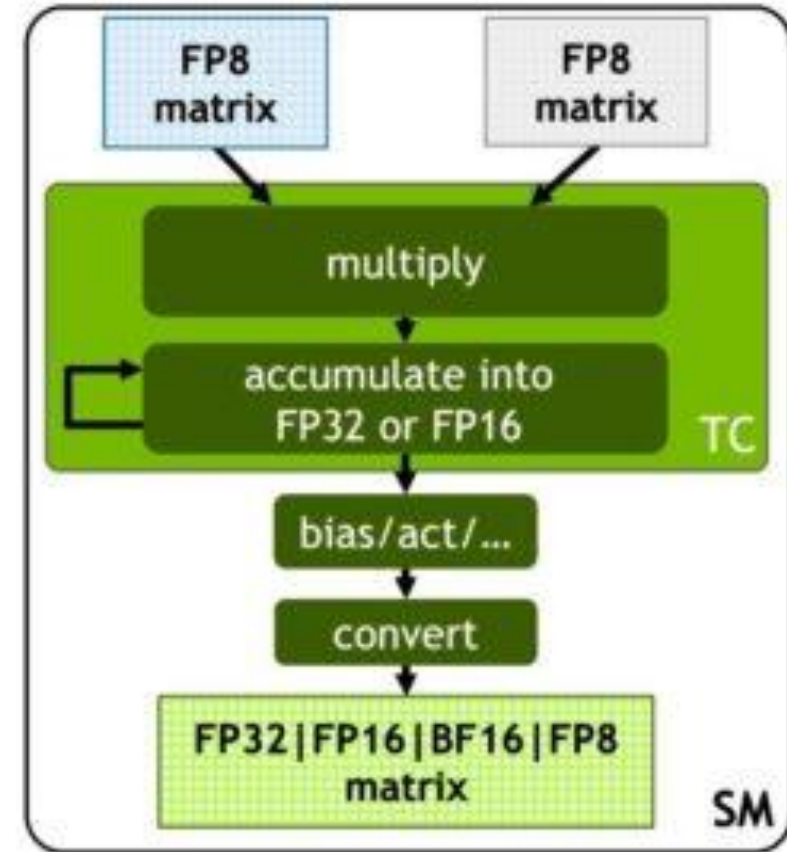
256KB shared L1 cache per SM



New in H100, FP8 operations



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

Assignment 1

https://lmsystem.github.io/lmsystemhomework/assignment_1/

recommend to use PSC. Other platform is not guaranteed.

Due date: Jan 28, 2026.

Late policy:

<https://lmsystem.github.io/lmsystem2026spring/docs/Logistics#late-day-policy>

Course Project

Reminder: start form your project team (2-3 students)

Submit your team member names by 2/18/2026

project proposal due 2/27/2026

Initial ideas for project (feel free to form your own):

[https://llmsystem.github.io/llmsystem2026spring/docs/Project
S](https://llmsystem.github.io/llmsystem2026spring/docs/Project_S)

Outline

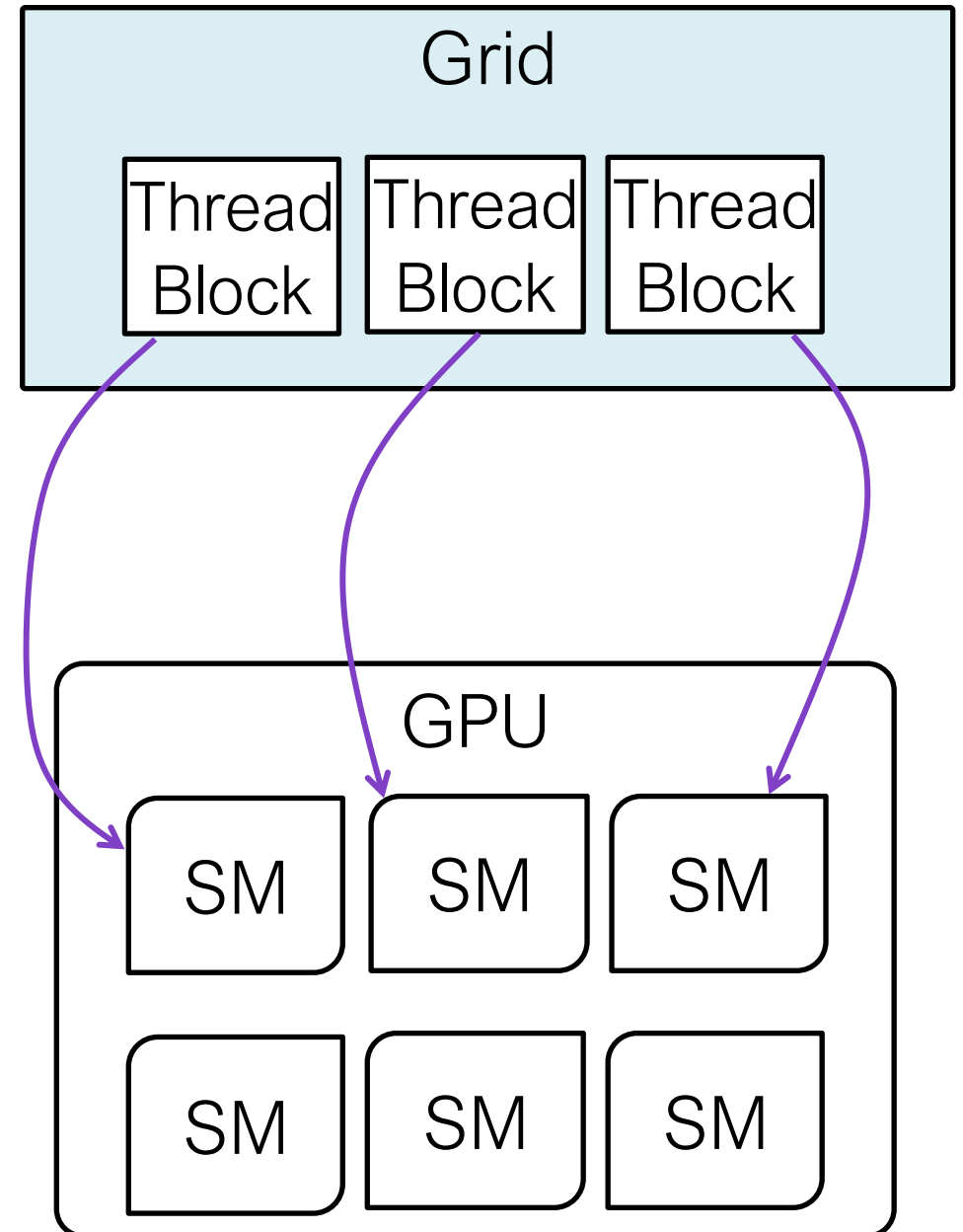
- ➔ • Basic GPU CUDA operations
 - memory management
 - creating threads
 - defining kernel functions for arithmetics
- Matrix/Tensor Computation on GPU

CUDA Kernel

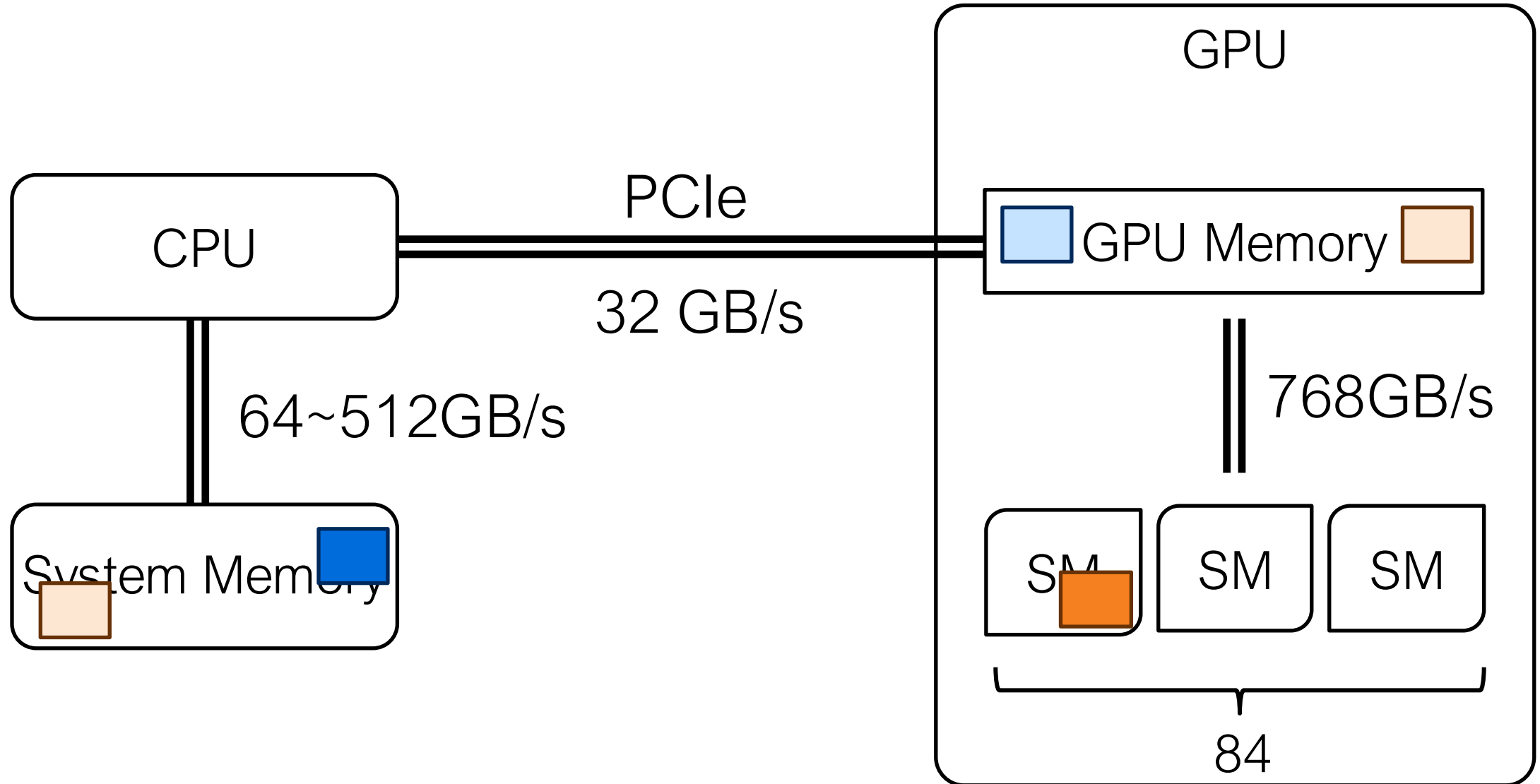
- Each kernel is a function (program) that runs on GPU
- Program itself is serial
- Can simultaneously run many (10k) threads at the same time
- Using thread index to compute on right portion of data

Running GPU kernel

- CPU invokes kernel grid
- Thread blocks in grid distributed to SMs
- Execute concurrently
 - Each SM runs multiple thread blocks
 - Each core runs one thread from one thread block



CPU-GPU Data Movement



CUDA Operations

- CPU allocates GPU memory: `cudaMalloc`
- CPU copies data to GPU memory (host to device): `cudaMemcpy`
- CPU launches GPU kernels
- CPU copies results from GPU (device to host): `cudaMemcpy`
- Freeing GPU memory `cudaFree`

Allocate GPU Memory

`cudaError_t cudaMalloc(void** devPtr, size_t size)`

`devPtr`- Pointer to allocated device memory

`size`- Requested allocation size in bytes

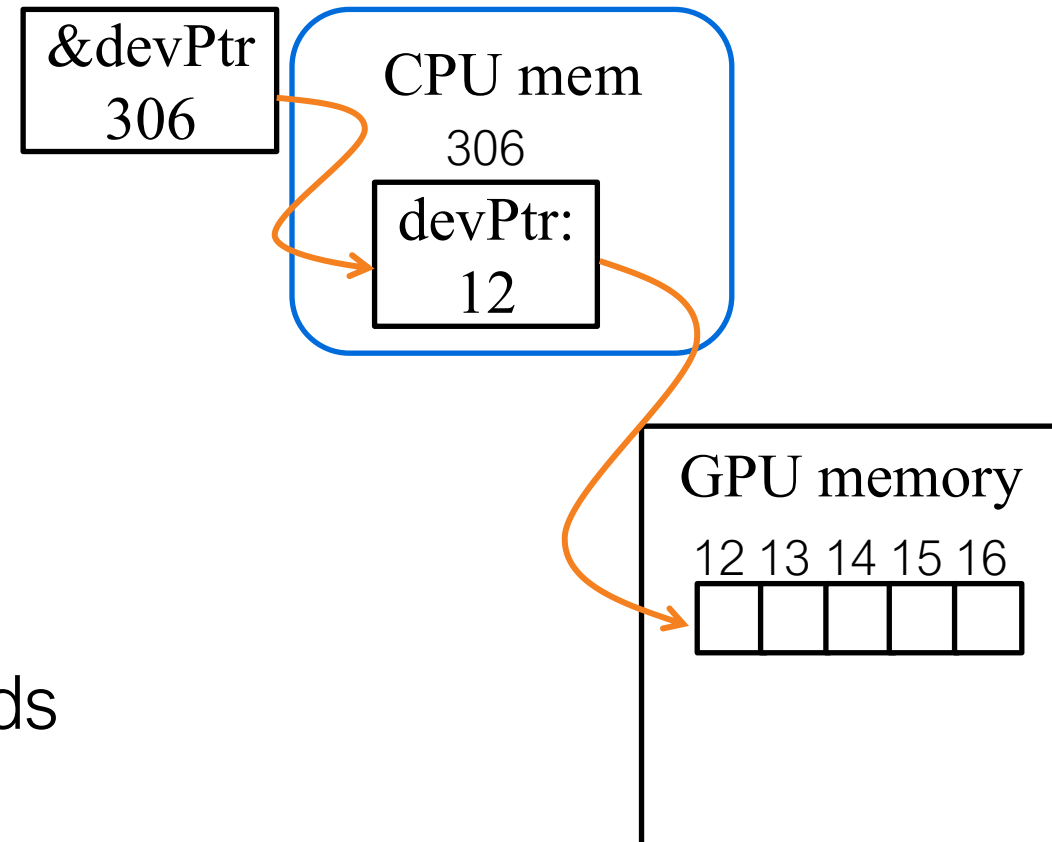
```
int *dA;
```

```
cudaMalloc(&dA, n * sizeof(int));
```

```
float *dB;
```

```
cudaMalloc(&dB, n * sizeof(float));
```

The allocated mem is accessible by all threads



Free GPU memory

- donot forgot!

```
cudaError_t cudaFree(void *devPtr);
```

Parameters

devPtr: A device pointer to the memory you want to free.

GPU memory

- Each thread has private registers (fastest to access)
- Each thread block has shared memory
 - Visible to all threads in a block
 - `__shared__`
- All threads can access global gpu memory
 - Persistent across kernel launches in the same app



Data Movement

Copy data from devices: cpu to gpu, gpu to cpu

```
cudaError_t cudaMemcpy(void* dst, const void* src,  
size_t count, cudaMemcpyKind kind)
```

Parameters:

dst: Destination memory address

src: Source memory address

count: Size in bytes to copy

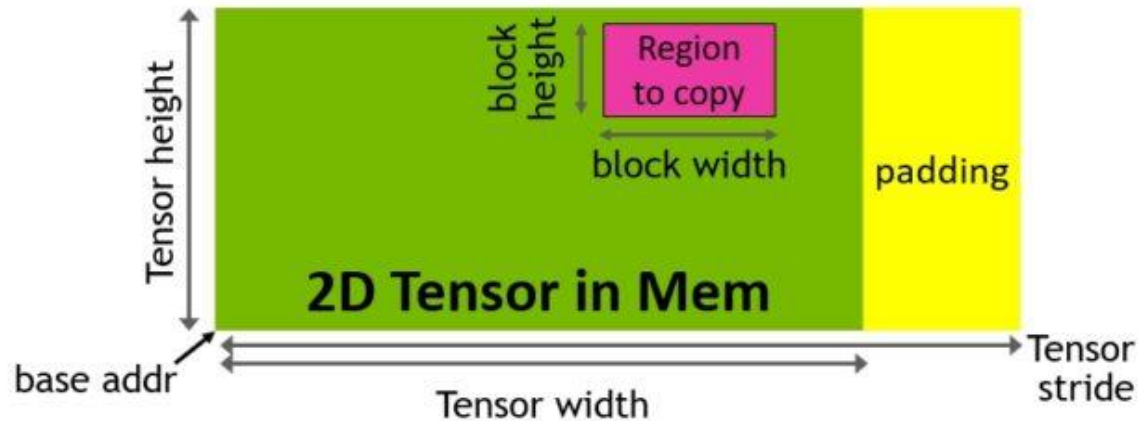
kind: Type of transfer, cudaMemcpyHostToDevice or
cudaMemcpyDeviceToHost

```
cudaMemcpy(dGPU, hCpu, n * sizeof(int), cudaMemcpyHostToDevice);
```

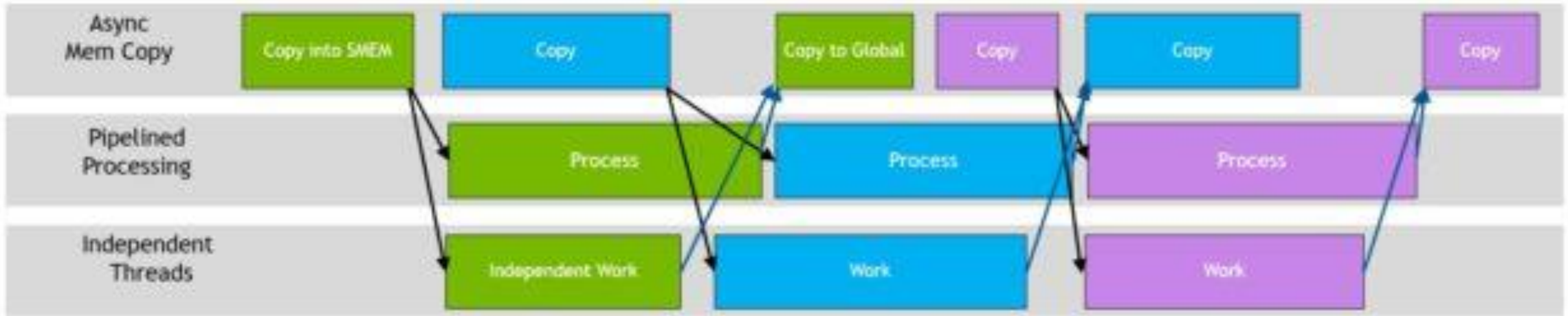
New in H100: Tensor Memory Accelerator (TMA)

```
void cuda::memcpy_async(void* destination, void const* source,  
Shape size, cuda::barrier<Scope, CompletionFunction>& barrier);
```

```
barrier.wait()
```



Asynchronous execution in Hopper GPU



CUDA Programming Model Exposure	A100	New for H100
Barrier.arrive(),	Asynchronous Barrier	Asynchronous Transaction Barrier
Barrier.wait()	Waiter spins in SMEM	Waiter sleeps until all threads arrive
Memcpy_async()	Direct copy to SMEM	Asynchronous mem copy unit (called TMA)

Declaration of Host/Device function

- Both host and device code in same `.cu` file
- Indicate where the code will run

keyword	call on	execute on
<code>__global__</code>	host (cpu)	device (gpu)
<code>__device__</code>	device (gpu)	device (gpu)
<code>__host__</code>	host	host

Defining Functions to be executed on GPU

Define kernel function, `__global__`

```
__global__ void VecAddKernel(int* A, int* B, int* C, int n) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if (i < n) {  
        C[i] = A[i] + B[i];  
    }  
}
```

```
int main() {  
    VecAddKernel<<<1, N>>>(A, B, C, N);  
}
```

Calling Kernel at Runtime

- Host program specifies grid-block-threads configurations for kernel at run time
 - Dg and Db are either dim3 or int

```
dim3 Dg(4, 2, 1);  
dim3 Db(8, 8, 1);  
kernelFuncName<<<Dg, Db>>>(args)
```
- Dg: size of grid (num. of blocks)
 - $Dg.x * Dg.y * Dg.z$ is num. of blocks
- Db: size of block
 - $Db.x * Db.y * Db.z$ is num. of threads per block, ≤ 1024)

Device Runtime Variables

- Host launches kernels on a gpu device
- Each kernel thread needs to know which thread it is running
- Compiler generates build-in variables, with x, y, z fields

gridDim	dim3	dimensions of grid
blockIdx	uint3	index of block within grid
blockDim	dim3	dimensions of block
threadIdx	uint3	index of thread within block

Calling CUDA Kernel from CPU

Running kernels on GPU

```
// n: the size of the vector  
int n = 1024;  
int threads_per_block = 256;  
int num_blocks = (n + threads_per_block - 1) /  
    threads_per_block;  
VecAddKernel<<<num_blocks, threads_per_block>>>(dA, dB, dC, n);
```

CUDA Code Examples for Matrix Computation

Matrix Multiplication with CUDA

- See notebook example.
- https://github.com/lmsystem/lmsys_code_examples/blob/main/simple_cuda_demo/CUDA_Code_Examples.ipynb
- You may upload and run it in Google Colab.

```
__global__ void MatAddKernel(float* A, float* B, float* C, int N) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int j = blockIdx.y * blockDim.y + threadIdx.y;  
    C[i * N + j] = A[i * N + j] + B[i * N + j];  
}
```

```
int main() {  
    int N = 32;  
    dim3 threads_per_block(N, N);  
    int num_blocks = 1;  
    MatAddKernel<<<num_blocks, threads_per_block>>>(dA, dB, dC, N);  
}
```

```
int main() {  
    dim3 threads_per_block(2, 4, 8);  
    dim3 blocks_per_grid(2, 3, 4);  
    fullKernel<<<blocks_per_grid, threads_per_block>>>(some_input,  
    some_output);  
}
```

24 blocks per grid

64 threads per block

1536 threads in total

can you run this simultaneously on A6000?

```

__global__ void fullKernel(float* din, float* dout) {
    int block_id = blockIdx.x + blockIdx.y * gridDim.x      + blockIdx.z *
gridDim.x * gridDim.y;
    int block_offset = block_id * blockDim.x *      blockDim.y * blockDim.z;
    int thread_offset = threadIdx.x
        + threadIdx.y * blockDim.x
        + threadIdx.z * blockDim.x * blockDim.y;
    int tid = block_offset + thread_offset;
    dout[tid] = func(din[tid]);
}

```

Vector Addition

```
void VecAddCUDA(int* Acpu, int* Bcpu, int* Ccpu, int n) {  
    int *dA, *dB, *dC;  
    cudaMalloc(&dA, n * sizeof(int));  
    cudaMalloc(&dB, n * sizeof(int));  
    cudaMalloc(&dC, n * sizeof(int));  
    cudaMemcpy(dA, Acpu, n * sizeof(int), cudaMemcpyHostToDevice);  
    cudaMemcpy(dB, Bcpu, n * sizeof(int), cudaMemcpyHostToDevice);  
    int threads_per_block = 256;  
    int num_blocks = (n + threads_per_block - 1) / threads_per_block;  
    VecAddKernel<<<num_blocks, threads_per_block>>>(dA, dB, dC, n);  
    cudaMemcpy(Ccpu, dC, n * sizeof(int), cudaMemcpyDeviceToHost);  
    cudaFree(dA);  
    cudaFree(dB);  
    cudaFree(dC);  
}
```

Matrix Addition

```
void MatAddCUDA(int* Acpu, int* Bcpu, int* Ccpu, int n) {  
    int *dA, *dB, *dC;  
    cudaMalloc(&dA, n * n * sizeof(int));  
    cudaMalloc(&dB, n * n * sizeof(int));  
    cudaMalloc(&dC, n * n * sizeof(int));  
    cudaMemcpy(dA, Acpu, n * n * sizeof(int), cudaMemcpyHostToDevice);  
    cudaMemcpy(dB, Bcpu, n * n * sizeof(int), cudaMemcpyHostToDevice);  
    int THREADS = 32;  
    int BLOCKS = (n + THREADS - 1) / THREADS;  
    dim3 threads(THREADS, THREADS); // should be <= 1024  
    dim3 blocks(BLOCKS, BLOCKS);  
    MatAddKernel<<<blocks, threads>>>(dA, dB, dC, n);  
    cudaMemcpy(Ccpu, dC, n * n * sizeof(int), cudaMemcpyDeviceToHost);  
    cudaFree(dA);  
    cudaFree(dB);  
    cudaFree(dC);  
}
```

Summary

- Basic GPU CUDA operations
 - memory allocation
 - data movement
 - creating threads and running on SMs
 - specifying number of threads and number of blocks in a grid
 - referring to data in GPU memory within a thread
 - using building index variables to refer to the data

Recommended Reading

- Chap 2,3,4 of "Programming Massively Parallel Processors, 4th Ed.
https://learning.oreilly.com/library/view/programming-massively-parallel/9780323984638/?sso_link=yes&sso_link_from=cmu-edu
- Free for CMU students