Intro to CUDA Programming

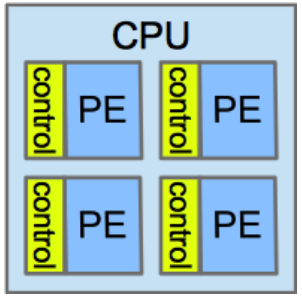
Operating Systems

CUDA

- Framework developed by NVIDIA
- For General Purpose GPU (GPGPU) programming
  - Factors out hardware differences across GPU models
  - We can mix CPU and GPU code in the same source file
  - Some tasks run on the CPU
  - With help from the programmer, compute-intensive tasks run on the GPU

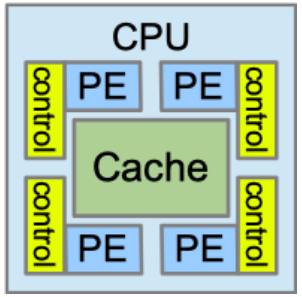
CPU vs GPU

- The CPU and GPU reflect different design philosophies
- CPU
  - few processing elements
  - each independently controllable

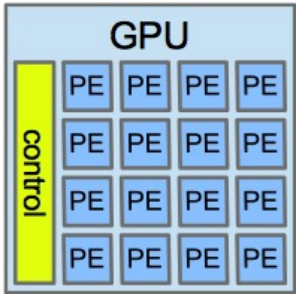


CPU vs GPU

- The CPU and GPU reflect different design philosophies
- CPU
  - few processing elements
  - each independently controllable
  - with lots of cache to make them fast

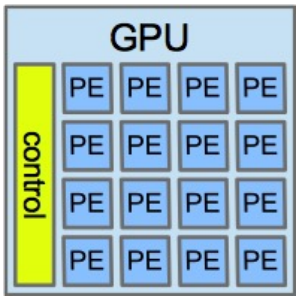


CPU vs GPU

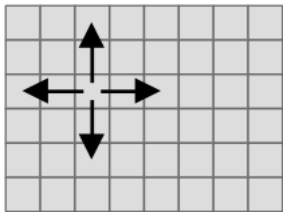


- GPU
  - lots and lots processors
  - Each less powerful than a modern CPU
  - All doing pretty much the same thing
    - SIMD model, each running the same instruction of the same program at the same time

CPU vs GPU

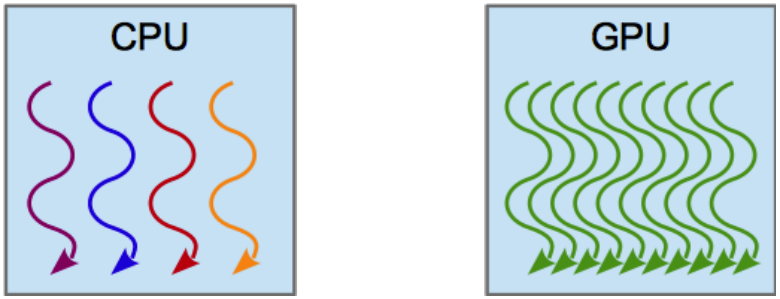


- GPU
  - lots and lots processors
  - Each less powerful than a modern CPU
  - All doing pretty much the same thing
    - SIMD model, each running the same instruction of the same program at the same time
  - Some cache, but …
    - With different notions of locality for typical GPU tasks
    - Much more real estate dedicated to processing than to cache



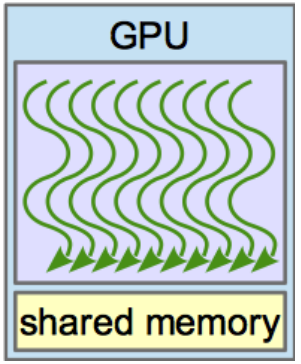
CPU vs GPU

- While the CPU is good for running comparatively few threads, each doing its own thing
- The GPU is good for running thousands of threads, each doing the same thing.



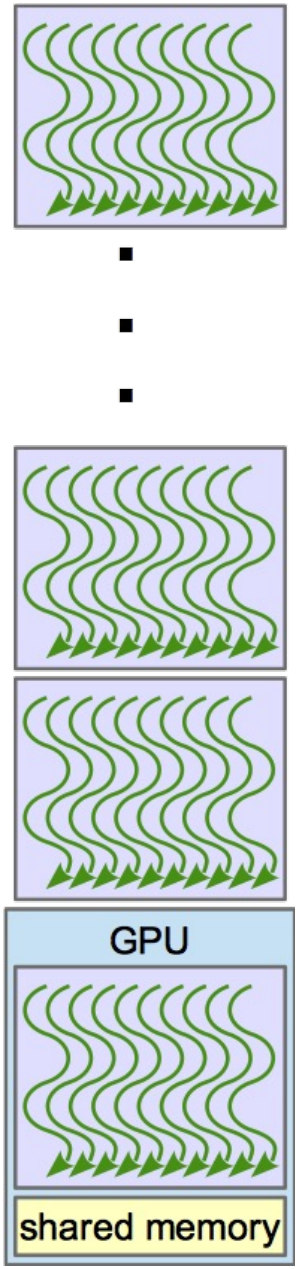
Running Jobs on the GPU

- In fact, that's (part of) how GPU jobs are organized
- You create a *block* of threads
  - With a block containing tens to hundreds of threads
  - Fast, *shared memory* lets threads in the block communicate & cooperate
  - Synchronization support (e.g., wait until all threads in the block are up to here)



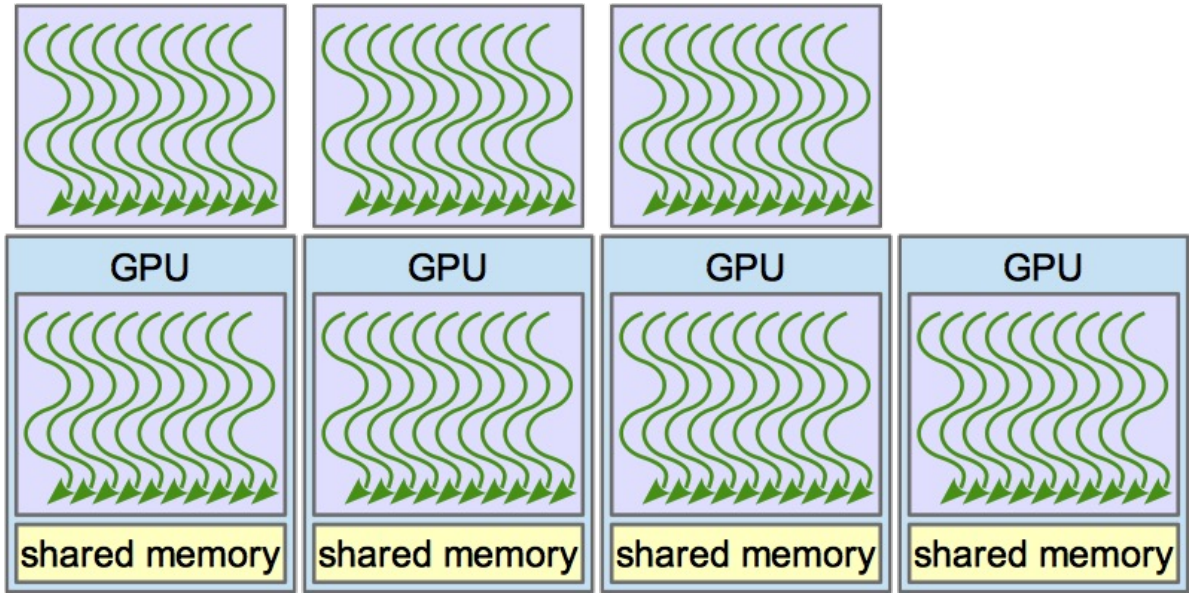
Larger GPU Jobs

- Shared memory and synchronization support limit the feasible block size.
- So, compute jobs are organized as multiple blocks.
  - Each block processed independently by the GPU
  - Threads within a block can share memory and synchronize.
  - Threads in two different blocks can't.



Larger GPU Jobs

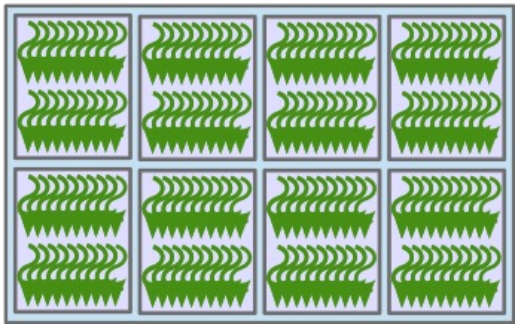
- With multiple or more powerful GPUs, blocks can be processed in parallel for improved performance



Organizing GPU Jobs

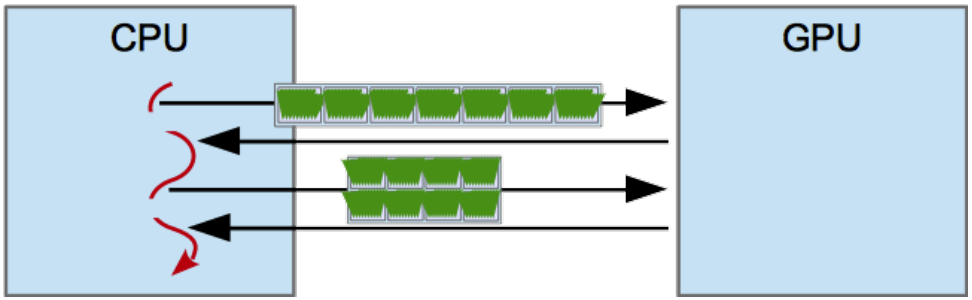
- So, a GPU job is really a collection of blocks (of threads)
- That's called a *grid*

  - Each thread in a block has an index
  - Each block in a grid has an index
  - Each thread uses its index to decide what computations it's responsible for
  - Threads get these automatically (you don't have to pass them in)
- Really, blocks and grids can be multi-dimensional

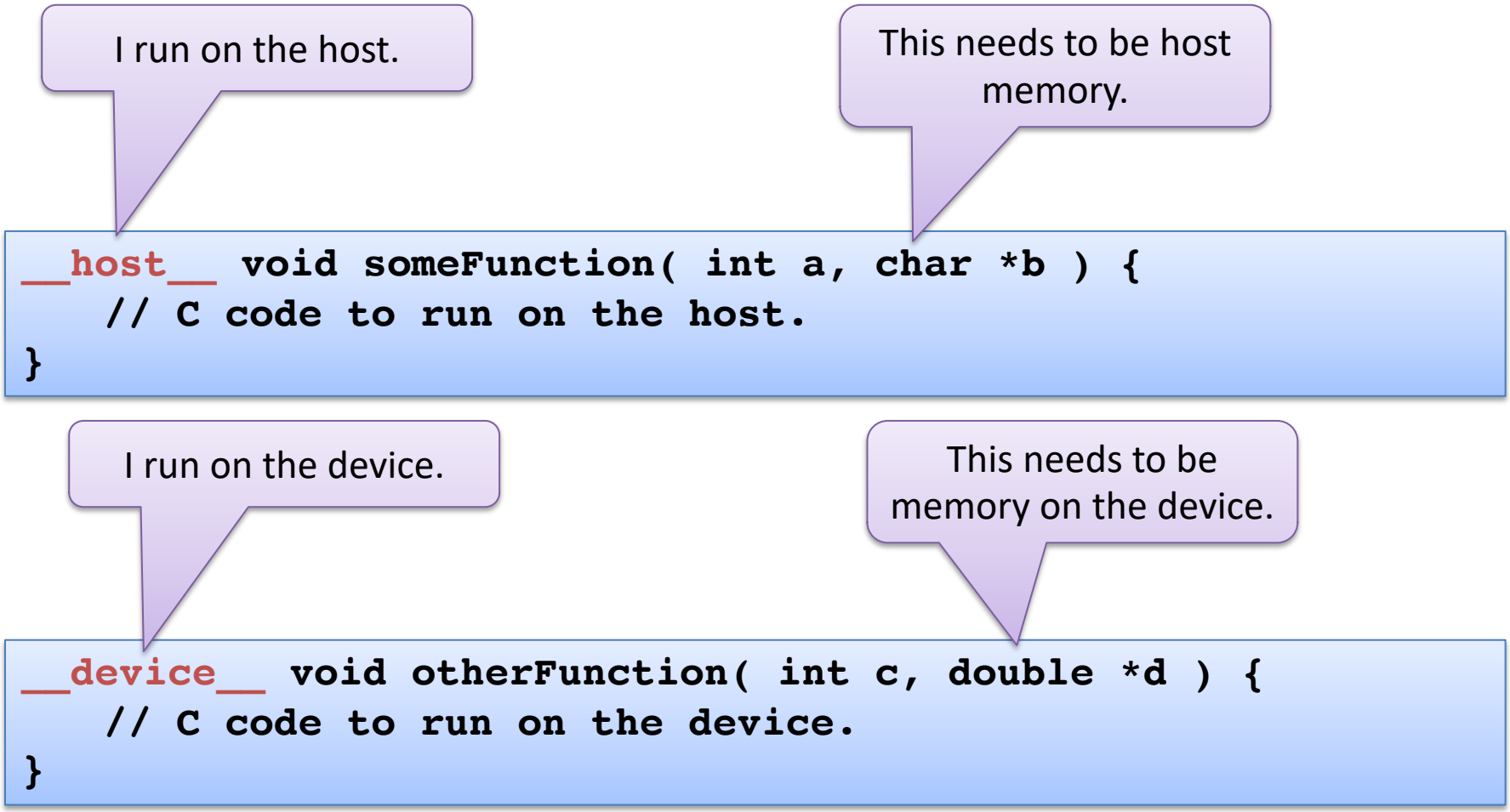


CPU/GPU Cooperation

- The CPU is in charge
  - It executes serial parts of the program (or very limited parallelism)
  - It prepares compute tasks for the GPU
  - It requests and waits for grids to execute on the GPU
- System on the left: the *host*
- System on the right: the *device*



Choosing Your Processor



A CUDA Kernel

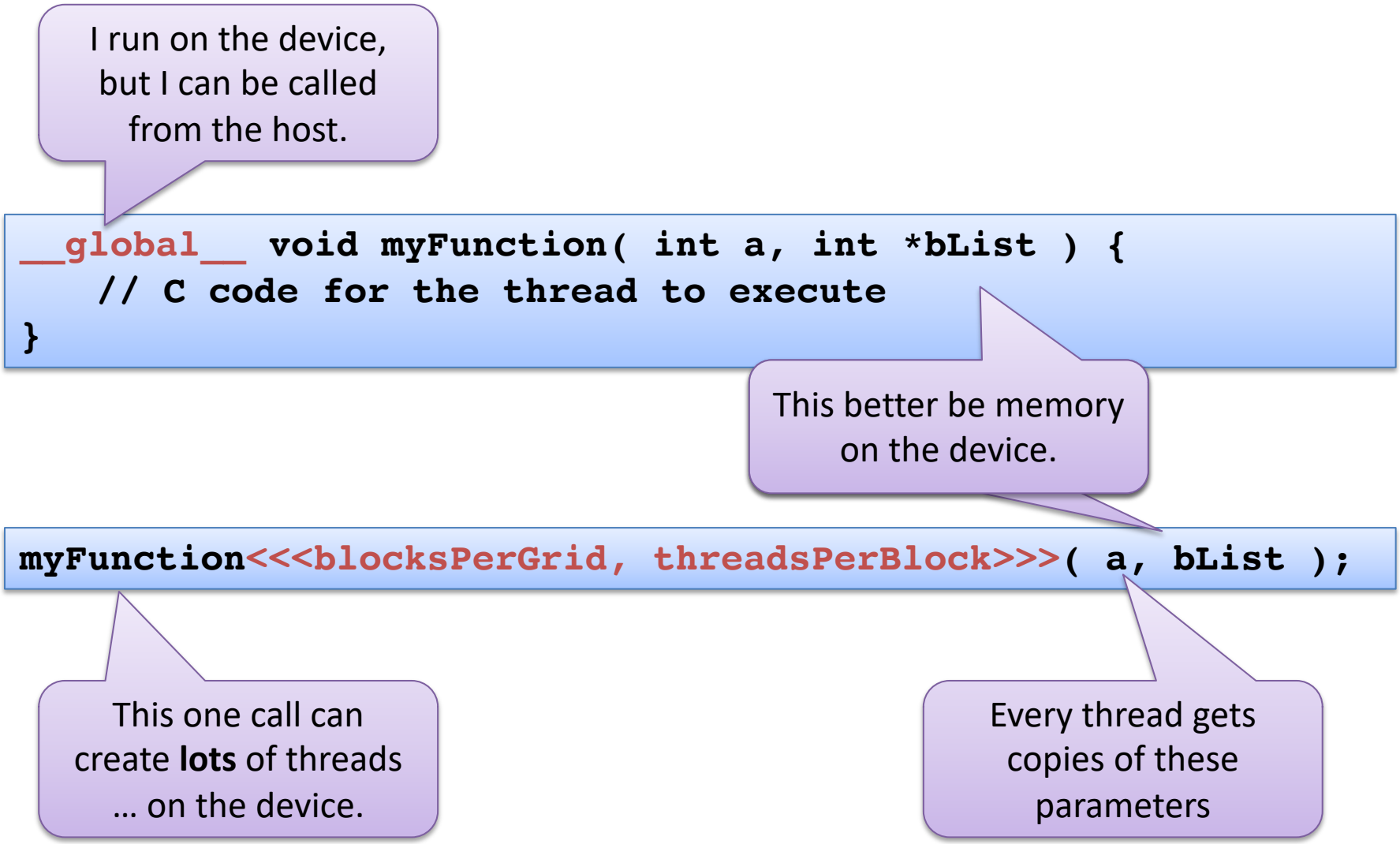
- Each thread in a grid executes a *kernel*
  - Just a name for a block of code to be run in parallel; no relation to the OS kernel
  - Extra language features to mark a kernel

```
__global__ void myFunction( int a, int *bList ) {
    // C code for the thread to execute
}
```

  - Language features to choose grid dimensions and run a kernel on the GPU

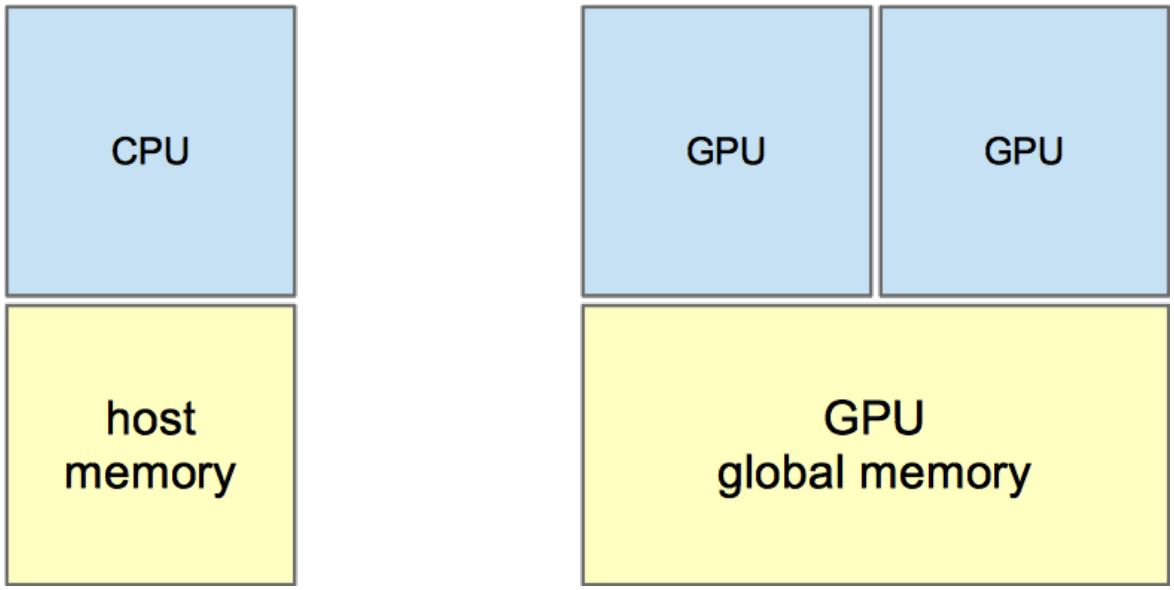
```
myFunction<<<blocksPerGrid, threadsPerBlock>>>( a, bList );
```

Things to Notice



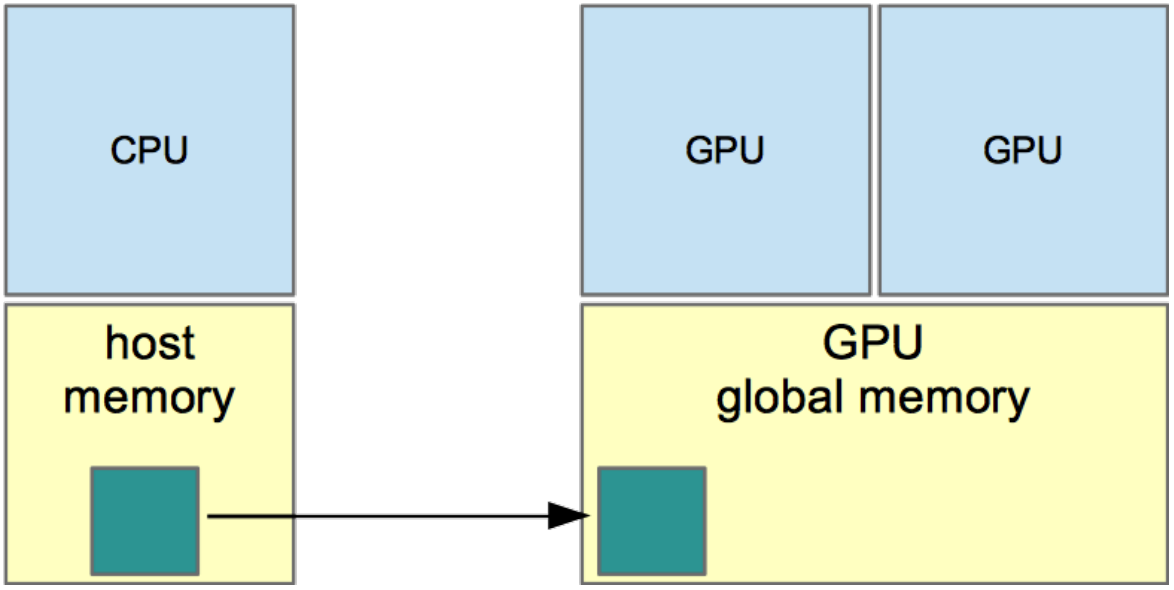
CPU vs GPU Memory

- The GPU device has its own *global memory*, accessible to all GPUs on the device
- But, not directly accessible to the host.



Preparing Device Input

- Normally, the host will prepare data
- Then, copy it to the device (really, it will ask the device to copy from host memory to its global memory)



Preparing Device Input

- First, we may need to allocate and get a pointer to memory on the device.

This will give us a pointer … one the host shouldn't dereference.

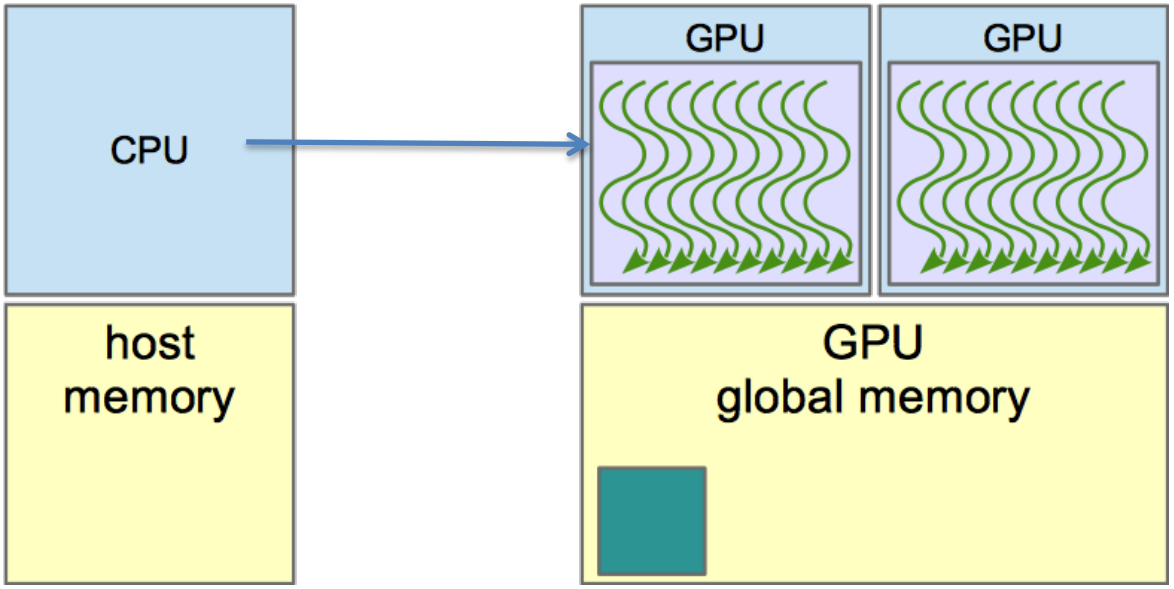
```
int *devList = NULL;
cudaMalloc( (void **)&devList, count * sizeof(int) );
```

- Then, we can ask the device to copy into this memory

```
cudaMemcpy( devList, myList, count * sizeof( int ),
            cudaMemcpyHostToDevice )
```

Running a Kernel

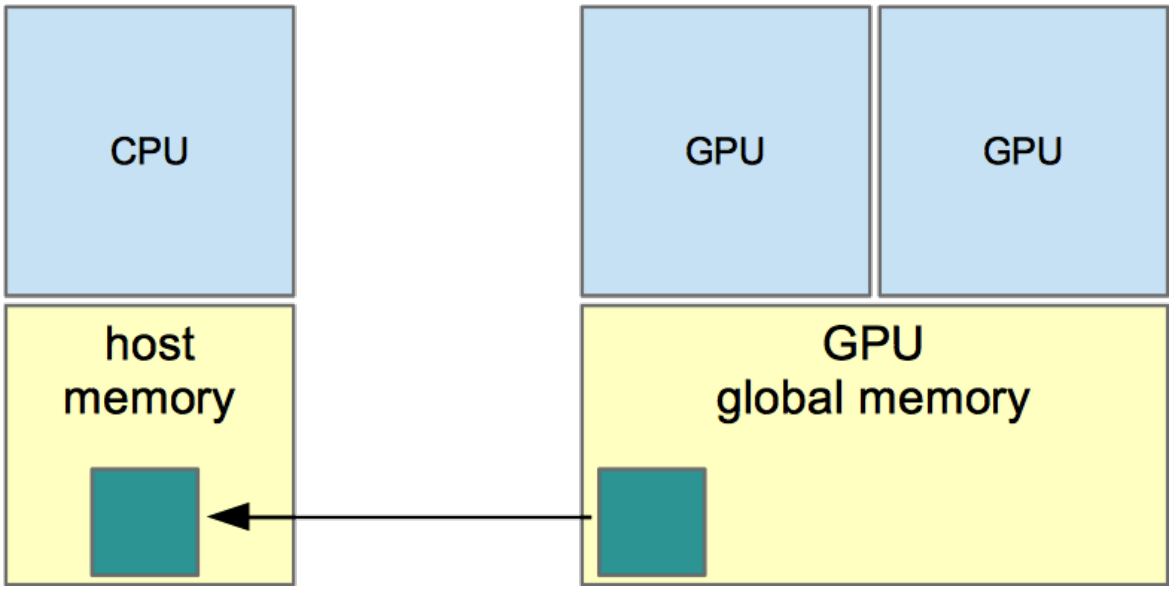
- Then, we ask the GPU to run a kernel (or maybe several, one after another)



Recovering Device Output

- Then, we copy results back to host memory

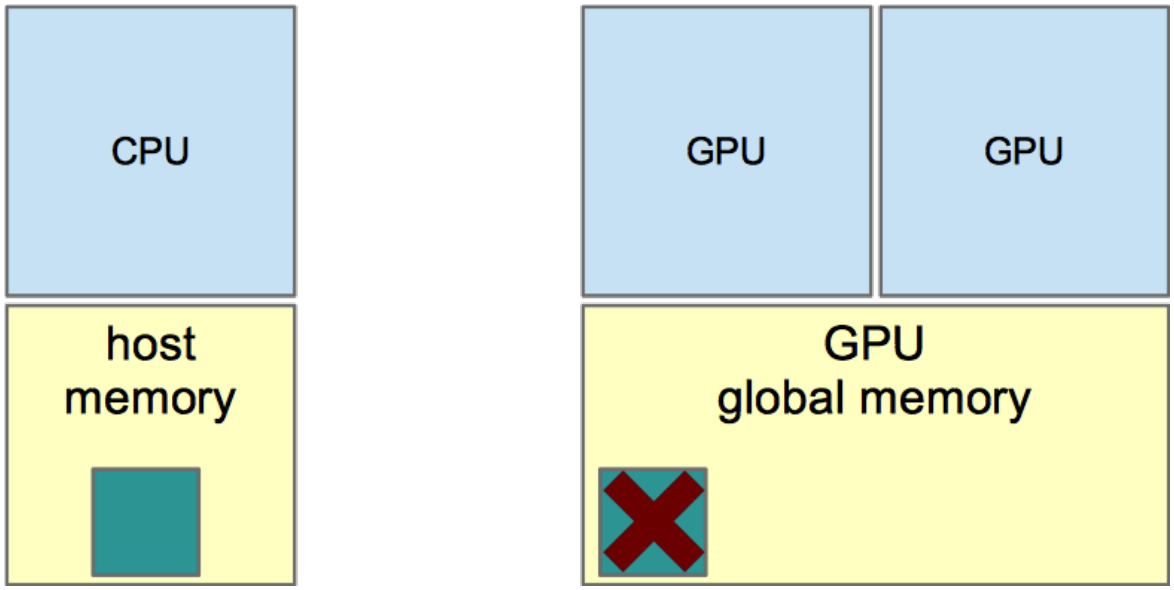
```
cudaMemcpy( myResults, devResults, length * sizeof(int),
            cudaMemcpyDeviceToHost )
```



Cleaning Up

- When we no longer need them, we can free device resources.

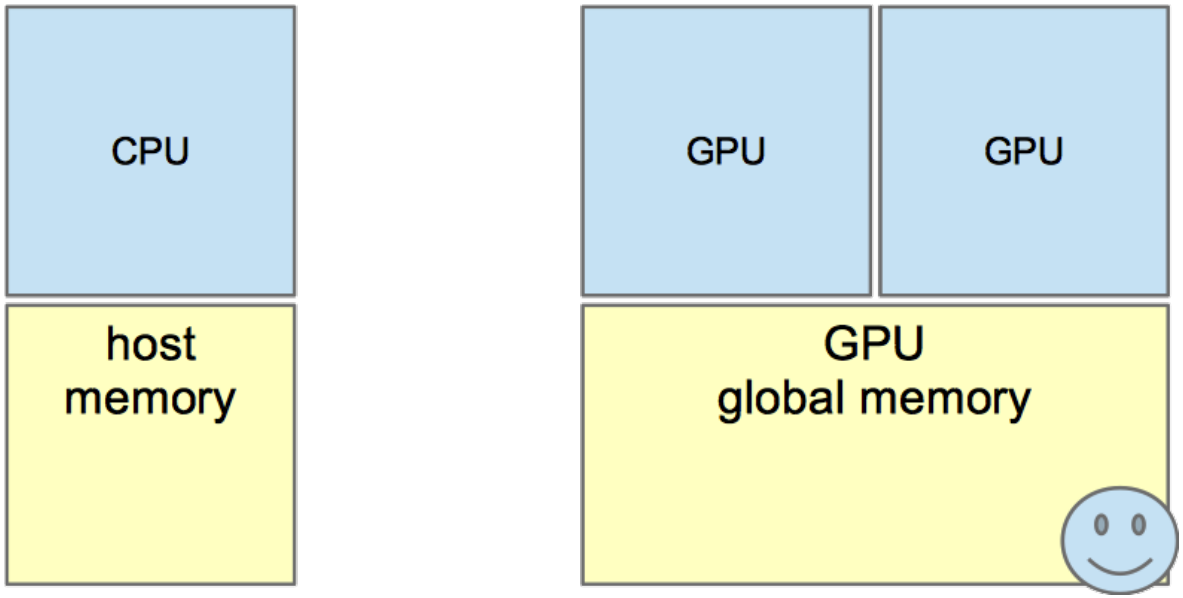
```
cudaFree( devResults );
```



Cleaning Up

- When we're completely done, we can reset the device.

```
cudaDeviceReset();
```



An Example

- I have a simple example to play with
- What does it do?
  - Creates a random array of integers
  - Copies it to the device
  - Runs a kernel on the array
  - Copies it back and prints some results
- Let's try:
  - Zeroing out every element
  - Adding one to every element
  - Replacing every element with the sum of all elements (oops)
  - Other ideas

What we Didn't See

- CUDA offers more than what we've covered
- Other regions of memory
  - Per-thread registers
  - Per-thread local memory
  - Per-block shared memory
  - Per-grid global memory
  - Per-grid constant memory
- Synchronization mechanisms
  - Barrier within a block: __syncthreads();
  - Atomic operations on shared memory