Introduction to Parallel Processing

Lecture 22 : Introduction to GPUs : MPI + CUDA

Professor Amanda Bienz

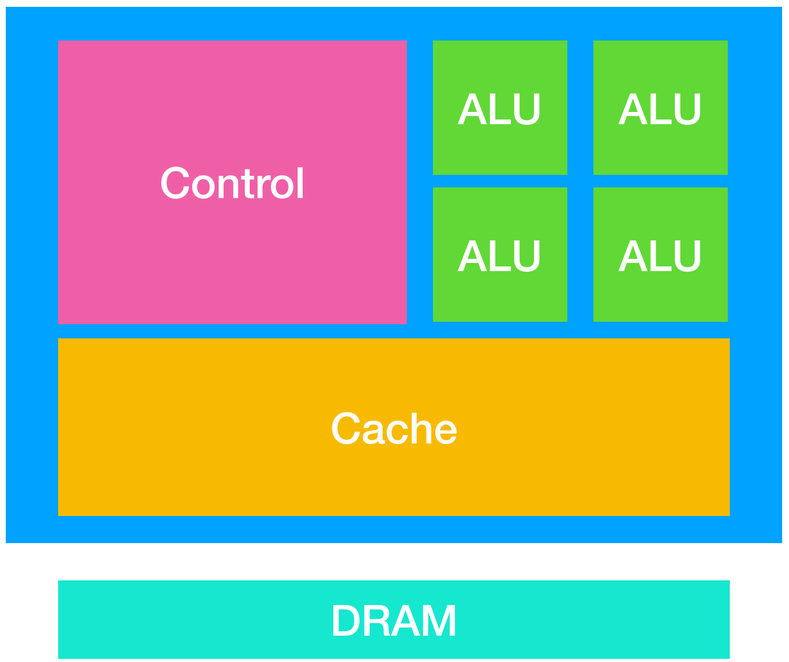
Programming Massively Parallel Processors by Wen-mei Hwu, David Kirk, Izzat Hajj

What are GPUs

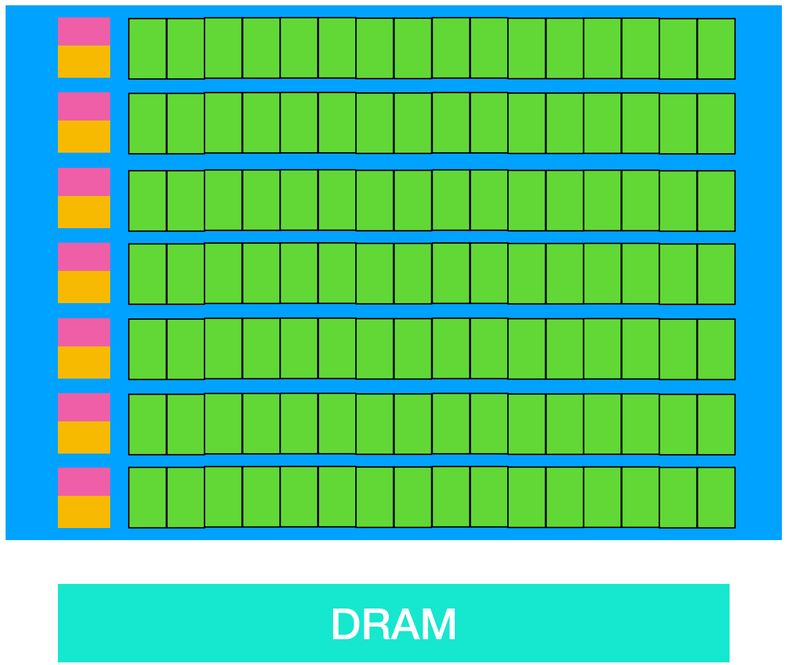
- Graphical processing units : hardware for providing graphical display on computer
- Thousands of pixels by thousands of pixels
- All pixels are updated at once (to change the image)
- Think video games… pixels are constantly being updated
- Idea : use this hardware to perform thousands of additions at once

CPU vs GPU

- CPUs and GPUs have fundamentally different designs



CPU



GPU

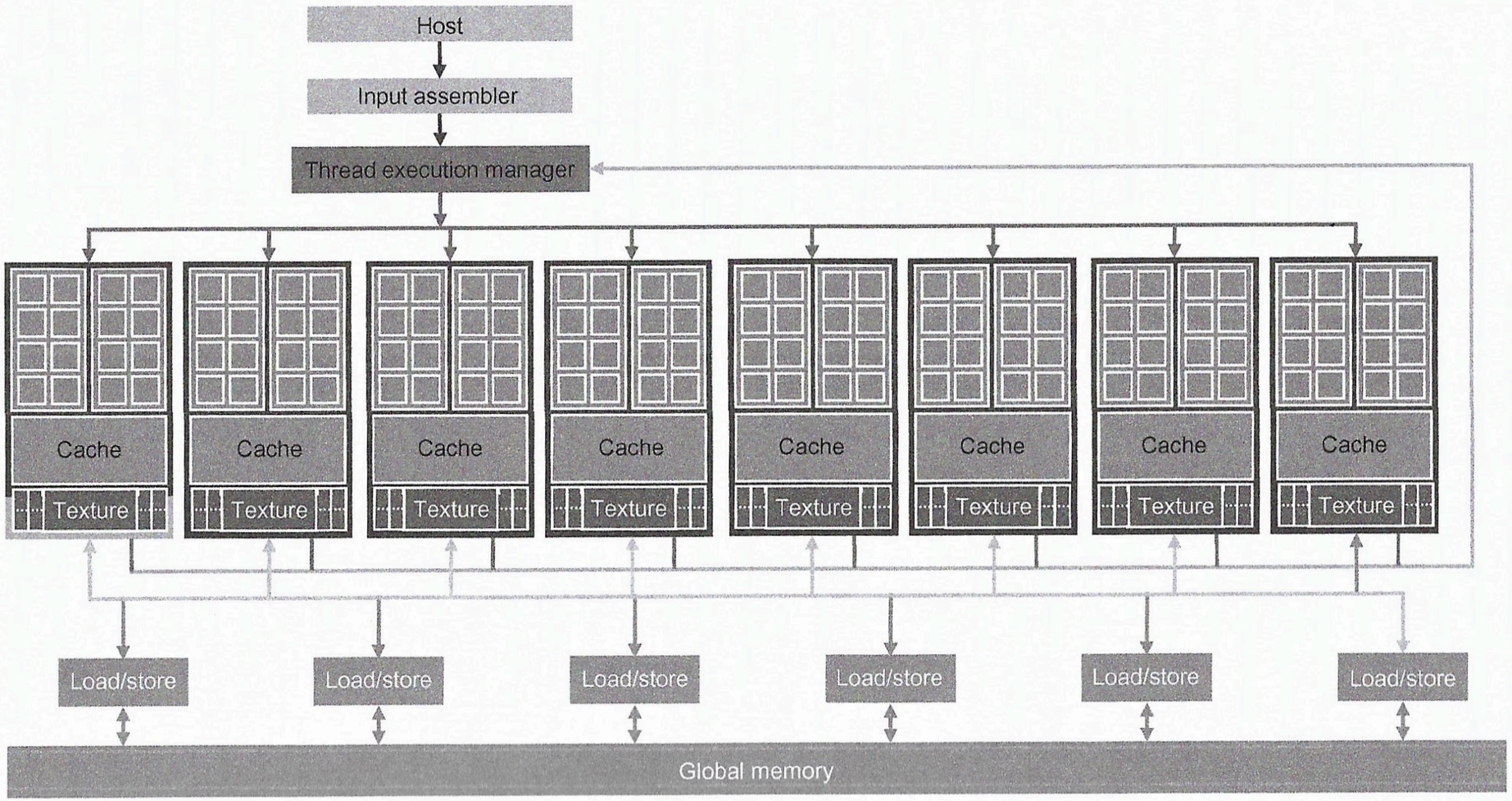


FIGURE 1.2

Architecture of a CUDA-capable GPU.

Data Parallel : GPUs are GREAT at this!

How CUDA Programs work

CPU Serial Code

Device Parallel Kernel
KernelA<<<nBlk,NTid>>>(args);

CPU Serial Code

Device Parallel Kernel
KernelB<<<nBlk,NTid>>>(args);

Hello World in CUDA

- Let’s step through a simple hello world program in CUDA
  - Have each thread on the GPU print “Hello World”
- Filename will end in .cu
- Compile with nvcc
- **On Xena, you must include the following architecture:**
  - nvcc -arch=sm_35 -o filename filename.cu

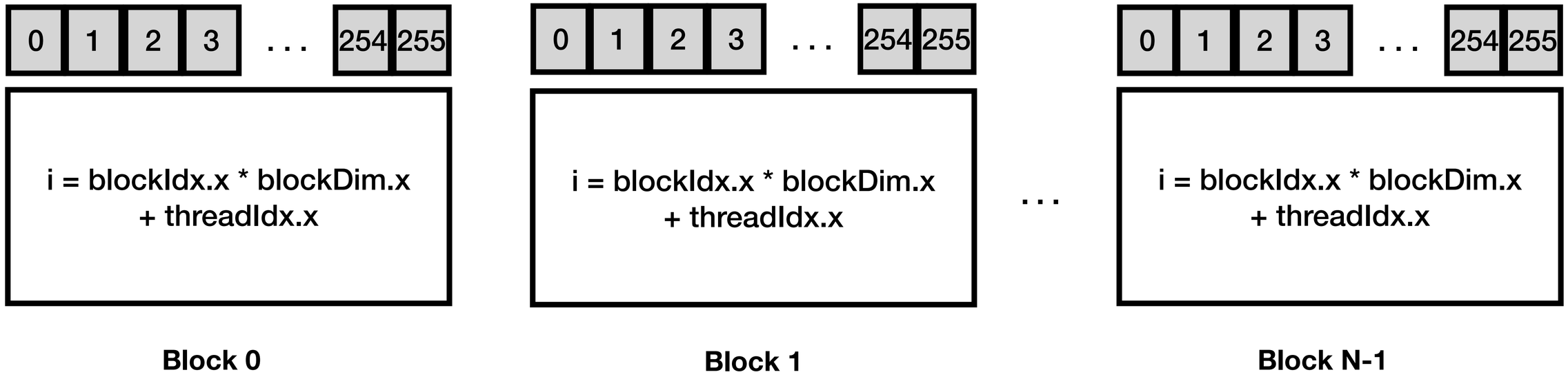
CUDA Function Declaration Keywords

- __device__ : executed on device, only callable from device
- __global__ : executed on device, only callable from host
- __host__ : executed on host, only callable from host

- Note : you can use both __host__ and __device__ in a function declaration… compiler generates two versions of object files for same function

Kernel Functions / Threading

- When launching kernel, threads are organized in two-level hierarchy
  - Each grid is organized in array of thread blocks (we will refer to these as **blocks**)
  - All blocks of grid are same size : each block can contain up to 1024 threads

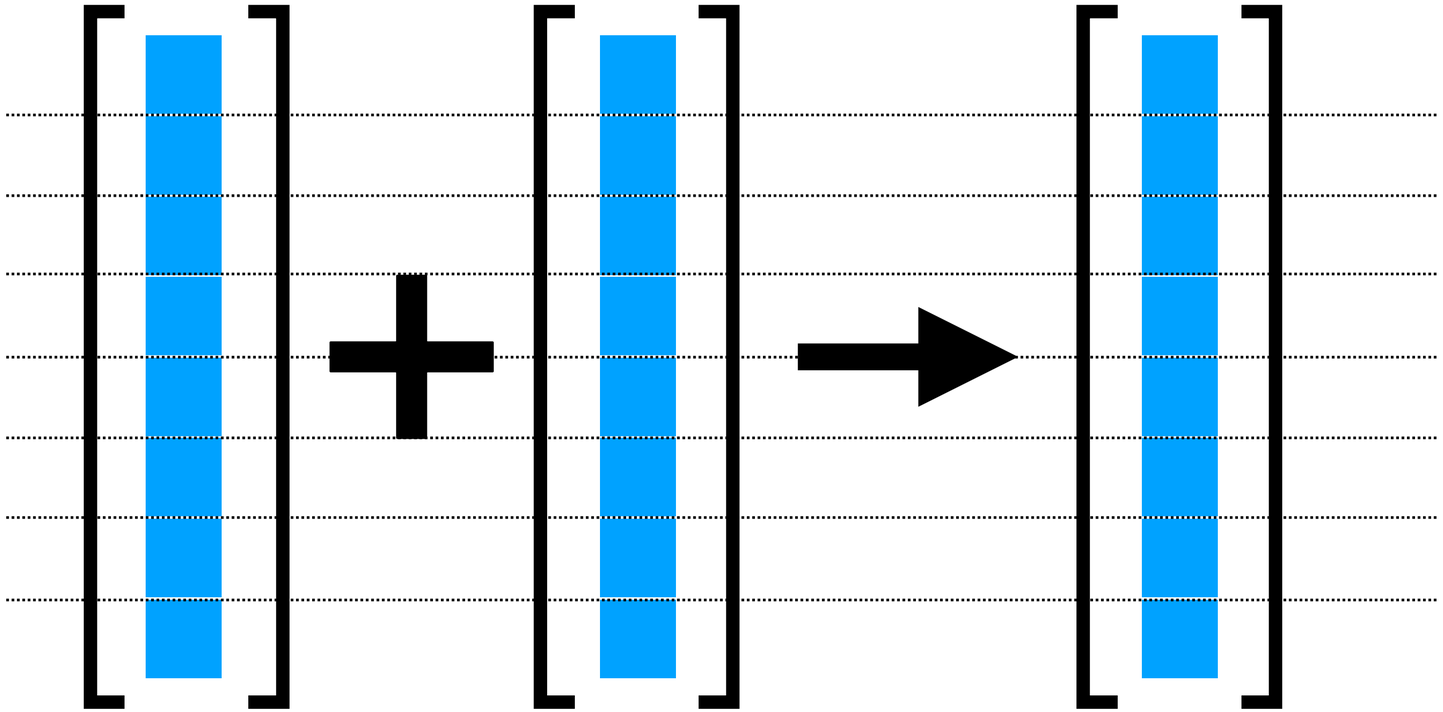


Hello World in CUDA

- Let’s update our Hello World program to include the following
  - Each thread’s block ID
  - The block dimension
  - Each thread’s thread index within the given block

Typically, Need Data on GPU

- Example : vector addition
  - All three vectors must be allocated **in GPU memory**



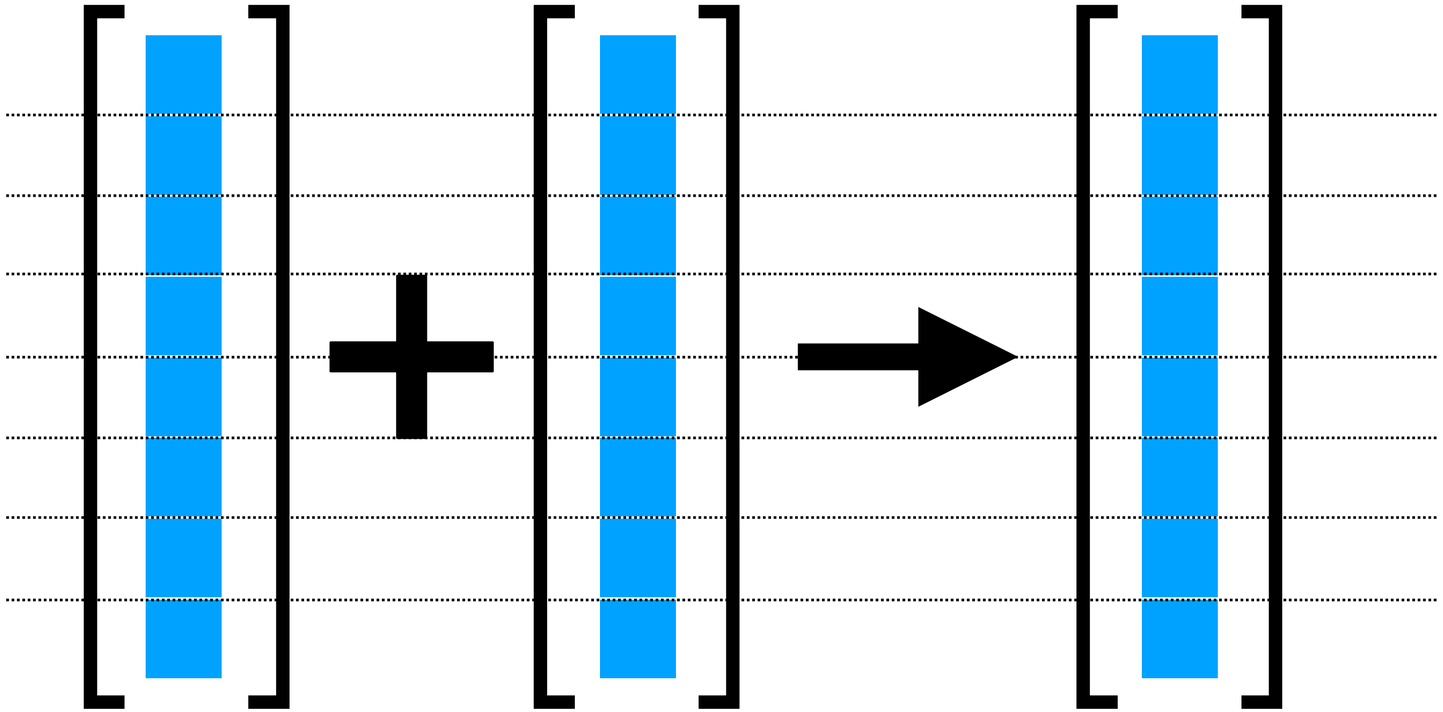
Allocating Device Memory

- cudaMalloc(void** buffer, int size)
  - allocates ‘size’ bytes in array ‘buffer’ on the device (GPU)

- cudaFree(void** buffer)
  - frees the memory from the GPU

Device Memory

- Cannot touch device memory from the CPU
- How do we initialize the values on the GPU? How do we read the solution from the CPU?



Copying between CPU and GPU

- cudaMemcpy(void* dest_buffer, void* src_buffer, size_t count,
  cudaMemcpyKind kind)
- Can transfer from host to device, device to host, or between devices
- Transfers from src_buffer to dest_buffer
- Number of bytes = count
- cudaMemcpyKind tells direction of transfer :
  - cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice

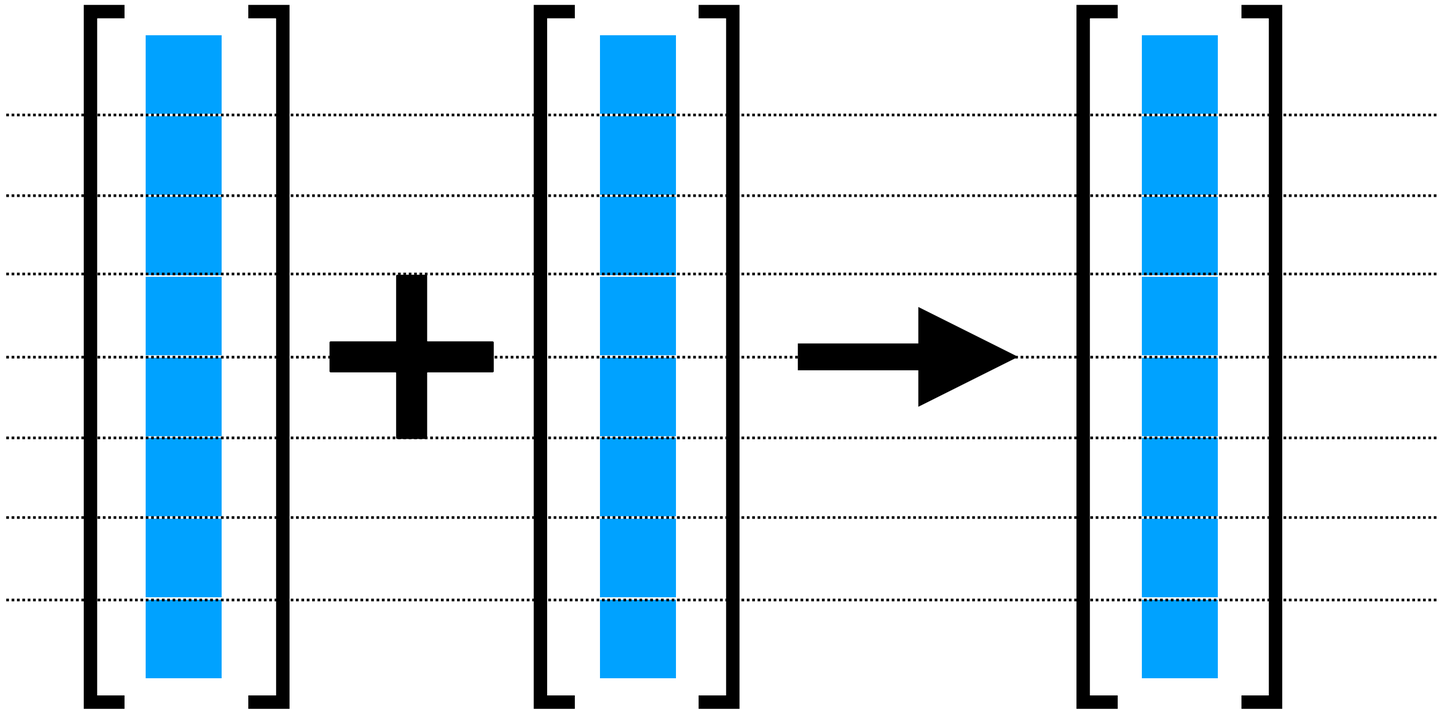
Allocating Host Memory

- malloc : allocates ***pageable*** memory… slow when transferring to GPU!

- cudaMallocHost(void** buffer, int size)
  - allocates ‘size’ bytes in ***pinned*** array ‘buffer’ on the host (CPU)
- cudaFreeHost(void** buffer)
  - frees the ***pinned*** memory from the CPU

Vector Addition

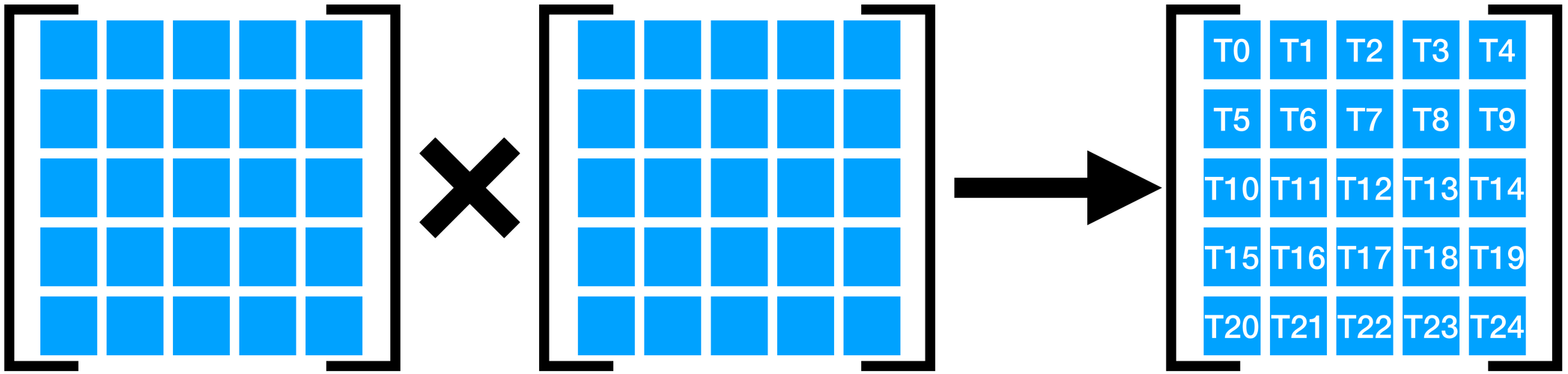
- Call vector additional with a number of 1D thread blocks
  - **Need each thread on the GPU to check that they have corresponding vector elements!**



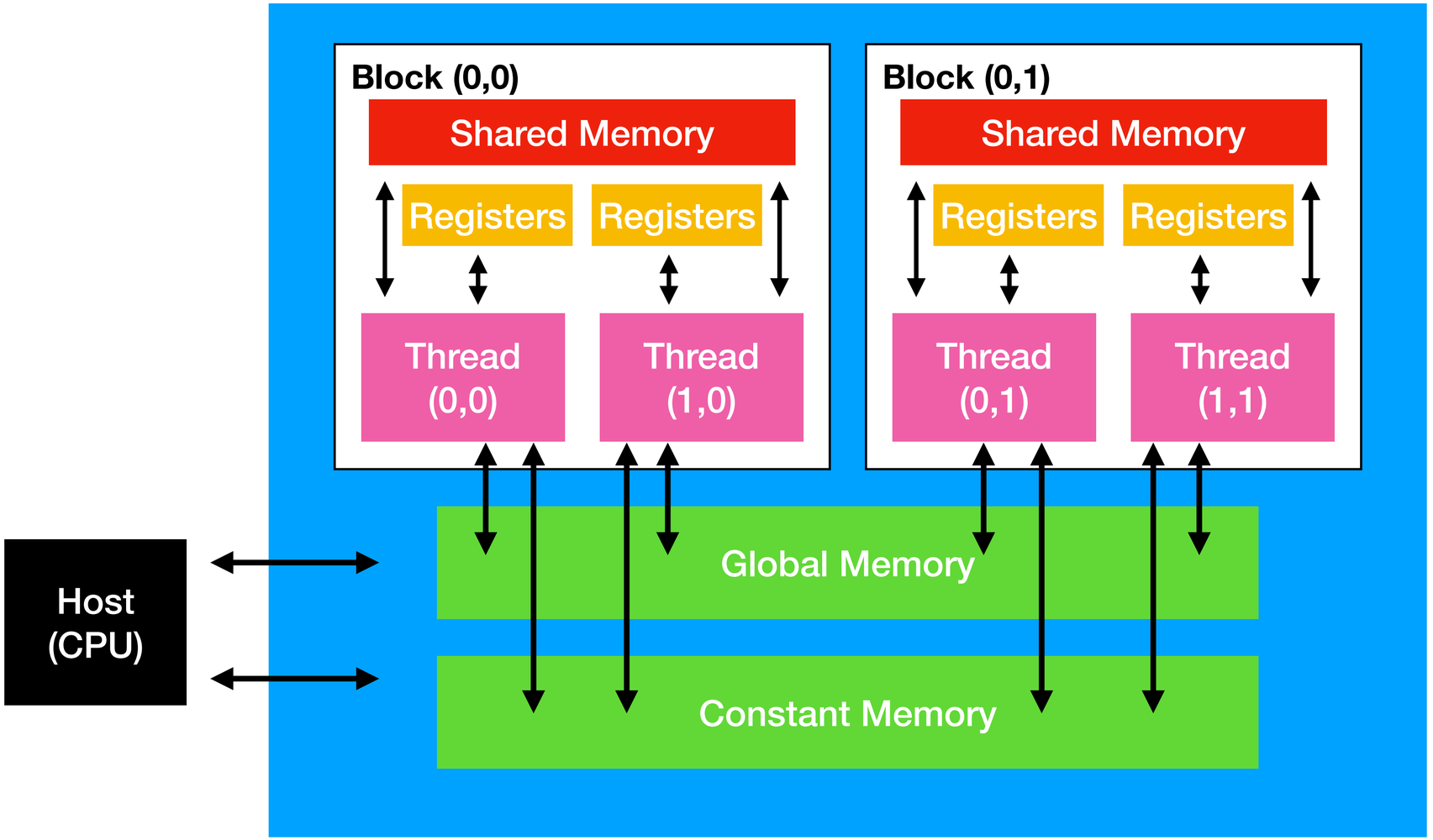
2 Dimensional Thread Layout

- What about matrix-matrix multiplication?

- Don’t want multiple threads writing to same element (shared memory) so why don’t we have each thread calculate one element of result matrix

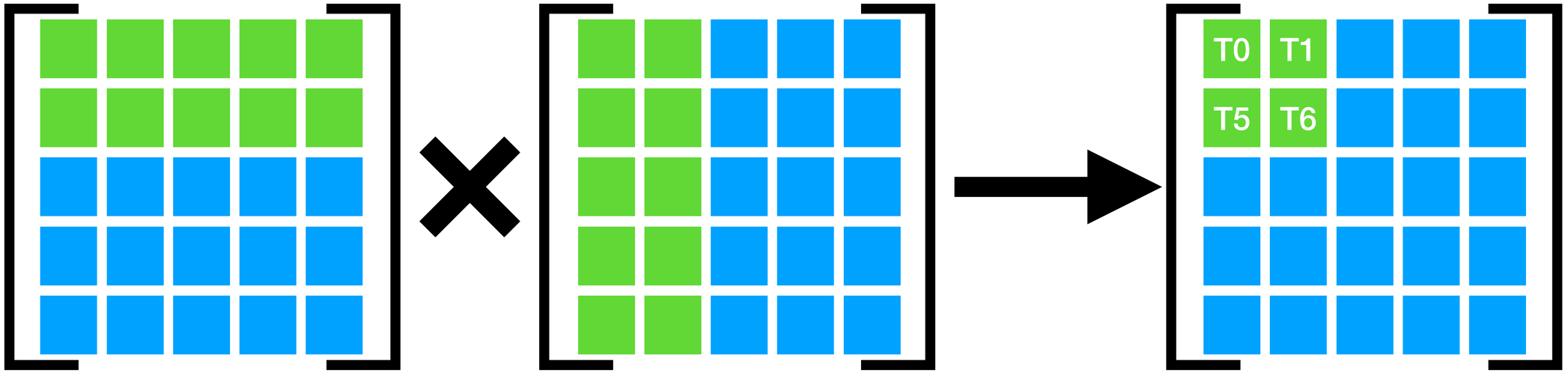


GPU Memory


Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Block (0,1)
Shared Memory
Registers
Registers
Thread (0,1)
Thread (1,1)
Global Memory
Constant Memory
Host (CPU)

Tiling

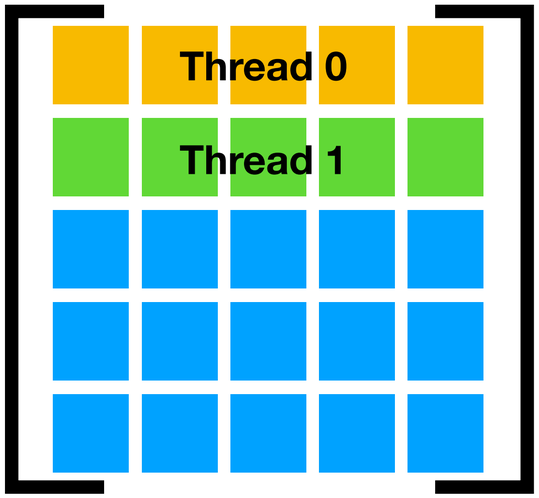
- Global memory is large, but slow
- Shared memory is small, but fast
- Tiling : partition the data into subsets, called tiles, so that each tile fits into shared memory



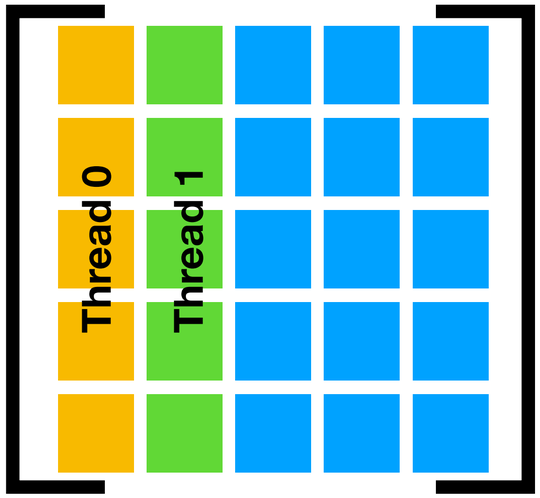
Coalesced Memory Accesses

- Hardware combines consecutive DRAM locations into a single consolidated memory access

- So, if thread 0 accesses global memory location N, thread 1 accesses N+1, thread 2 accesses N+2, and so on, the accesses can be combined into a single request



Not coalesced



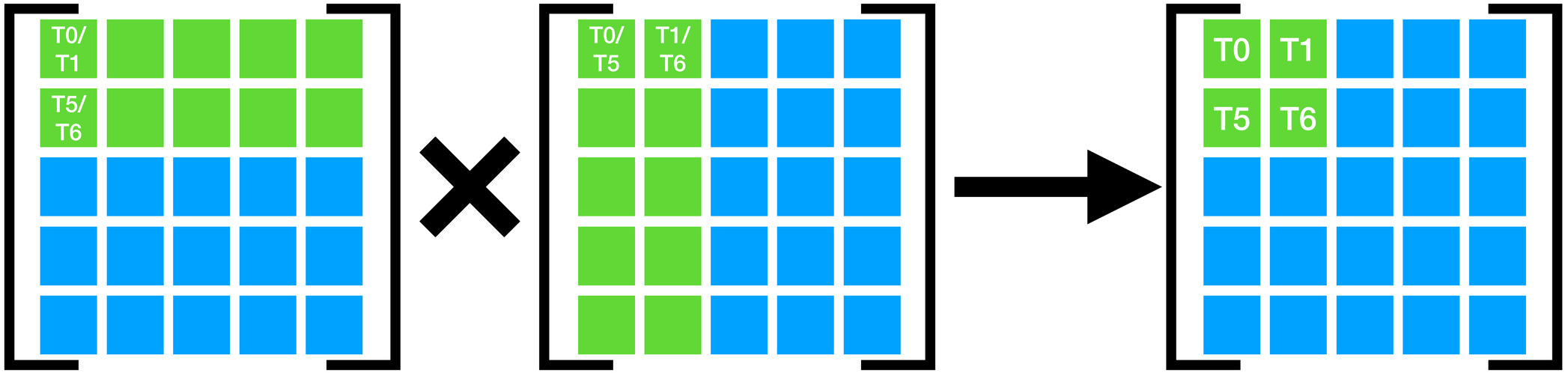
Coalesced

Warps

- Each thread block is partitioned into warps (often a warp has 32 threads)
- SIMD hardware executes all threads in a warp as a bundle… one instruction is run for all threads in the same warp
- If all threads in a warp follow same execution path, this hardware works well, but if they take different control paths, SIMD hardware will take multiple passes through the divergent paths
- Example : If/Else statements… want all threads in a wrap to execute the same clause (not some execute if while others execute else)

Thread Granularity

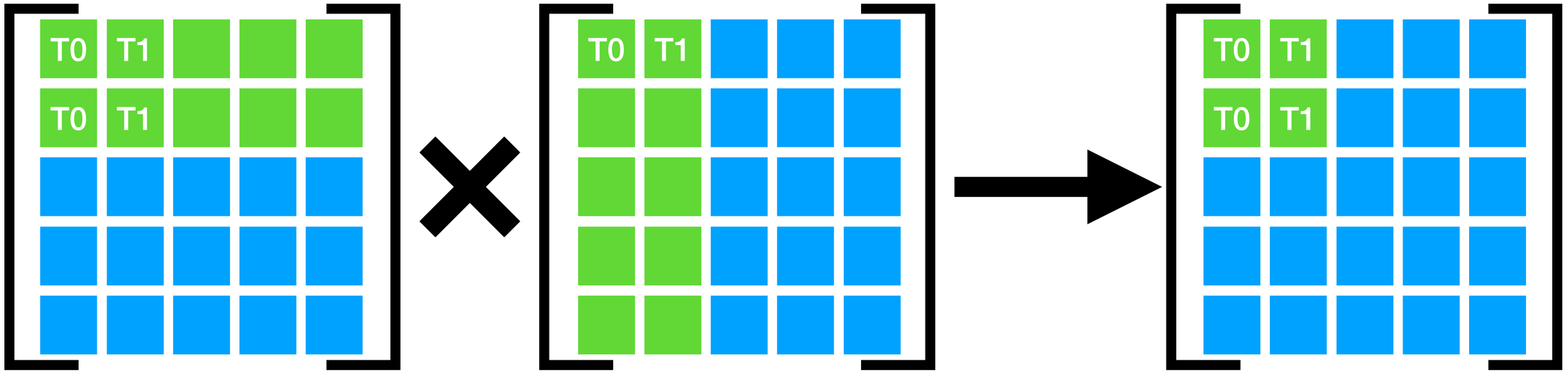
- Often, want one thread to calculate more then a single element of resulting matrix because redundant work performed by multiple threads



Thread Granularity

- Trade-off … now more serial work per thread, need to use more registers, more shared memory

- “In practice, combining up to four adjacent horizontal blocks to compute adjacent horizontal tiles significantly improves performance of large” matrix multiplies



Too complicated for a week

- There is too much to learn to expect you could write performant CUDA code after a week of class
- Really, need to spend a semester on this
- However, there are some great alternatives
  - Existing codebases (including Nvidia codebases)
  - Code generation (Loopy)
  - OpenMP offloading (we will talk about this later this semester)